Practical Parallel Computing (実践的並列コンピューティング)

2025 Class No.10
[CUDA Part] (2)
Using Many Threads in CUDA

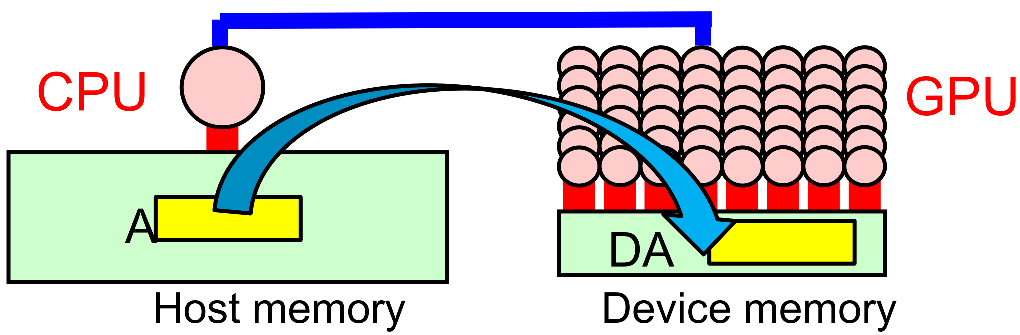
Toshio Endo
endo@scrc.iir.isct.ac.jp

Overview of This Course

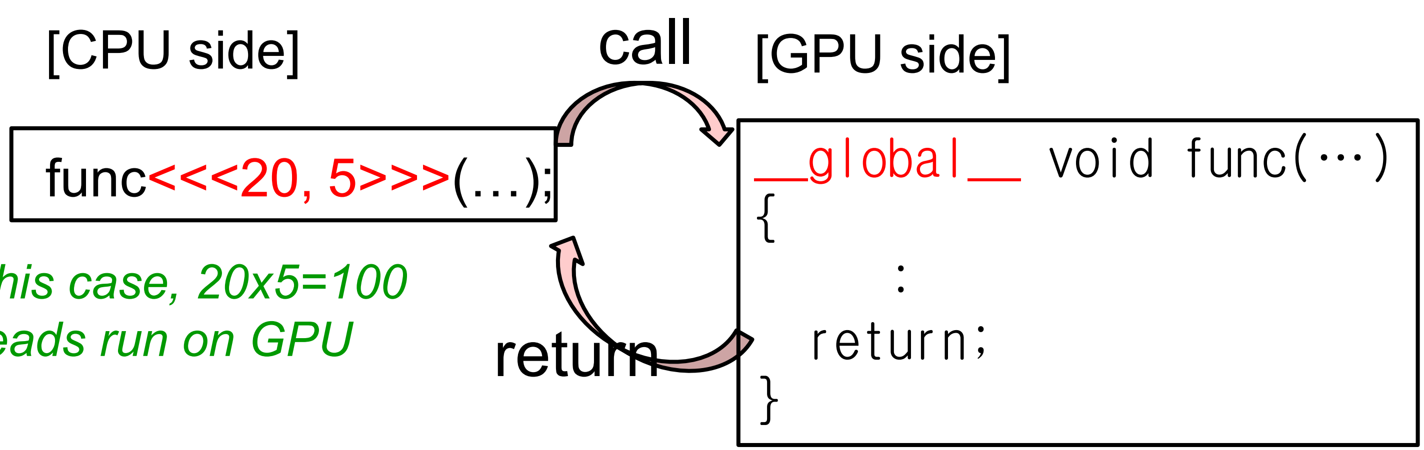
- Introduction Part
  - 2 classes
- OpenMP (OMP) Part
  - 4 classes
  - Report (required)
- OpenACC (ACC) Part
  - 2 classes
  - Report (required)
- CUDA Part
  - 3 classes ← We are here (2/3)
  - Report (elective)
- MPI Part
  - 3 classes
  - Report (elective)

Review: How We Write CUDA Programs

- Data is prepared on GPU device memory
  - cudaMalloc(), cudaMemcpy()



