Introduction to CUDA Parallel Programming
CUDA平行計算導論

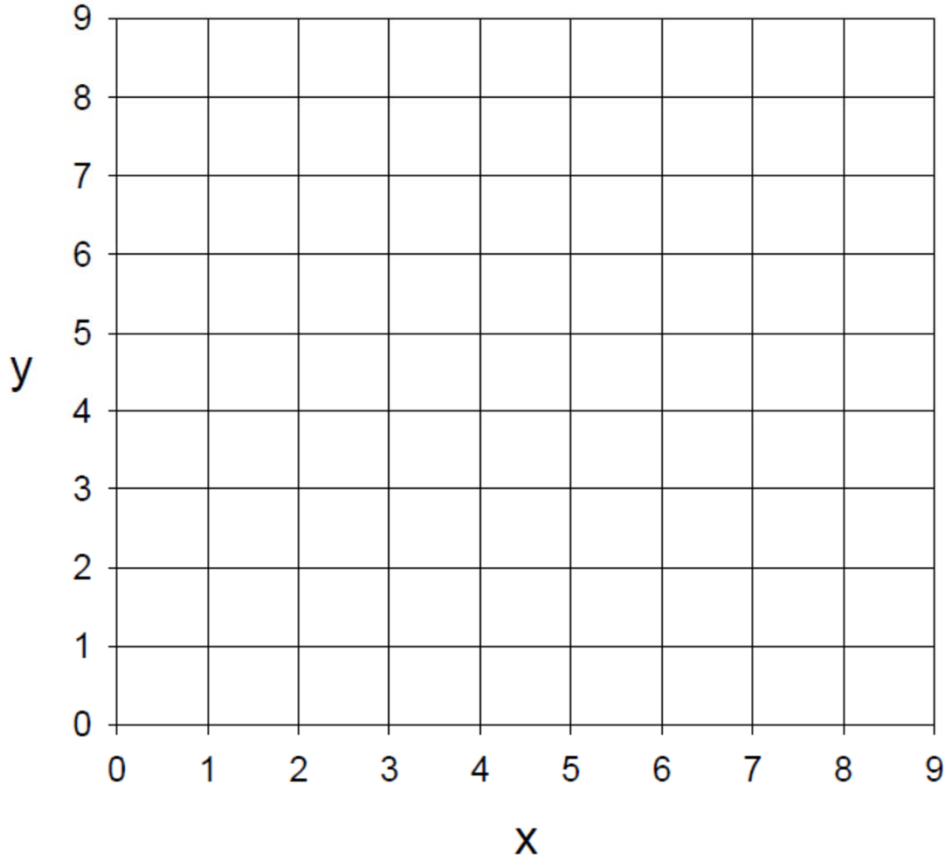
https://ceiba.ntu.edu.tw/1092Phys8061_CUDA

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

This lecture will cover:

- To Solve Laplace and Poisson Equations on 2D Lattice with Boundary Conditions
- Texture Memory

Field Theory on the 2D Lattice (1)



Field Theory on the 2D Lattice (2)

$\phi(x,y)$: field variable at the site (x,y),

$x = 0,\ 1,\ \cdots, N_x - 1,$

$y = 0,\ 1,\ \cdots, N_y - 1,$

$i = x + N_x y = 0,\ 1,\ \cdots, N_x N_y - 1,$

Thus the field variables can be labelled as $\phi(i)$, field vector

$$\begin{pmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{N-2} \\ \phi_{N-1} \end{pmatrix}, \quad N = N_x N_y$$

Field Theory on the 2D Lattice (3)

Consider the classical field theory of electrostic potential.

