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

Vector Addition (1)

$$C = A + B$$

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-2} \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-2} \\ a_{N-1} \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{N-2} \\ b_{N-1} \end{pmatrix}$$

Vector Addition (2)

$$C \quad = \quad A \quad + \quad B$$

$$\begin{array}{l} \text{Block0} \left\{ \vdots \right. \\ \\ \text{Block1} \left\{ \vdots \right. \\ \quad \vdots \\ \text{Block(m-1)} \left\{ \vdots \right. \end{array} \quad \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-2} \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-2} \\ a_{N-1} \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{N-2} \\ b_{N-1} \end{pmatrix}$$

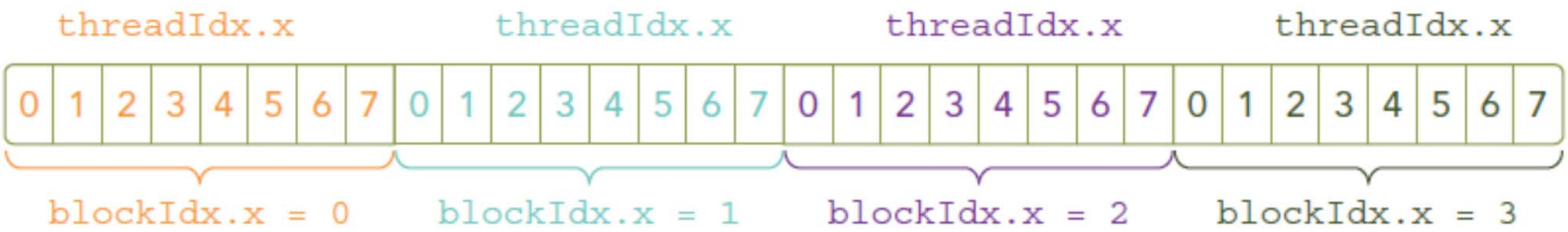
Use 1D grid and 1D blocks

vector index: i = blockDim.x * blockIdx.x + threadIdx.x

Vector Addition (3)

$$C = A + B$$

$$\begin{array}{r} \text{Block0}\left\{\vdots\right. \\ \\ \text{Block1}\left\{\vdots\right. \\ \vdots \\ \text{Block(m-1)}\left\{\vdots\right. \end{array}
\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-2} \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-2} \\ a_{N-1} \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{N-2} \\ b_{N-1} \end{pmatrix}$$



Vector Addition (4)

To make the computation more intensive, change it to the addition of the inverses of two vectors.

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

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-2} \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} 1/a_0 \\ 1/a_1 \\ 1/a_2 \\ \vdots \\ 1/a_{N-2} \\ 1/a_{N-1} \end{pmatrix} + \begin{pmatrix} 1/b_0 \\ 1/b_1 \\ 1/b_2 \\ \vdots \\ 1/b_{N-2} \\ 1/b_{N-1} \end{pmatrix}$$

Vector Addition (5)

```
int main(void)     // host (CPU) code fragment
{
   int  N=40960000;

   // Allocate vectors in CPU (host) memory
   h_A = (float*)malloc(N*sizeof(float));
   h_B = (float*)malloc(N*sizeof(float));
   h_C = (float*)malloc(N*sizeof(float));

   // Initialize vectors h_A and h_B with random numbers
   RandomInit(h_A, N);
   RandomInit(h_B, N);

   for (int i = 0; i < N; i++)
      h_C[i] = 1.0/h_A[i] + 1.0/h_B[i];
}
```

Vector Addition (6)

```
// Device (GPU) code fragment

__global__ void VecAdd(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
      C[i] = 1.0/A[i] + 1.0/B[i];
    __syncthreads();   //acts as a barrier at which all threads in a block
                       // must wait before any is allowed to proceed.
}
```

Vector Addition (7)

```
// Device (GPU) code fragment

__global__ void VecAdd(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
      C[i] = 1.0/A[i] + 1.0/B[i];
    __syncthreads();
}

// Host code fragment
int main(void) {
   ...
  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
   ...
}
```