- CPU function calls GPU kernel functions

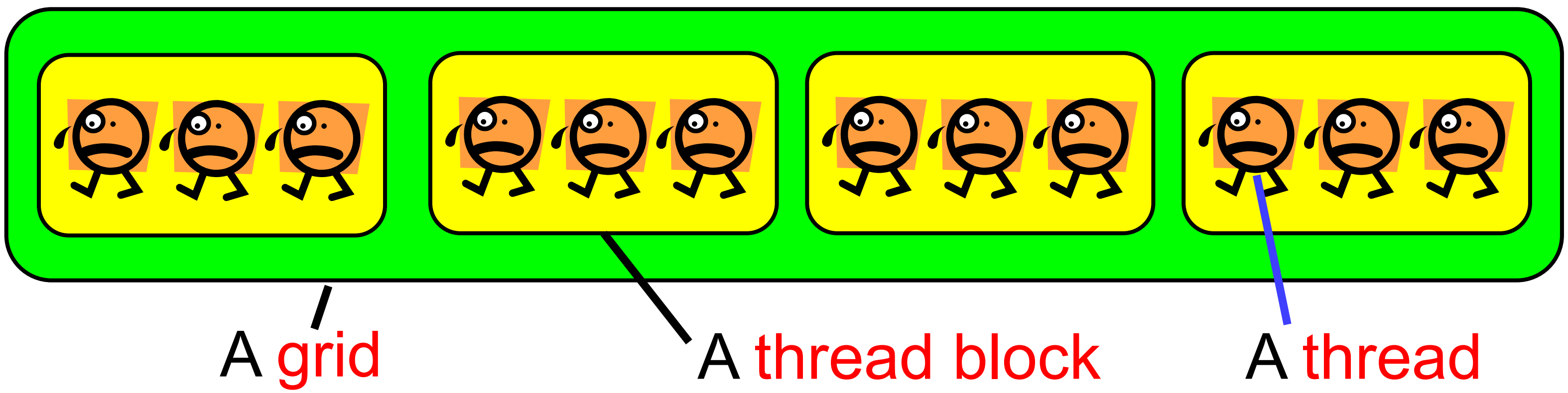


In this case, 20x5=100 threads run on GPU

Review: Grid, Thread Blocks, Threads

A grid has several thread blocks

A thread block has several threads



cf) func <<< 4, 3 >>> (); → 12 threads

Number of thread blocks = gridDim

Number of threads per block = blockDim

Review: How is Number of Threads Designed?

We have to deicide 2 numbers

<<<block number, block size>>>

A better way would be

(1)We decide total number of threads P

- P should be very large, such as >1,000,000

(2)We tune each block size BS

- Good candidate are 16, 32, 64, … 1024

(3)Then block number is P/BS

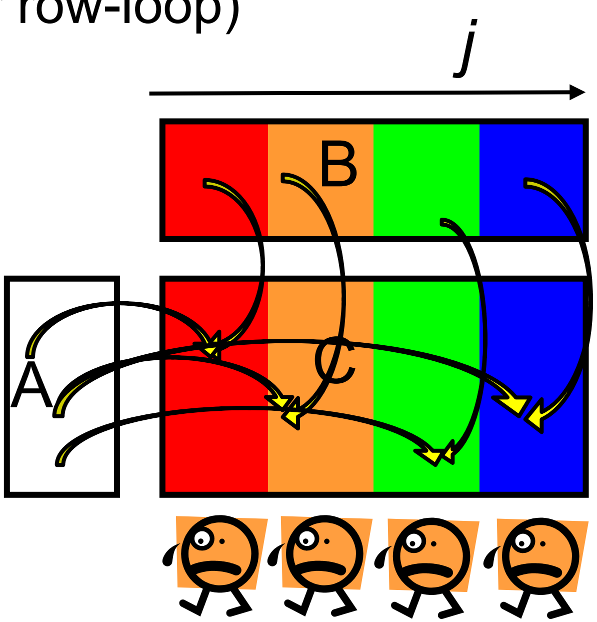
- It is better to consider indivisible cases

Review: mm-1dpar Sample is Slow
Related to [C3]

CUDA versions of mm samples are at
- ppcomp-ex/cuda/mm-1dpar/ (slow)
- ppcomp-ex/cuda/mm/ (faster)