$\phi(x,y)$: field variable at the site (x,y) is a real number

$x = 0,\ 1,\ \cdots, N_x - 1,$

$y = 0,\ 1,\ \cdots, N_y - 1,$

$i = x + N_x y = 0,\ 1,\ \cdots, N_x N_y - 1,$

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2}$$

$$\to \frac{\phi(x+a,y) + \ \phi(x-a,y) - 2\phi(x,y)}{a^2} + \frac{\phi(x,y+a) + \ \phi(x,y-a) - 2\phi(x,y)}{a^2}$$

$$= \frac{\phi(x+a,y) + \ \phi(x-a,y) + \phi(x,y+a) + \ \phi(x,y-a) - 4\phi(x,y)}{a^2} \equiv D\phi$$

Solving the Poisson equation $\nabla^2\phi = \rho$ amounts to solving $D\phi = \rho$

Laplace Equation on the 2D Lattice

For $\rho = 0$, the Poisson's Eq. becomes the Laplace Eq. $\nabla^2\phi = 0$

On the 2D lattice,

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0$$

$$\rightarrow \frac{\phi(x+a,y) + \phi(x-a,y) + \phi(x,y+a) + \phi(x,y-a) - 4\phi(x,y)}{a^2} = 0$$

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

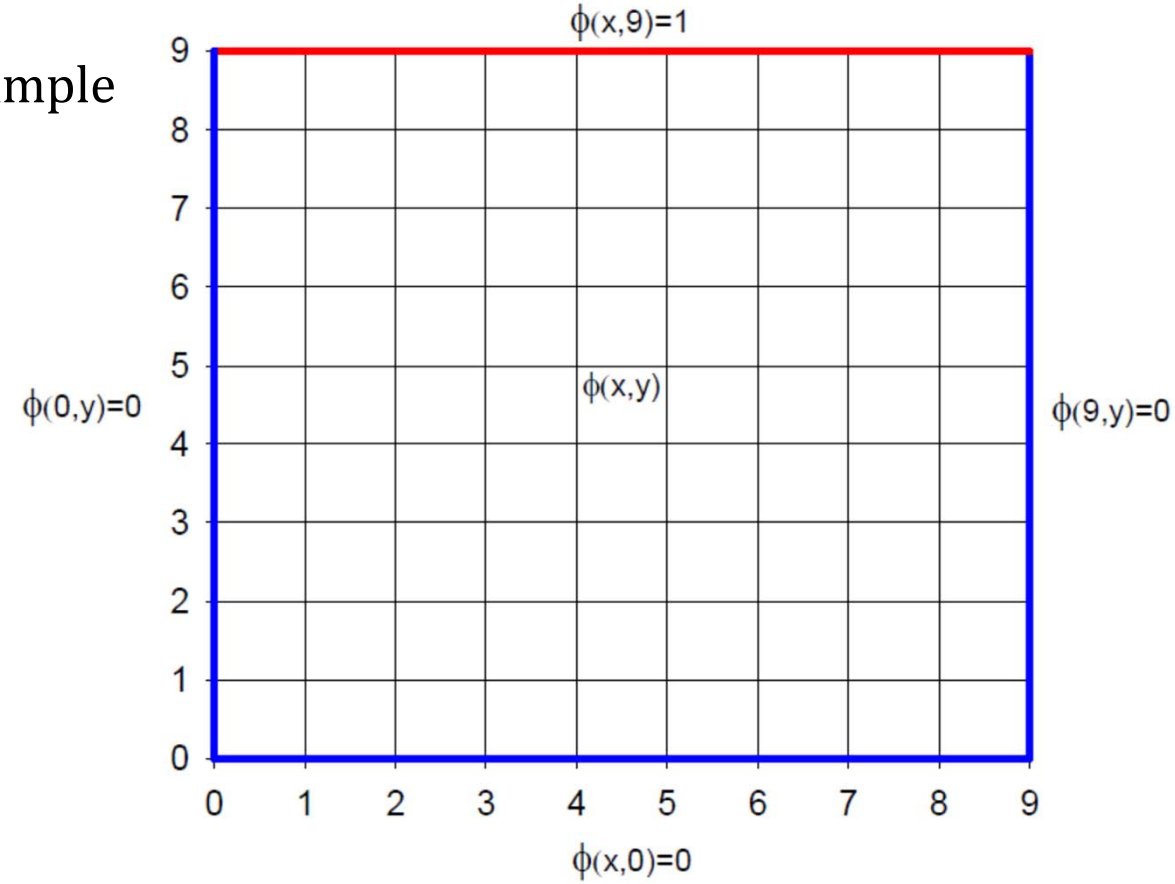
In the lattice unit $a = 1$,

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

To Solve Laplace Equation on the 2D Lattice with Boundary Condition (1)

