Introduction to CUDA Parallel Programming
CUDA平行計算導論

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

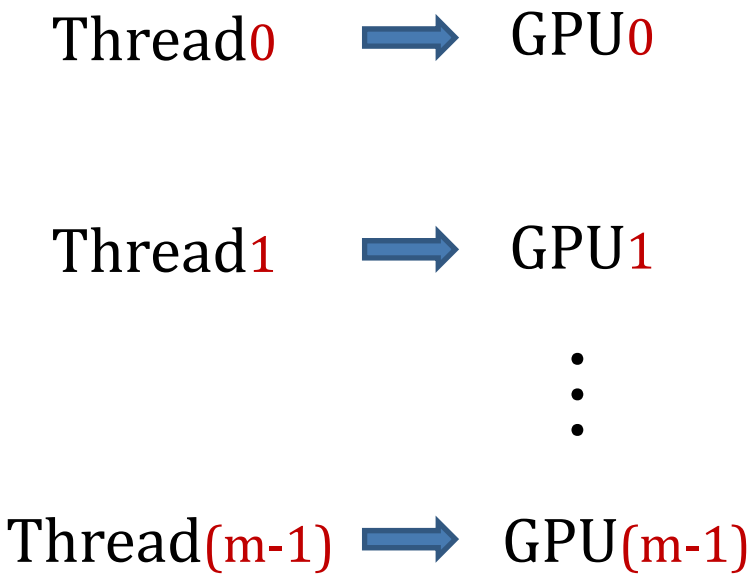
Professor Ting-Wai Chiu (趙挺偉)
Email: twchiu@phys.ntu.edu.tw
Physics Department
National Taiwan University

This lecture will cover:

- Solving Laplace Equation with multi-GPUs
- Heat Diffusion

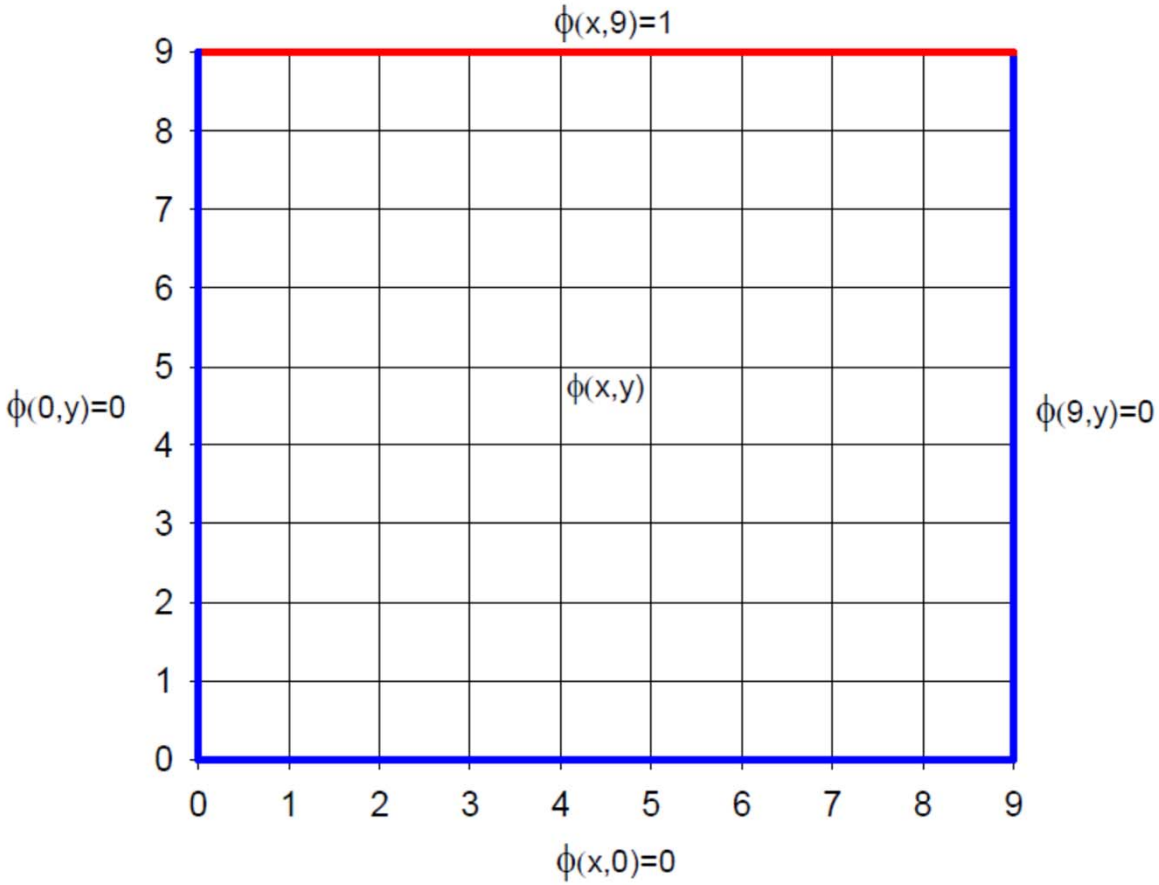
CUDA with multi-GPUs

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.