Vector Addition (8)

```
// Host code fragment
int main(void) {
   ...
// Allocate vectors in device memory
  cudaMalloc((void**)&d_A, N*sizeof(float));
  cudaMalloc((void**)&d_B, N*sizeof(float));
  cudaMalloc((void**)&d_C, N*sizeof(float));

// Copy vectors from host memory to device memory
  cudaMemcpy(d_A, h_A, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, N*sizeof(float), cudaMemcpyHostToDevice);
  ...
   VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
  ...
}
```

// See the complete codes at
twqcd80:/home/cuda_lecture_2021/vecAdd_1GPU

Timing the GPU/CPU code in CUDA

➢ Timing the GPU code

```
cudaEvent_t start, stop;
    // do some work on the CPU

cudaEventCreate(&start);        //create and start the timer
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );
    // do some work on the GPU

cudaEventRecord( stop, 0 );      // stop the timer
cudaEventSynchronize( stop );
float elapsedTime;
cudaEventElapsedTime( &elapsedTime, start, stop ) );
printf( "Time for GPU: %3.1f ms\n", elapsedTime );
cudaEventDestroy( start );       // destroy the timer
cudaEventDestroy( stop );
```

Guidelines for developing CUDA code

- Documentation is essential for maintaining your codes, not just for yourself, but also for all potential users.

Guidelines for developing CUDA code

- Documentation is essential for maintaining your codes, not just for yourself, but also for all potential users.
- Compare the GPU and CPU results, using a problem with a feasible size for CPU.

Guidelines for developing CUDA code

- Documentation is essential for maintaining your codes, not just for yourself, but also for all potential users.
- Compare the GPU and CPU results, using a problem with a feasible size for CPU.
- Timing the GPU and CPU codes.

Guidelines for developing CUDA code

- Documentation is essential for maintaining your codes, not just for yourself, but also for all potential users.
- Compare the GPU and CPU results, using a problem with a feasible size for CPU.
- Timing the GPU and CPU codes.
- To determine the optimal block size for each kernel. This should be repeated for each kind of GPU, and for each size of the problem.

Guidelines for developing CUDA code

- Documentation is essential for maintaining your codes, not just for yourself, but also for all potential users.
- Compare the GPU and CPU results, using a problem with a feasible size for CPU.
- Timing the GPU and CPU codes.
- To determine the optimal block size for each kernel.
  This should be repeated for each kind of GPU,
  and for each size of the problem.
- To tune your CUDA code to attain the maximal performance for a GPU could be a long and tedious process. The limit is your understanding of the algorithm, the GPU hardware, and your imagination.

Matrix Addition (1)

$$C = A + B, \qquad c_{ji} = a_{ji} + b_{ji}$$

$$\begin{pmatrix} c_{11} & \cdots & c_{1N} \\ \vdots & c_{ji} & \vdots \\ c_{N1} & \cdots & c_{NN} \end{pmatrix} = \begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & a_{ji} & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix} + \begin{pmatrix} b_{11} & \cdots & b_{1N} \\ \vdots & b_{ji} & \vdots \\ b_{N1} & \cdots & b_{NN} \end{pmatrix}$$