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

A simple example



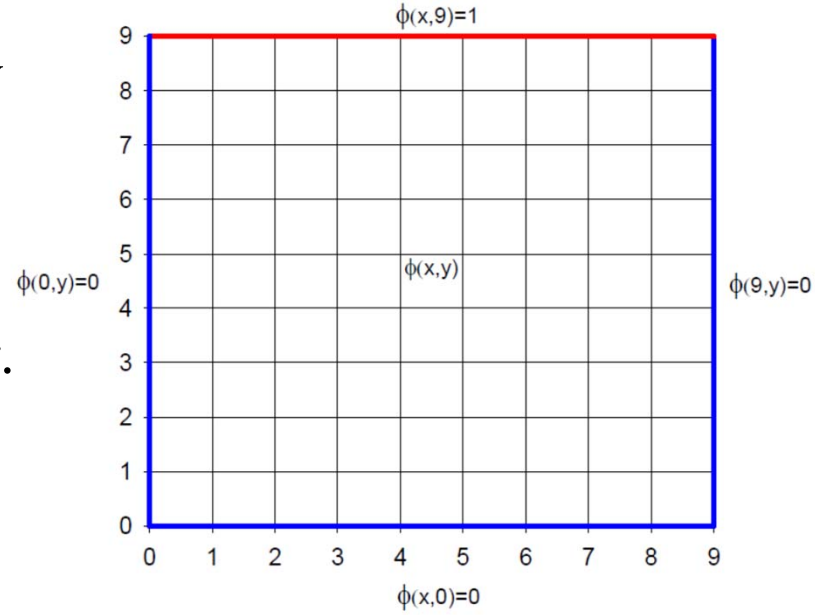
To Solve Laplace Equation on the 2D Lattice with Boundary Condition (2)

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 $\left\|r\right\|\equiv\left\|Ax-b\right\|<\varepsilon$, where $r=Ax-b$ is called the residual vector. For the Laplace equation, $r=\nabla^2\phi$.



GPU Kernel for Solving Laplace Equation on the 2D Lattice with Boundary Conditions

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).

(See the sample code in twqcd80:/home/cuda_lecture_2021/laplaceTex/laplace.cu)

Poisson Equation on the 2D Lattice

Consider the Poisson equation $\nabla^2\phi = \rho$

with a point charge q at (x_0, y_0), $\rho(x,y) = q\delta(x-x_0)\delta(y-y_0)$

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = \rho(x,y)$$

$$\rightarrow \frac{\phi(x+a,y) + \phi(x-a,y) + \phi(x,y+a) + \phi(x,y-a) - 4\phi(x,y)}{a^2} = q\frac{\delta_{x,x_0}}{a}\frac{\delta_{y,y_0}}{a}$$

$$\rightarrow \phi(x,y) = \frac{1}{4}\left[\phi(x+a,y) + \phi(x-a,y) + \phi(x,y+a) + \phi(x,y-a) - q\delta_{x,x_0}\delta_{y,y_0}\right]$$

In the lattice unit $a = 1$,

$$\phi(x,y) = \frac{1}{4}\left[\phi(x+1,y) + \phi(x-1,y) + \phi(x,y+1) + \phi(x,y-1) - q\delta_{x,x_0}\delta_{y,y_0}\right]$$

To Solve Poisson Equation on the 2D Lattice with Boundary Condition (2)

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

$$\phi_{i+1}(x,y)=\frac{1}{4}\Big[\phi_i(x+1,y)+\phi_i(x-1,y)+\phi_i(x,y+1)+\phi_i(x,y-1)-q\delta_{x,x_0}\delta_{y,y_0}\Big],$$

$i=0,1,\cdots$ until

$$\left\|\nabla^2\phi-\rho\right\| = \sqrt{\sum_{x,y}\left|\nabla^2\phi(x,y)-\rho(x,y)\right|^2}$$

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

Thus the kernel for solving the Poisson equation can be obtained from that of solving the Laplace equation with obvious modifications.