Solving Laplace Equation with multi-GPUs

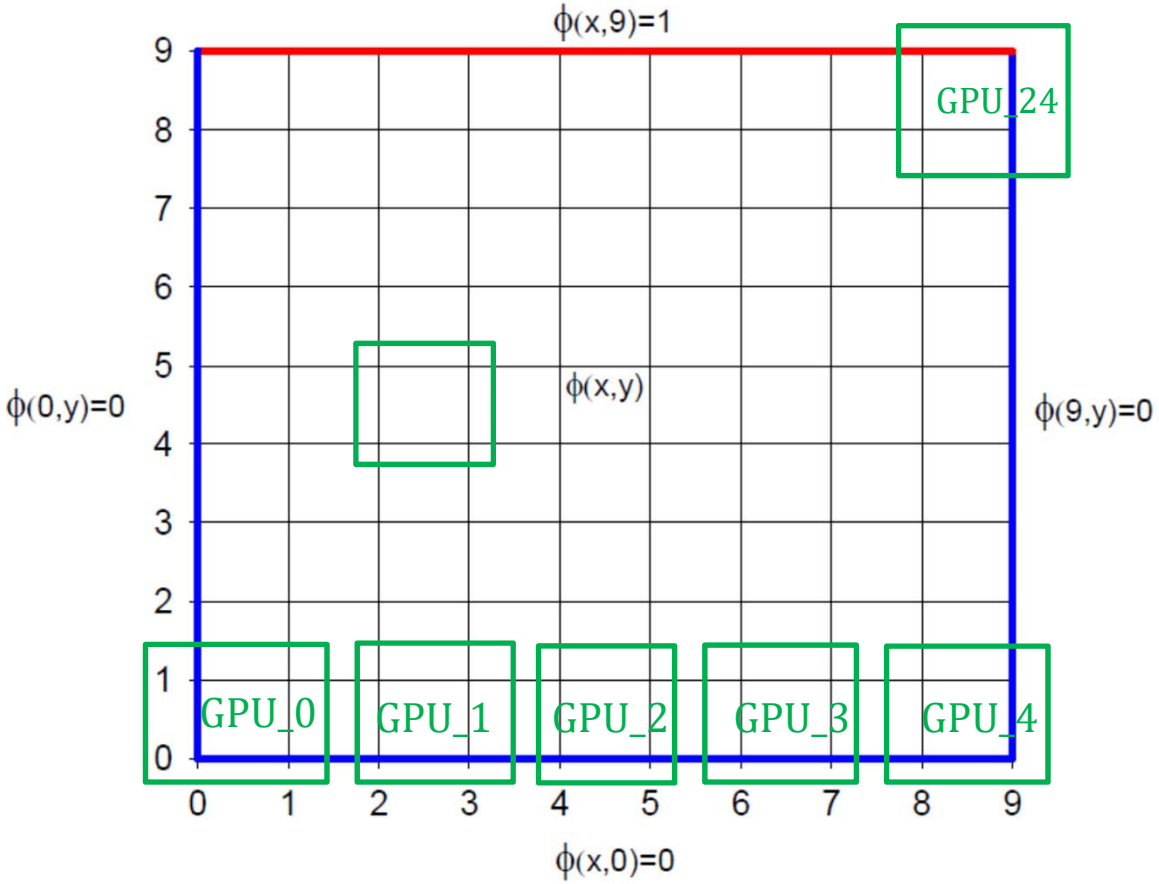
$$\phi(x,y) = \frac{1}{4}\left[\phi(x+1,y) + \phi(x-1,y) + \phi(x,y+1) + \phi(x,y-1)\right]$$