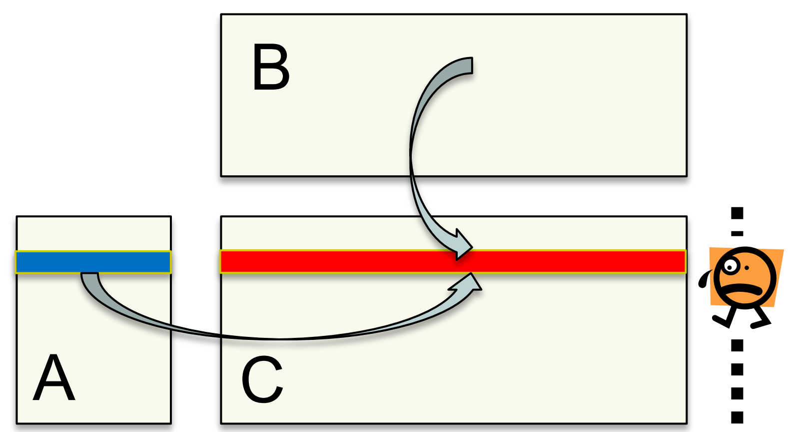
OpenMP

- Parallelize column-loop (or row-loop)



CUDA (mm-1dpar)

- We can create many threads
- 1 thread computes 1 row
  - We use m threads



m threads are too few for GPU

In Case of mm-acc (OpenACC)

- mm-use “acc loop independent” twice
  - j loop (size n) and i loop (size m)

```
#pragma acc data copyin(A[0:m*k],B[0:k*n]),copy(C[0:m*n])
#pragma acc kernels
#pragma acc loop independent   // j loop is parallelized
  for (j = 0; j < n; j++) {
#pragma acc loop seq
    for (l = 0; l < k; l++) {
#pragma acc loop independent    // i loop is parallelized
      for (i = 0; i < m; i++) {
        Ci,j += Ai,l * Bl,j;
      } } }
```

➔ This program creates m * n threads internally

➔ Also we want to create m * n threads on CUDA

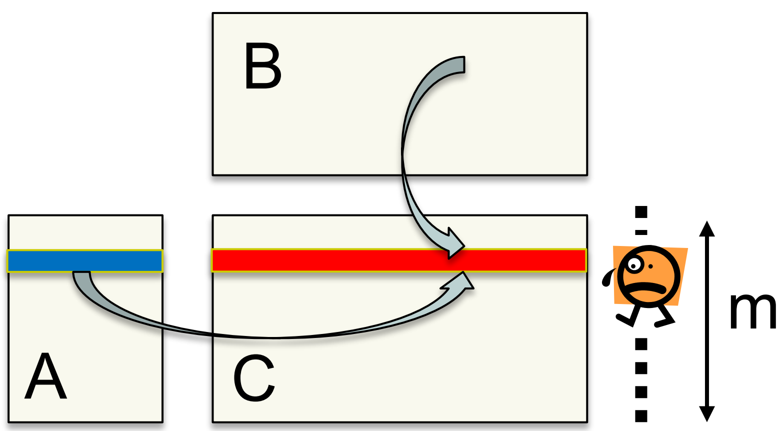
Improvement of mm: How to Use More Threads