Solving Laplace Equation on the 2D Lattice with Boundary Conditions – CPU code (1)

```
int main(void)
{
    int Nx,Ny;    // lattice size
    ...
    int size = Nx*Ny*sizeof(float);
    h_new = (float*)malloc(size);
    h_old = (float*)malloc(size);
    memset(h_old, 0, size);   // set the values to zero
    memset(h_new, 0, size);

    for(int x=0; x<Nx; x++) { // impose the boundary conditions
      h_new[x+Nx*(Ny-1)]=1.0;
      h_old[x+Nx*(Ny-1)]=1.0;
    }
    double error = 10*eps;  // any value bigger eps is OK
    int iter = 0;           // counter for iterations
    volatile bool flag = true;

    float t, l, r, b;       // top, left, right, bottom
    double diff;
    int site, ym1, xm1, xp1, yp1;
```

Solving Laplace Equation on the 2D Lattice with Boundary Conditions – CPU code (2)

```
while ( (error > eps) && (iter < MAX) ) {
 if(flag) {
   error = 0.0;
   for(int y=0; y<Ny; y++) {
   for(int x=0; x<Nx; x++) {
     if(x==0 || x==Nx-1 || y==0 || y==Ny-1) {
     }            // do nothing to the boundaries
     else {
       site = x+y*Nx;
       xm1 = site - 1;    // x-1
       xp1 = site + 1;    // x+1
       ym1 = site - Nx;   // y-1
       yp1 = site + Nx;   // y+1
       b = h_old[ym1];    // bottom
       l = h_old[xm1];    // left
       r = h_old[xp1];    // right
       t = h_old[yp1];    // top
       h_new[site] = 0.25*(b+l+r+t);
       diff = h_new[site]-h_old[site];
       error = error + diff*diff;
     }
   }
   }
 }
```

Solving Laplace Equation on the 2D Lattice with Boundary Conditions – CPU code (3)

```
      else {
        error = 0.0;
        for(int y=0; y<Ny; y++) {
        for(int x=0; x<Nx; x++) {
          if(x==0 || x==Nx-1 || y==0 || y==Ny-1) {
          }
          else {
            site = x+y*Nx;
            xm1 = site - 1;    // x-1
            xp1 = site + 1;    // x+1
            ym1 = site - Nx;   // y-1
            yp1 = site + Nx;   // y+1
            b = h_new[ym1];
            l = h_new[xm1];
            r = h_new[xp1];
            t = h_new[yp1];
            h_old[site] = 0.25*(b+l+r+t);
            diff = h_new[site]-h_old[site];
            error = error + diff*diff;
          }
        }
        }
      }
      flag = !flag;   iter++;   error = sqrt(error);
    }// Complete code at twqcd80:/home/cuda_lecture_2021/laplaceTex/laplace_cpu.cu
```