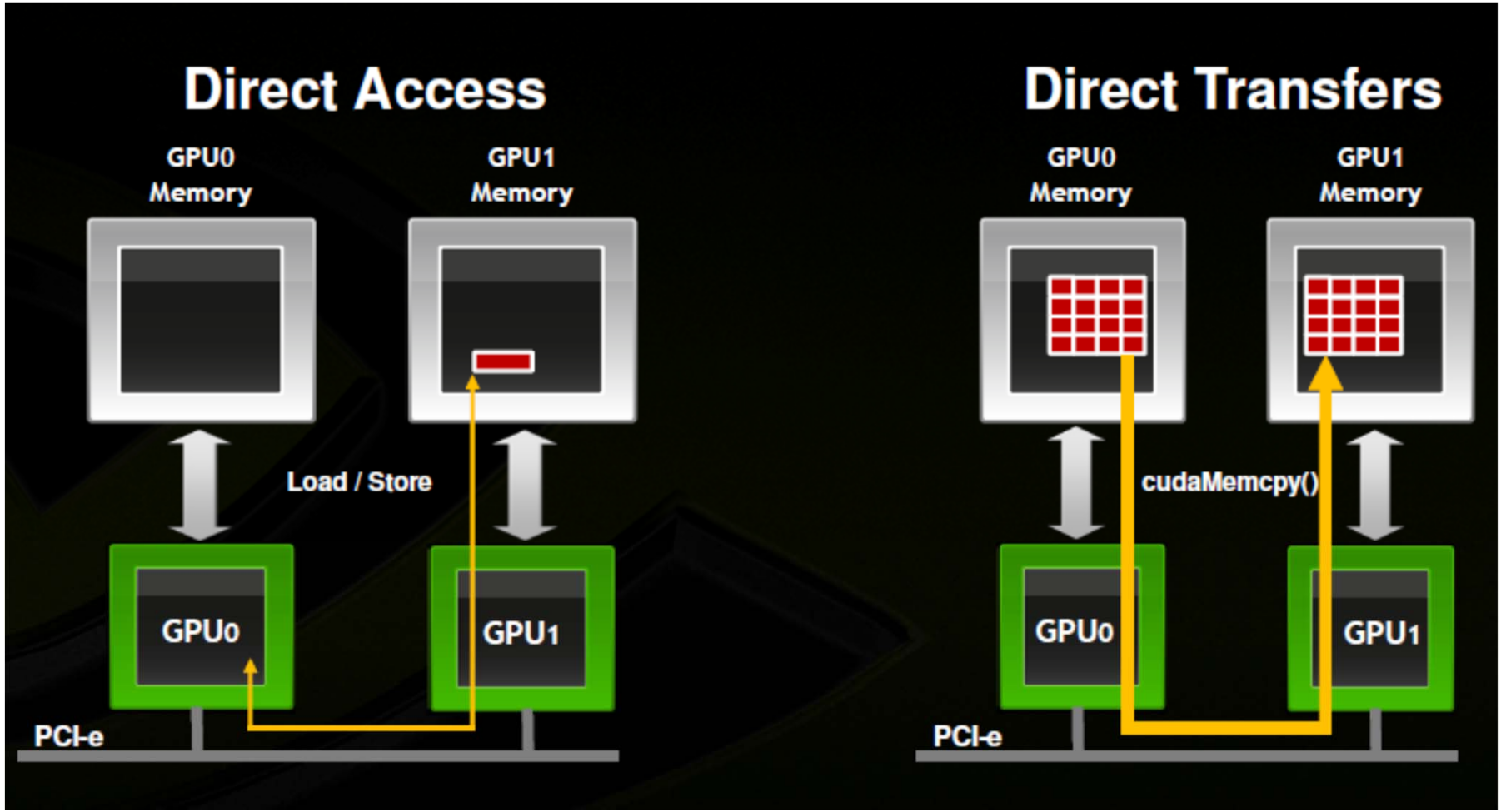
Solving Laplace Equation with multi-GPUs

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

With (NGx, NGy) GPUs, to decompose the 2D lattice into NGx*NGy equal partitions, e.g.,



To Enable P2P Communications between GPUs



P2P eliminates system memory allocation & copy overhead

Template to Enable P2P between Neighboring GPUs

```
...
#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
{ ...
  int cpuid_x, cpuid_y;
  cpu_thread_id = omp_get_thread_num();  // each thread gets its own id
  cpuid_x       = cpu_thread_id % NGx;   // x position
  cpuid_y       = cpu_thread_id / NGx;   // y position
  cudaSetDevice(Dev[cpu_thread_id]);     // set device

  int cpuid_r = (cpuid_x+1)%NGx + cpuid_y*NGx;        // right
  cudaDeviceEnablePeerAccess(Dev[cpuid_r],0);
  int cpuid_l = (cpuid_x+NGx-1)%NGx + cpuid_y*NGx;    // left
  cudaDeviceEnablePeerAccess(Dev[cpuid_l],0);
  int cpuid_t = cpuid_x + ((cpuid_y+1)%NGy)*NGx;      // top
  cudaDeviceEnablePeerAccess(Dev[cpuid_t],0);
  int cpuid_b = cpuid_x +((cpuid_y+NGy-1)%NGy)*NGx;   // bottom
  cudaDeviceEnablePeerAccess(Dev[cpuid_b],0);
  ...
} // end OpenMP
```