In mm, computation of each C element is independent with each other

cuda/mm-1dpar

- 1 thread computes 1 row

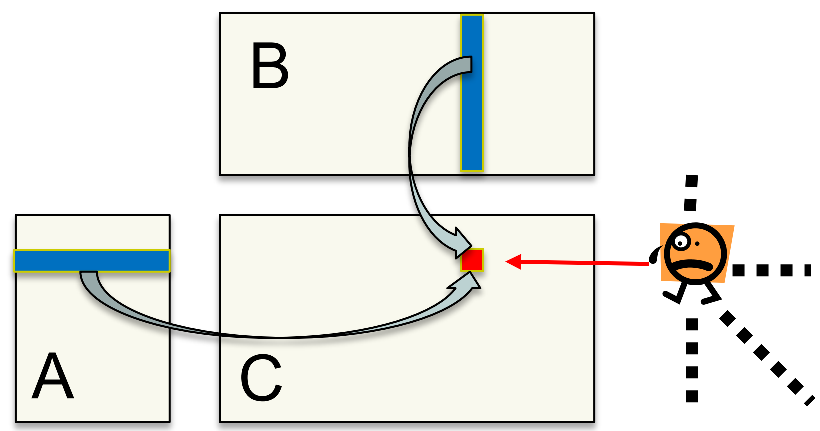
➔ We use m threads



cuda/mm

- 1 thread computes 1 element

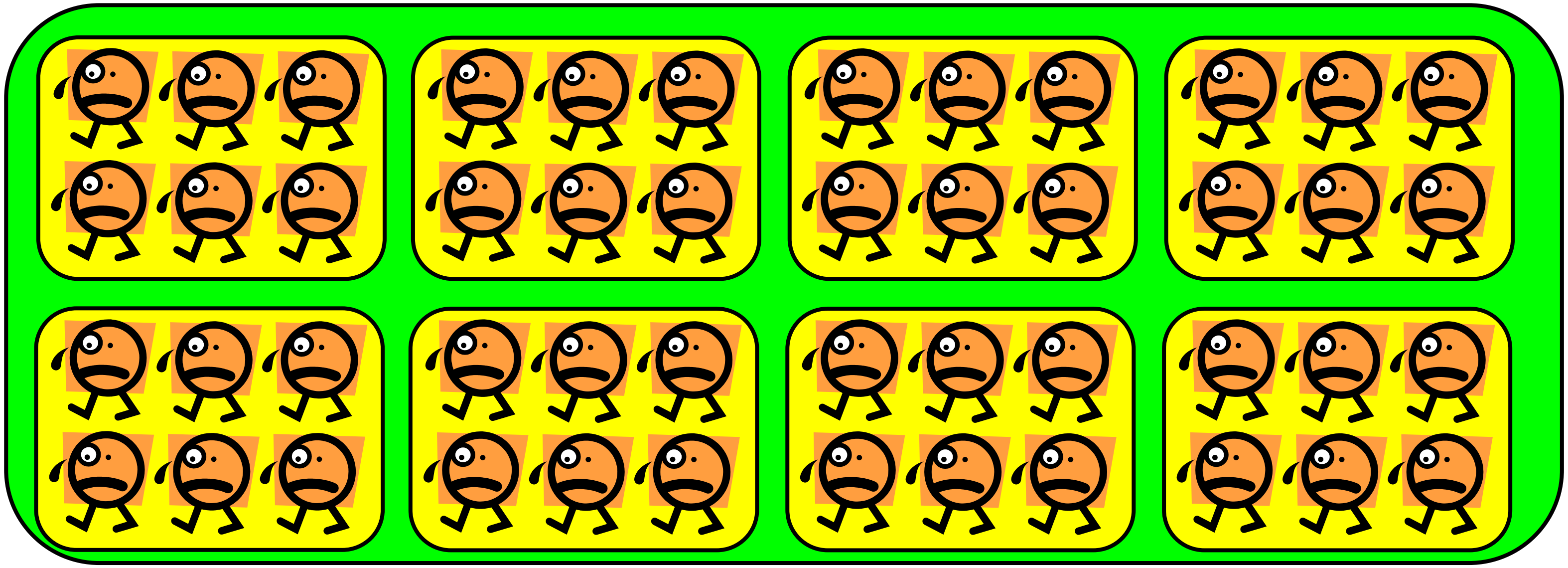
➔ We use m × n threads !!



Creating Threads with 2D/3D IDs

- Now we want to make m*n (may be >1,000,000) threads
  - <<<(m*n)/BS, BS>>> is ok, but coding is bothersome
- On CUDA, thread blocks and thread can have 3D IDs
  - gridDim and blockDim may have "dim3" type, 3D vector structure with x, y, z fields