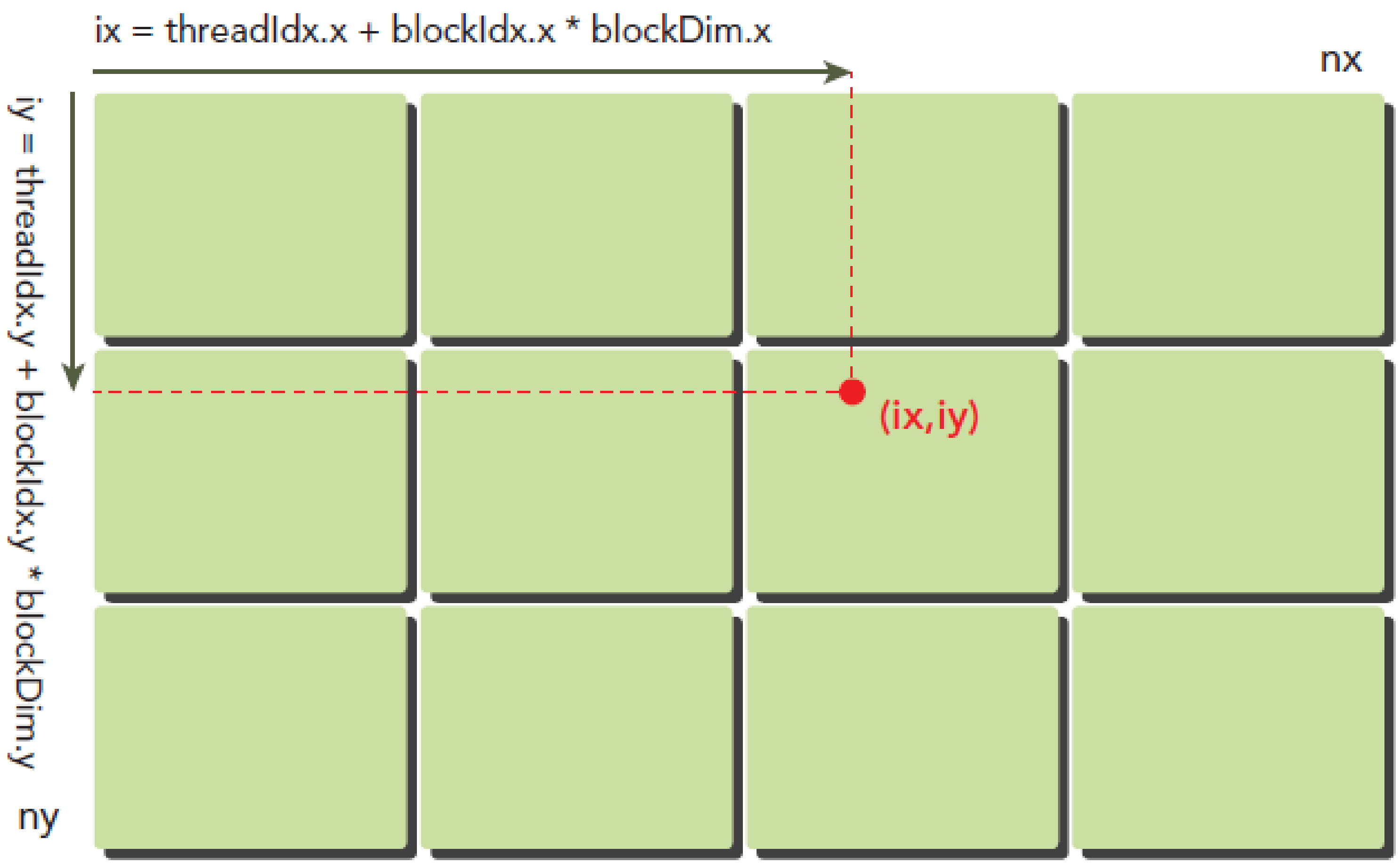
It is natural to use 2D grid and 2D blocks

matrix indices:

i = blockDim.x * blockIdx.x + threadIdx.x

j = blockDim.y * blockIdx.y + threadIdx.y

Matrix Addition (2)