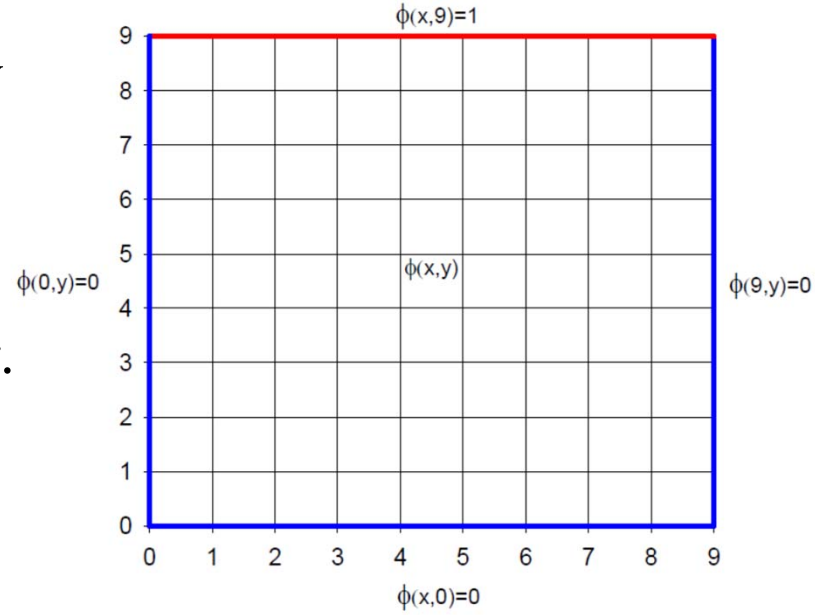
Solving Laplace Equation on the 2D Lattice

With any initial configuration satisfying the b.c., to iterate according to

$$\phi_{i+1}(x,y)=\frac{1}{4}\left[\phi_i(x+1,y)+\phi_i(x-1,y)+\phi_i(x,y+1)+\phi_i(x,y-1)\right],\ i=0,1,\cdots \text{until}$$

$$\left\|\nabla^2\phi\right\|=\sqrt{\sum_{x,y}\left|\nabla^2\phi(x,y)\right|^2}=\sqrt{\sum_{x,y}\left|\phi_{i+1}(x,y)-\phi_i(x,y)\right|^2}<\varepsilon \text{ (stopping criterion).}$$

In general, to solve a linear system $Ax=b$ by iterative method, the convergence of x is to satisfy the criterion $\|r\|\equiv\|Ax-b\|<\varepsilon$, where $r=Ax-b$ is called the residual vector. For the Laplace equation, $r=\nabla^2\phi$.



The Algorithm for Solving Laplace Equation on the 2D Lattice

To implement the iteration

$$\phi_{i+1}(x,y)=\frac{1}{4}\left[\phi_i(x+1,y)+\phi_i(x-1,y)+\phi_i(x,y+1)+\phi_i(x,y-1)\right],\ i=0,1,\cdots$$

it suffices to introduce two arrays, ϕ_{old} and ϕ_{new} and to switch their roles with a logical flag, e.g.,

$\{\text{flag} = \text{true},\ \phi_{\text{old}}=\phi_0\ ,\ \phi_{\text{new}}=\phi_1\},\ \{\text{flag} = \text{false},\ \phi_{\text{new}}=\phi_1\ ,\ \phi_{\text{old}}=\phi_2\},\cdots$

Then each thread can handle the updating at each site.

To determine whether the solution converges to the desired accuracy:

$$\left\|\nabla^2\phi\right\|=\sqrt{\sum_{x,y}\left|\nabla^2\phi(x,y)\right|^2}=\sqrt{\sum_{x,y}\left|\phi_{i+1}(x,y)-\phi_i(x,y)\right|^2}<\varepsilon\ \text{(stopping criterion).}$$

The error at each site $\left|\phi_{i+1}(x,y)-\phi_i(x,y)\right|^2$ can be computed by each thread.