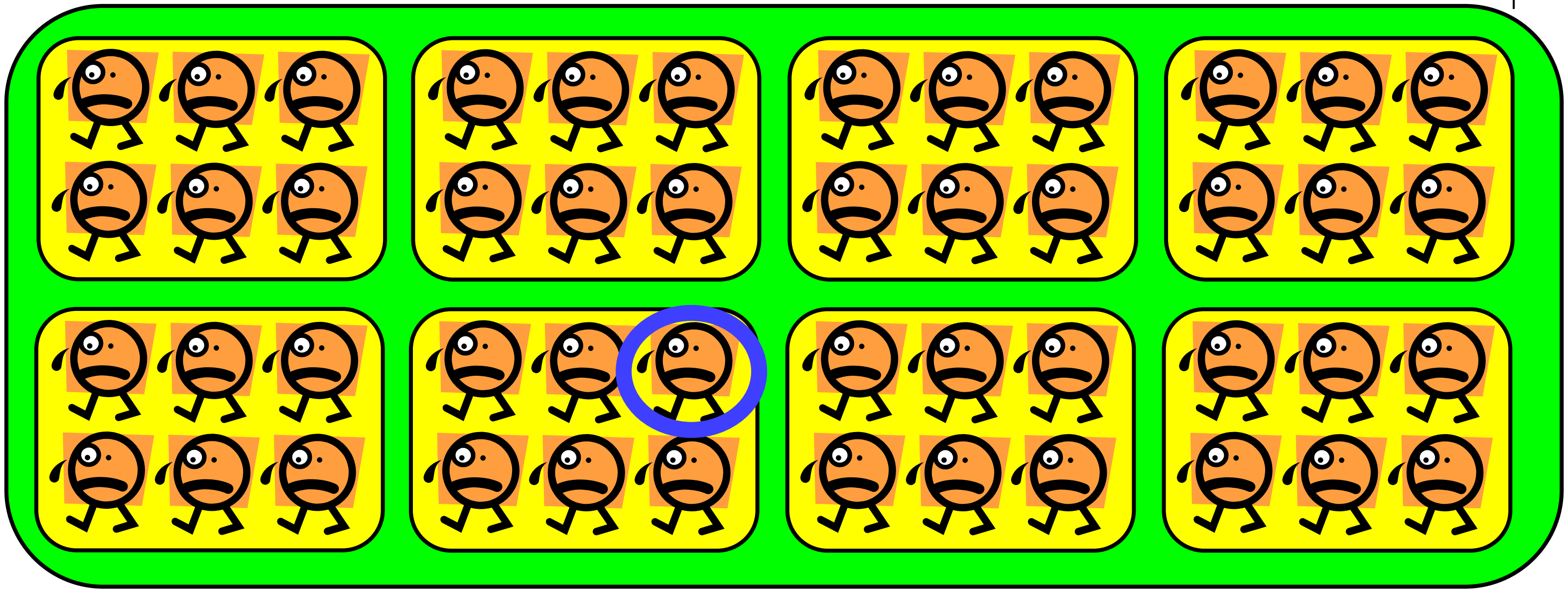
cf) func <<< dim3(4,2,1), dim3(3,2,1) >>> (); → 48 threads



※ This example is the case of 2D (Z dimensions are 1)

Thread IDs in multi-dimensional cases

In the case of func <<< dim3(4,2,1), dim3(3,2,1) >>> ();



- For every thread,

  gridDim.x=4, gridDim.y=2, gridDim.z=1

  blockDim.x=3, blockDim.y=2, blockDim.z=1
- For the thread with blue mark,

  blockIdx.x=1, blockIdx.y=1, blockIdx.z=0

  threadIdx.x=2, threadIdx.y=0, threadIdx.z=0

Notes on 1D Specification

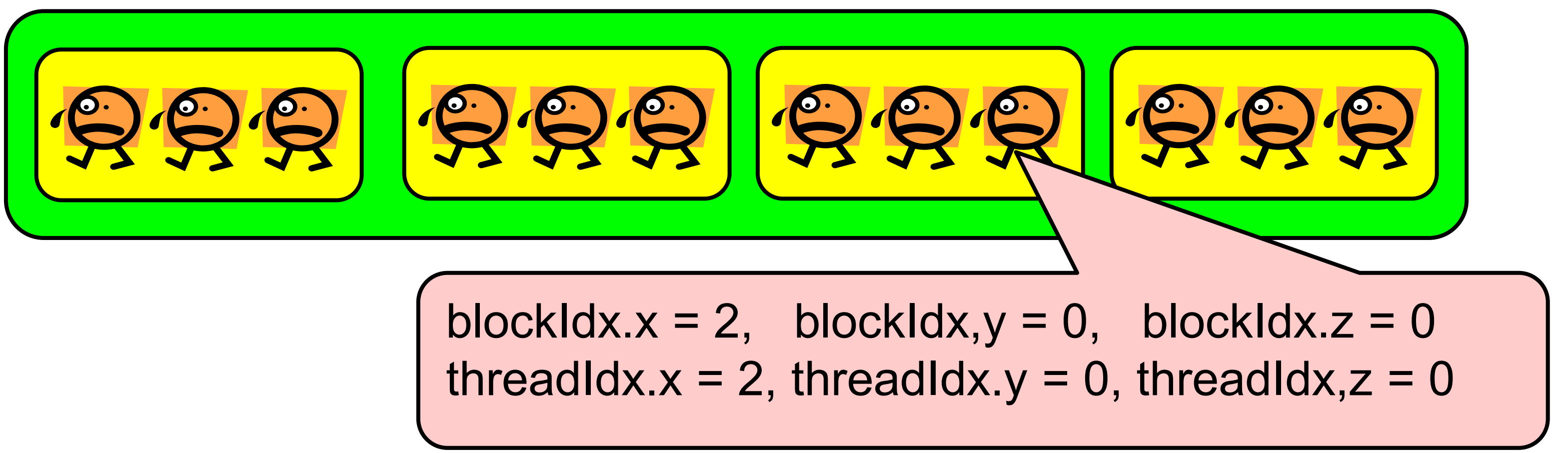
- So far we used 1D gridDim, 1D blockDim
- They are special cases of 3D IDs