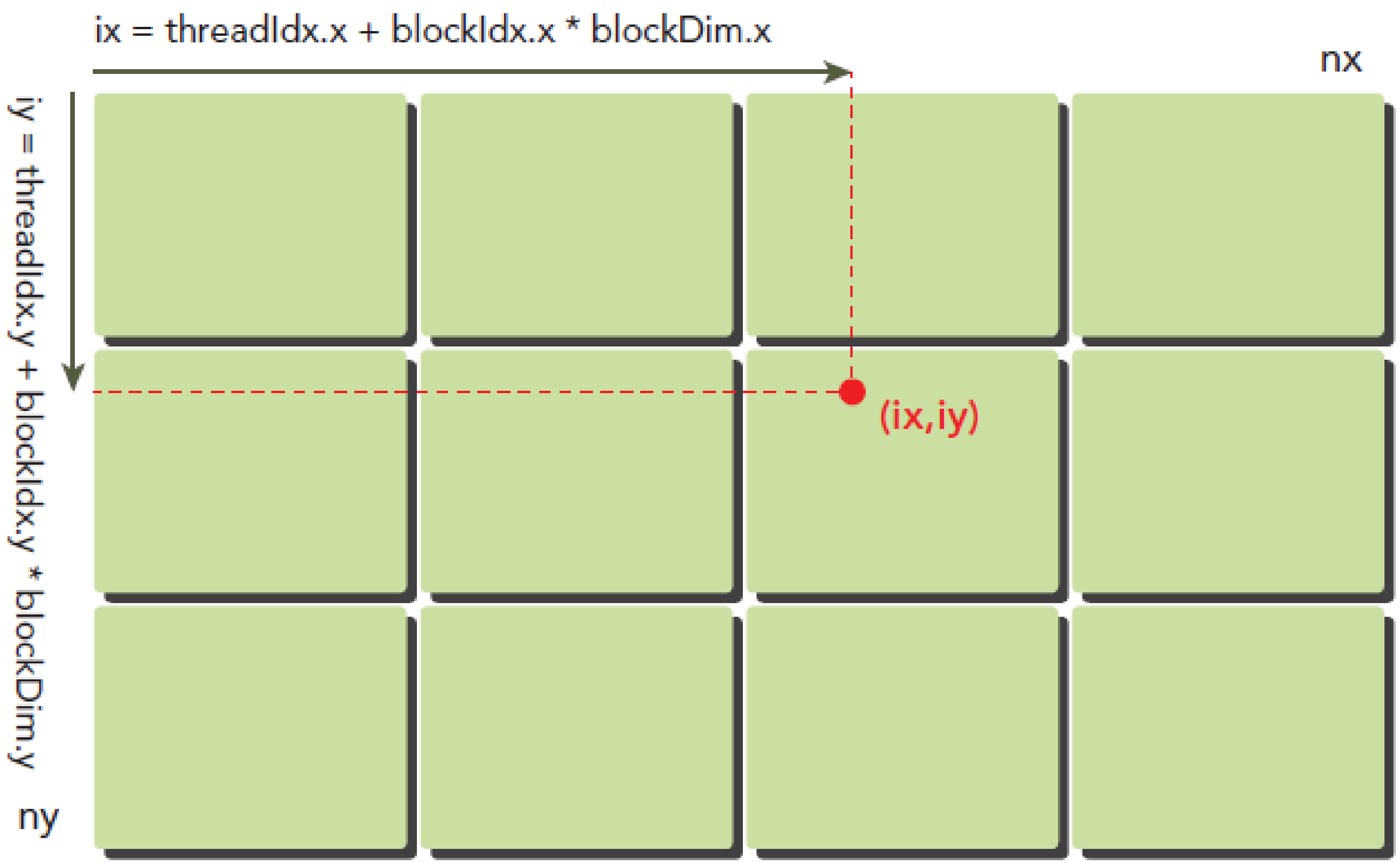
matrix coordinate: (ix,iy)
global linear memory index: idx = iy*nx + ix

Matrix Addition (3)

```
__global__ void matAdd(float *A, float *B, float *C, int N)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i < N && j < N)
      C[j][i] = A[j][i] + B[j][i];
}

int main()
{
   int  N=1024;
   dim3 dimBlock(16, 16);
   dim3 dimGrid((N + dimBlock.x – 1) / dimBlock.x,
                         (N + dimBlock.y – 1) / dimBlock.y);
    matAdd<<<dimGrid, dimBlock>>>(A, B, C, N);
}
```

Field Theory on the 2D Lattice (1)



matrix coordinate: (ix,iy)
global linear memory index: idx = iy*nx + ix

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 $\phi(i)$, a vector

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

Field Theory on the 2D Lattice (3)

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

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

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

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

$$\nabla^2\phi \to \phi(x+1,y) + \phi(x-1,y) + \phi(x,y+1) + \phi(x,y-1) - 4\phi(x,y)$$
$$\equiv D\phi$$

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