Then their partial sum of each block can be obtained by parallel reduction.

Finally the sum of all errors of all blocks is obtained by the CPU (host).

Salient features of the Implementation

```
// Global variables declarations

float*  h_new;              // pointer to the working space
float*  h_old;              // pointer to the working space
float*  h_1;                // host field vectors (working space)
float*  h_2;                // host field vectors (working space)
float*  h_C;                // sum of diff*diff of each block
float*  g_new;              // GPU solution back to the host
float** d_1;                // device field vectors (working space)
float** d_2;                // device field vectors (working space)
float** d_C;                // sum of diff*diff of each block

int     MAX=1000000;        // maximum iterations
double  eps=1.0e-10;        // stopping criterion
```

Salient features of the Implementation (cont)

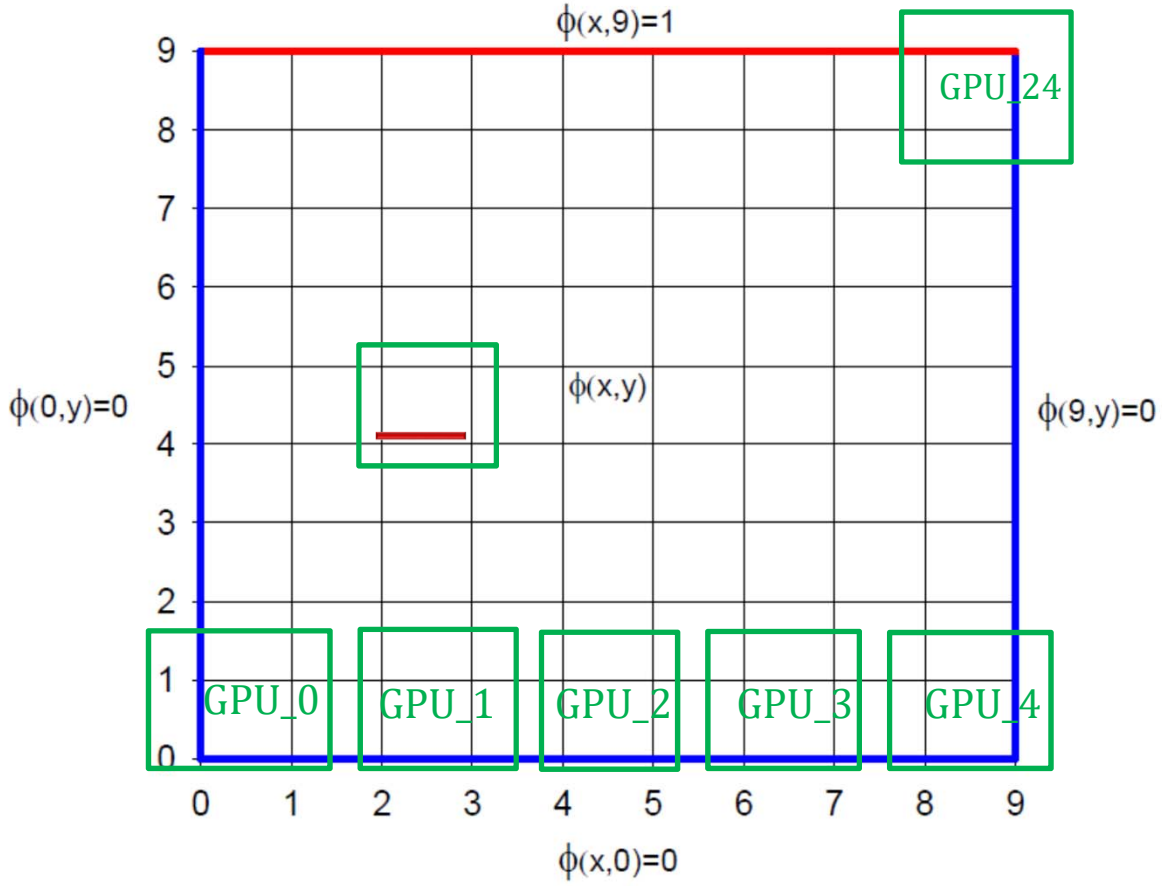
```
//  Allocate field vector in host memory

int N    = Nx*Ny;
int size = N*sizeof(float);
int sb   = bx*by*sizeof(float);
h_1   = (float*)malloc(size);
h_2   = (float*)malloc(size);
h_C   = (float*)malloc(sb);
g_new = (float*)malloc(size);

// Initialize the field vector with boundary conditions
memset(h_1, 0, size);
memset(h_2, 0, size);
for (int x=0; x<Nx; x++) {
  h_1[x+Nx*(Ny-1)]=1.0;
  h_2[x+Nx*(Ny-1)]=1.0;
}
//  Allocate working space for GPUs
sm = tx*ty*sizeof(float);  // size of the shared memory

d_1 = (float **)malloc(NGPU*sizeof(float *));
d_2 = (float **)malloc(NGPU*sizeof(float *));
d_C = (float **)malloc(NGPU*sizeof(float *));
```