The followings are same meanings

```
func <<<4, 3>>> ();
```

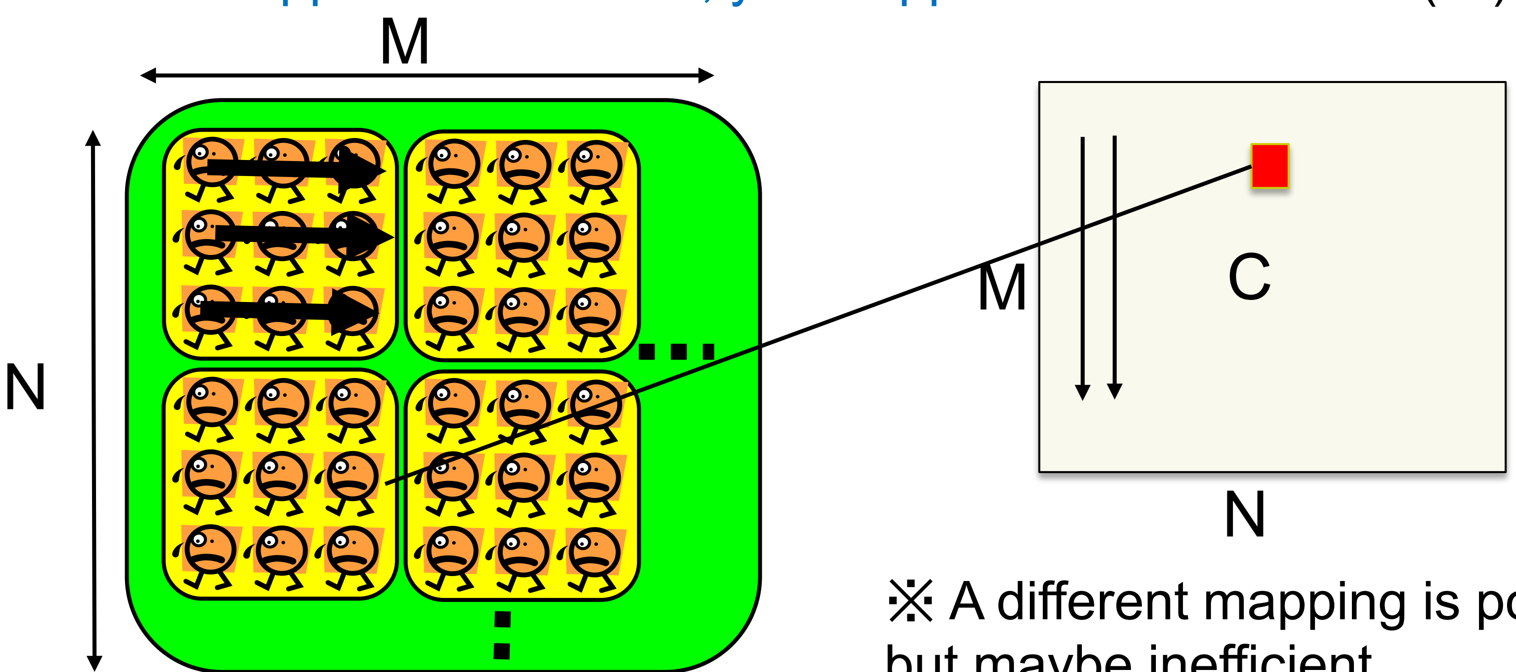
```
func <<<dim3(4,1,1), dim3(3,1,1)>>> ();
```

```
dim3 grid = dim3(4,1,1);
dim3 block = dim3(3,1,1);
func <<<grid, block>>> ();
```