Solving Laplace Equation on the 2D Lattice with Boundary Conditions – GPU code (1)

```
__global__ void laplacian(float* phi_old, float* phi_new, float* C, bool flag)
{
    extern __shared__ float cache[];
    float  t, l, r, b;     // top, left, right, bottom
    float  diff=0.0;
    int    site, ym1, xm1, xp1, yp1;
    int Nx = blockDim.x*gridDim.x;
    int Ny = blockDim.y*gridDim.y;
    int x = blockDim.x*blockIdx.x + threadIdx.x;
    int y = blockDim.y*blockIdx.y + threadIdx.y;
    int cacheIndex = threadIdx.x + threadIdx.y*blockDim.x;
    site = x + y*Nx;
    if((x == 0) || (x == Nx-1) || (y == 0) || (y == Ny-1) )
      diff = 0.0;
    else {
      xm1 = site - 1; xp1 = site + 1; ym1 = site - Nx; yp1 = site + Nx;
      if(flag) {
        b = phi_old[ym1]; l = phi_old[xm1]; r = phi_old[xp1]; t = phi_old[yp1];
        phi_new[site] = 0.25*(b+l+r+t);
      }
      else {
        b = phi_new[ym1]; l = phi_new[xm1]; r = phi_new[xp1]; t = phi_new[yp1];
        phi_old[site] = 0.25*(b+l+r+t);
      }
      diff = phi_new[site]-phi_old[site];
    }
```

Solving Laplace Equation on the 2D Lattice with Boundary Conditions – GPU code (2)

```
    cache[cacheIndex]=diff*diff;
    __syncthreads();

    int ib = blockDim.x*blockDim.y/2;    // perform parallel reduction
    while (ib != 0) {
      if(cacheIndex < ib)
        cache[cacheIndex] += cache[cacheIndex + ib];
      __syncthreads();
      ib /=2;
    }
    int blockIndex = blockIdx.x + gridDim.x*blockIdx.y;
    if(cacheIndex == 0)  C[blockIndex] = cache[0];
}

int main(void)
{  ...
   int sm = tx*ty*sizeof(float);   // size of the shared memory in each block
   while ( (error > eps) && (iter < MAX) ) {
     laplacian<<<blocks,threads,sm>>>(d_old, d_new, d_C, flag);
     cudaMemcpy(h_C, d_C, sb, cudaMemcpyDeviceToHost);
     error = 0.0;
     for(int i=0; i<bx*by; i++) error += h_C[i];
     error = sqrt(error);  iter++;  flag = !flag;
   }
    ...
}  // Complete code at twqcd80:/home/cuda_lecture_2021/laplaceTex/laplace.cu
```

Texture Memory (1)

The texture memory can be regarded as a read-only cache to enhance the speed of reading data from the device memory, espeically for the data in a local region of the physical space-time but separated far apart in the linear memory.

For example, $\phi(x, y)$ and $\phi(x, y+1)$ are neighbors in the physical space, but their values are stored in the linear memory at positions $x + yN_x$ and $x + (y+1)N_x$ with separation N_x.

Texture Memory (2)

The texture memory can be regarded as a read-only cache to enhance the speed of reading data from the device memory, espeically for the data in a local region of the physical space-time but separated far apart in the linear memory. For example, $\phi(x, y)$ and $\phi(x, y+1)$ are neighbors in the physical space, but their values are stored in the linear memory at positions $x + yN_x$ and $x + (y+1)N_x$ with separation N_x.

Since all physical laws are local in the spacetime, which can be written as partial differential eqs. (e.g., Maxwell eqs., Schrödinger eq., $\cdots$), texture memory are useful for optimization of CUDA codes in science and engineering.

Basic Steps of Setting up Texture Memory

```
__align__(8) texture<float>  texOld;  // declare the texture globally
__align__(8) texture<float>  texNew;

int main(void){
 ...
 cudaMalloc((void**)&d_new, size);
 cudaMalloc((void**)&d_old, size);
 cudaBindTexture(NULL, texOld, d_old, size);  // bind the texture to
 cudaBindTexture(NULL, texNew, d_new, size);  // corresponding array
 ...                                          // in the device memory
 cudaFree(d_new); cudaFree(d_old);
 cudaUnbindTexture(texOld);          // unbind the texture at the end
 cudaUnbindTexture(texNew);
 ...
}
__global__ void laplacian(float* phi_old, float* phi_new, ...){
    ...
    b = tex1Dfetch(texOld, ym1);     // read d_old via texOld
    ...
    b = tex1Dfetch(texNew, ym1);
    ...
}
// complete code at twqcd80:/home/cuda_lecture_2021/laplaceTex/laplaceTex.cu
```