Salient features of the Implementation (cont)

```
 omp_set_num_threads(NGPU);
#pragma omp parallel private(cpu_thread_id)
{
  // enabling P2P between neighboring GPUs (see the template)

  // Allocate vectors in device memory
  cudaMalloc((void**)&d_1[cpu_thread_id], size/NGPU);
  cudaMalloc((void**)&d_2[cpu_thread_id], size/NGPU);
  cudaMalloc((void**)&d_C[cpu_thread_id], sb/NGPU);

  // Copy vectors from host memory to device memory
  for (int j=0; j < Ly; j++) {      // Lx,Ly: lattice size in each GPU
    float *h, *d;
    h = h_1 + cpuid_x*Lx + (cpuid_y*Ly+j)*Nx;
    d = d_1[cpu_thread_id] + j*Lx;
    cudaMemcpy(d, h, Lx*sizeof(float), cudaMemcpyHostToDevice);
  }
  for (int j=0; j < Ly; j++) {
    float *h, *d;
    h = h_2 + cpuid_x*Lx + (cpuid_y*Ly+j)*Nx;
    d = d_2[cpu_thread_id] + j*Lx;
    cudaMemcpy(d, h, Lx*sizeof(float), cudaMemcpyHostToDevice);
  }

  #pragma omp barrier
}             // end OpenMP
```

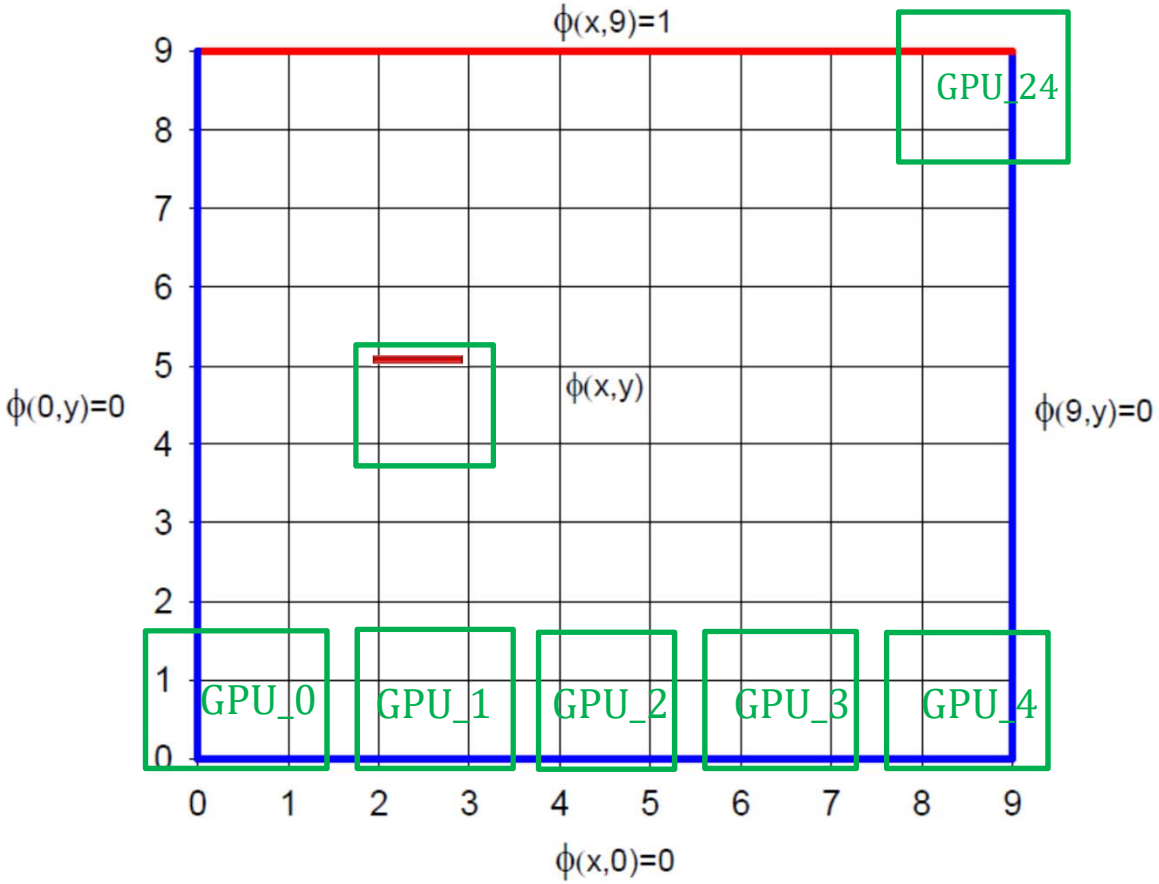
Salient features of the Implementation (cont)

With (NGx, NGy) GPUs, to decompose the 2D lattice into NGx*NGy equal partitions, e.g.,



Salient features of the Implementation (cont)

With (NGx, NGy) GPUs, to decompose the 2D lattice into NGx*NGy equal partitions, e.g.,



Salient features of the Implementation (cont)

```
while ((error > eps) && (iter < MAX)) {
  #pragma omp parallel private(cpu_thread_id)
  {
    int cpuid_x, cpuid_y;
    cpu_thread_id = omp_get_thread_num();
    cpuid_x = cpu_thread_id % NGx; cpuid_y = cpu_thread_id/NGx;
    cudaSetDevice(Dev[cpu_thread_id]);

    float **d_old, **d_new;
    float *dL_old, *dR_old, *dT_old, *dB_old, *d0_old, *d0_new;
    d_old  = (flag == true) ? d_1 : d_2;
    d_new  = (flag == true) ? d_2 : d_1;
    d0_old = d_old[cpu_thread_id];
    d0_new = d_new[cpu_thread_id];
    dL_old = (cpuid_x == 0) ? NULL : d_old[cpuid_x-1+cpuid_y*NGx];
    dR_old = (cpuid_x == NGx-1) ? NULL : d_old[cpuid_x+1+cpuid_y*NGx];
    dB_old = (cpuid_y == 0    ) ? NULL : d_old[cpuid_x+(cpuid_y-1)*NGx];
    dT_old = (cpuid_y == NGy-1) ? NULL : d_old[cpuid_x+(cpuid_y+1)*NGx];
    laplacian<<<blocks,threads,sm>>>(d0_old, dL_old, dR_old, dB_old,
                                     dT_old, d0_new, d_C[cpu_thread_id]);
    cudaDeviceSynchronize();
    cudaMemcpy(h_C+bx*by/NGPU*cpu_thread_id, d_C[cpu_thread_id], sb/NGPU,
                      cudaMemcpyDeviceToHost);
  } // end OpenMP
  error = 0.0;
  for(int i=0; i<bx*by; i++) error = error + h_C[i];
  error = sqrt(error);
  iter++; flag = !flag;
}
```

Salient features of the Implementation (cont)

```
//  Copy result from device memory to host memory

#pragma omp parallel private(cpu_thread_id)
{
  int cpuid_x, cpuid_y;
  cpu_thread_id = omp_get_thread_num();
  cpuid_x       = cpu_thread_id % NGx;
  cpuid_y       = cpu_thread_id / NGx;
  cudaSetDevice(Dev[cpu_thread_id]);

  float* d_new = (flag == true) ? d_2[cpu_thread_id] : d_1[cpu_thread_id];
  for (int i=0; i < Ly; i++) {
      float *g, *d;
      g = g_new + cpuid_x*Lx + (cpuid_y*Ly+i)*Nx;
      d = d_new + i*Lx;
      cudaMemcpy(g, d, Lx*sizeof(float), cudaMemcpyDeviceToHost);
  }

  cudaFree(d_1[cpu_thread_id]);
  cudaFree(d_2[cpu_thread_id]);
  cudaFree(d_C[cpu_thread_id]);

} // end OpenMP
```

```
__global__ void
laplacian(float* phi0_old, float* phiL_old, float* phiR_old, float* phiB_old,
          float* phiT_old, float* phi0_new, float* C)
{
    extern __shared__ float cache[];
    float  t, l, c, r, b;     // top, left, center, right, bottom
    float  diff;
    int    site, skip;
    int Lx = blockDim.x*gridDim.x;    // sub-lattice size in each GPU
    int Ly = blockDim.y*gridDim.y;
    int x  = blockDim.x*blockIdx.x + threadIdx.x;  // position in the sub-lattice
    int y  = blockDim.y*blockIdx.y + threadIdx.y;
    int cacheIndex = threadIdx.x + threadIdx.y*blockDim.x;
    site = x + y*Lx;
    skip = 0; diff = 0.0; b = 0.0; l = 0.0; r = 0.0; t = 0.0;
    c = phi0_old[site];

    if (x == 0) {     // left boundary of the sub-lattice
      if (phiL_old != NULL) {  // if not the left boundary of the entire lattice
        l = phiL_old[(Lx-1)+y*Lx]; r = phi0_old[site+1]; }
      else
        skip = 1;     // if it is the left boundary of the entire lattice
    }
    else if (x == Lx-1) {
      if(phiR_old != NULL) {
        l = phi0_old[site-1]; r = phiR_old[y*Lx]; }
      else
        skip = 1;
    }
    else {
      l = phi0_old[site-1]; r = phi0_old[site+1];} // continue in next page
```

```
      if (y == 0) {
        if (phiB_old != NULL) {
          b = phiB_old[x+(Ly-1)*Lx]; t = phi0_old[site+Lx]; }
        else
          skip = 1;
      }
      else if (y == Ly-1) {
        if (phiT_old != NULL) {
          b = phi0_old[site-Lx]; t = phiT_old[x]; }
        else
          skip = 1;
       }
       else {
         b = phi0_old[site-Lx]; t = phi0_old[site+Lx];
       }
       if (skip == 0) {
          phi0_new[site] = 0.25*(b+l+r+t);
          diff = phi0_new[site]-c;
       }
     }
     // each thread saves its error estimate to the shared memory
     cache[cacheIndex]=diff*diff;
     __syncthreads();

     // parallel reduction in each block
     ...
     // save the partial sum of each block to C
     ...
}          //  complete code in twqcd80:/home/cuda_lecture_2021/laplace2D_NGPU
```

Heat Diffusion

$$\nabla^2 u(x,y,z,t) = \frac{1}{c^2}\frac{\partial u}{\partial t}$$

$$c^2 = K/\sigma\rho$$

u : temperature

K : thermal conductivity

σ : specific heat

ρ : density

In 2D, $u_{i,j}^{k} \equiv u\left(x,y,t\right),\ \ x = i\Delta,\ \ y = j\Delta,\ \ t = k\delta t$

$$\frac{u_{i+1,j}^{k} + u_{i-1,j}^{k} - 2u_{ij}^{k}}{\Delta^2} + \frac{u_{ij+1}^{k} + u_{ij-1}^{k} - 2u_{ij}^{k}}{\Delta^2} = \frac{1}{c^2}\frac{1}{\delta t}\left(u_{ij}^{k+1} - u_{ij}^{k}\right)$$

$$\frac{1}{\Delta^2}\left[u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k}\right] = \frac{1}{c^2}\frac{1}{\delta t}u_{i,j}^{k+1} + \left(\frac{4}{\Delta^2} - \frac{1}{c^2}\frac{1}{\delta t}\right)u_{i,j}^{k}$$

$$u_{i,j}^{k+1} = \frac{c^2\delta t}{\Delta^2}\left[u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{ij+1}^{k} + u_{ij-1}^{k}\right] + \left(1 - \frac{4c^2\delta t}{\Delta^2}\right)u_{ij}^{k}$$

$$= \frac{\omega}{4}\left[u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{ij+1}^{k} + u_{ij-1}^{k}\right] + (1-\omega)u_{ij}^{k}, \quad \omega \equiv \frac{4c^2\delta t}{\Delta^2}$$

For steady state conduction $\dfrac{\partial u}{\partial t} = 0$

Diffusion Eq. $\Rightarrow$ Laplace Eq. $\nabla^2 u = 0$

But we can still use the time-dependent solution to obtain the steady state solution by assigning $t = k \cdot \delta t$ as the computer time and k as the index of the iteration.

$$u_{i,j}^{k+1} = \frac{\omega}{4}\left[u_{i,j+1}^{k} + u_{i,j-1}^{k} + u_{i+1,j}^{k} + u_{i-1,j}^{k}\right] + (1-\omega)u_{i,j}^{k}$$

$$\frac{c^2 \delta t}{\Delta^2} = \left(\frac{\omega}{4}\right), \quad 1 < \omega < 2$$

The optimal value of ω is

$$\omega_{\text{opt}} = \frac{4}{2 + \sqrt{4 - \left(\cos\frac{\pi}{N_x} + \cos\frac{\pi}{N_y} \right)^2}}$$