Using More Threads in cuda/mm Sample

- The total number of threads should be m*n
- How do we determine gridDim, blockDim?
  - <<<m, n>>> does not work for CUDA constraints
- Here, we determine blockDim as x=16, y=16 → 256 threads per block
  - Then gridDim is computed from M, N
- x is mapped to row index, y is mapped to column index (※)



※ A different mapping is possible, but maybe inefficient

CPU Side Code in cuda/mm

```
matmul_kernel<<<dim3(m / BS, n / BS, 1), dim3(BS, BS, 1)>>>
        (DA, DB, DC, m, n, k);
```

gridDim: dim3(m / BS, n / BS, 1)

blockDim: dim3(BS, BS, 1)

BS=16 in this sample

Using rounding up for indivisible cases...

```
dim3 grid = dim3((m+BS-1)/BS, ((n+BS-1)/BS), 1);
dim3 block = dim3(BS, BS, 1);
matmul_kernel<<<grid, block>>>(DA, DB, DC, m, n, k);
```

GPU Side Code in cuda/mm

```
__global__ void matmul_kernel(double *DA, double *DB, double *DC, int m, int n, int k)
{
    int i, j, l;
    j = blockIdx.y * blockDim.y + threadIdx.y;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= m || j >= n) return; // do nothing

    double cij = DC[i+j*ldc];
    for (l = 0; l < k; l++) {
        cij += DA[i+l*m] * DB[l+j*n];
    }
    DC[i+j*ldc] = cij;
}
```

A thread gets
entire thread IDs (x,y)
j 🡐 y dimension, i 🡐 x dimension

Avoid too many threads

A thread computes C_{ij}
➔ Only `l` loop (dot prod)

Notes about Report [C3]

- Compare speeds of
  - cuda/mm-1dpar: m threads
  - cuda/mm: m*n threads
  - Please use various matrices sizes (m, n, k)
- Evaluate effect of data transfer cost
  - The sample prints measured time in detail
  - Discussions are similar to OpenACC version (ppcomp25-8 slides)
    - Computation complexity: O(mnk)
    - Data transfer complexity: O(mk+kn+mn)

CUDA Rules on Number of Threads

func<<<A, B>>> (...); *(A, B are integers)* is same as
func<<<dim3(A,1,1), dim3(B,1,1)>>> (...);

gridDim blockDim

func<<<dim3(gx, gy, gz), dim3(bx, by, bz)>>> (...);

≦ 2^{31}-1

≦ 65535

≦ 1024

≦64

Also, bx*by*bz must be ≦1024

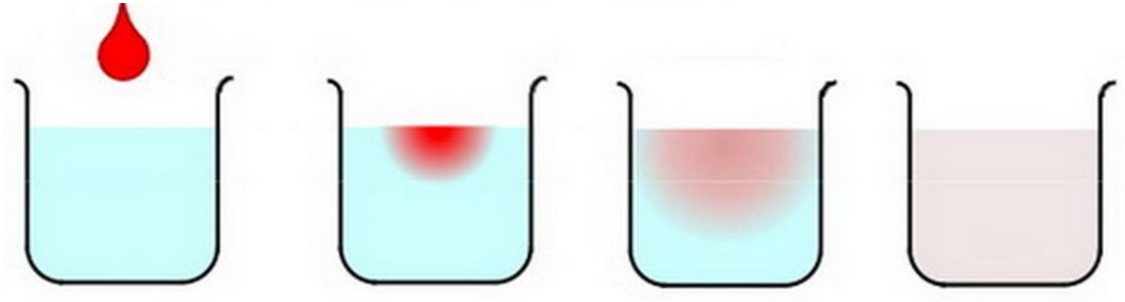
Size of BlockDim has severe limitation ☹

Cf) <<<m, n>>> causes an error if n>1024 ☹

“diffusion” Sample Program

Target of [C1] , details are in ppcomp25-4

An example of diffusion phenomena:

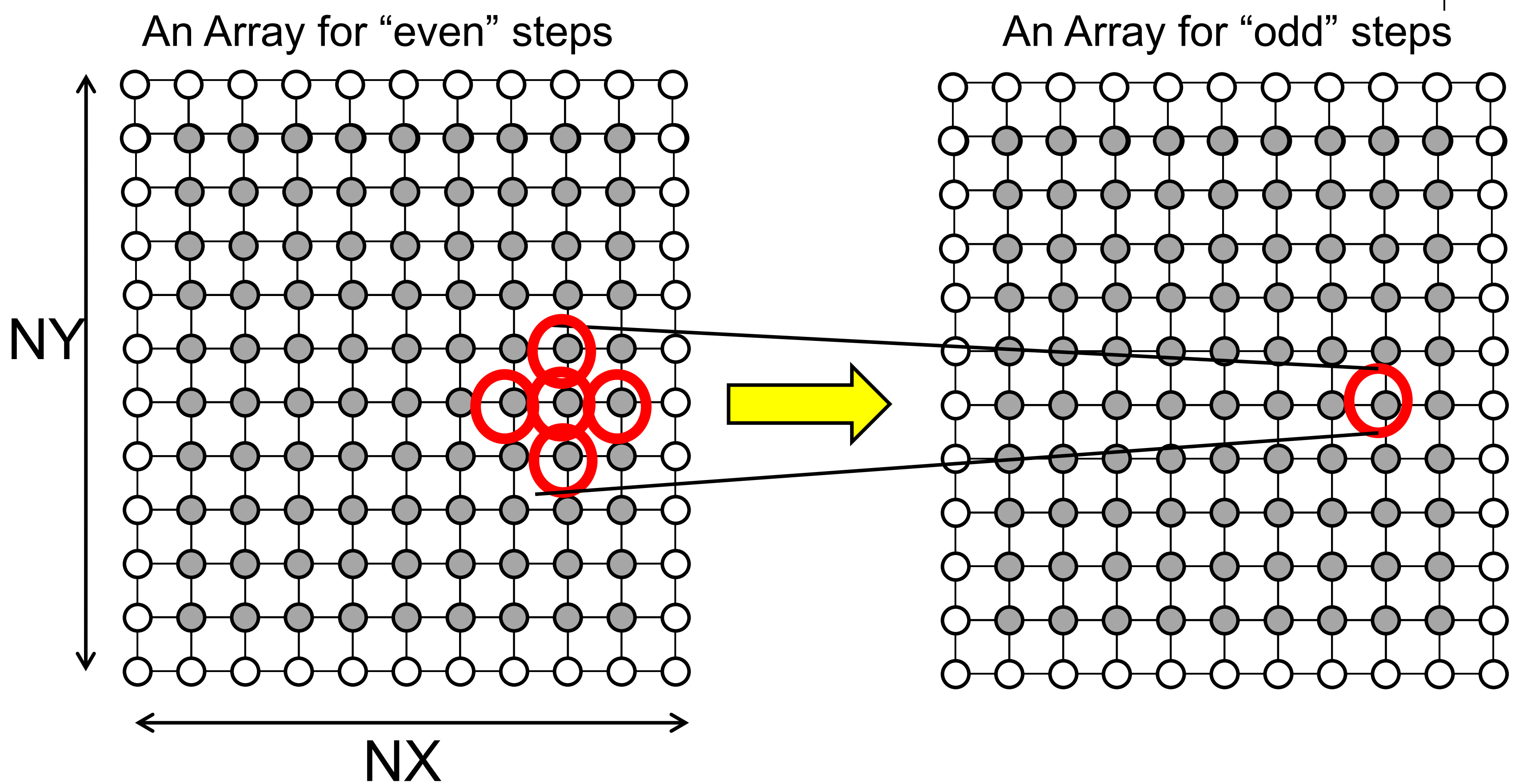


The ink spreads gradually, and finally the density becomes uniform (Figure by Prof. T. Aoki)

Base version: ppcomp-ex/base/diffusion/
You can use ppcomp-ex/cuda/diffusion/

```
cd ppcomp-ex/cuda/diffusion
module load nvhpc
make
./diffusion 20      // number of time steps
```

Discussion on diffusion sample



Both arrays have to be on GPU device memory when computations are done

Consideration of Parallelizing Diffusion

- x, y loops can be parallelized
- t loop cannot be parallelized

```
[Data transfer from CPU to GPU]
for (t = 0; t < nt; t++) {

  for (y = 1; y < NY-1; y++) {
    for (x = 1; x < NX-1; x++) {
          :
    }
  }
}
[Data transfer from GPU to CPU]
```

GPU computation must be a distinct function (GPU kernel function)

It's better to transfer data *out of* t-loop

Preparing GPU Kernel Function

- Unlike OpenACC/OpenMP, region on GPU must be a separated function. Code needs large change!

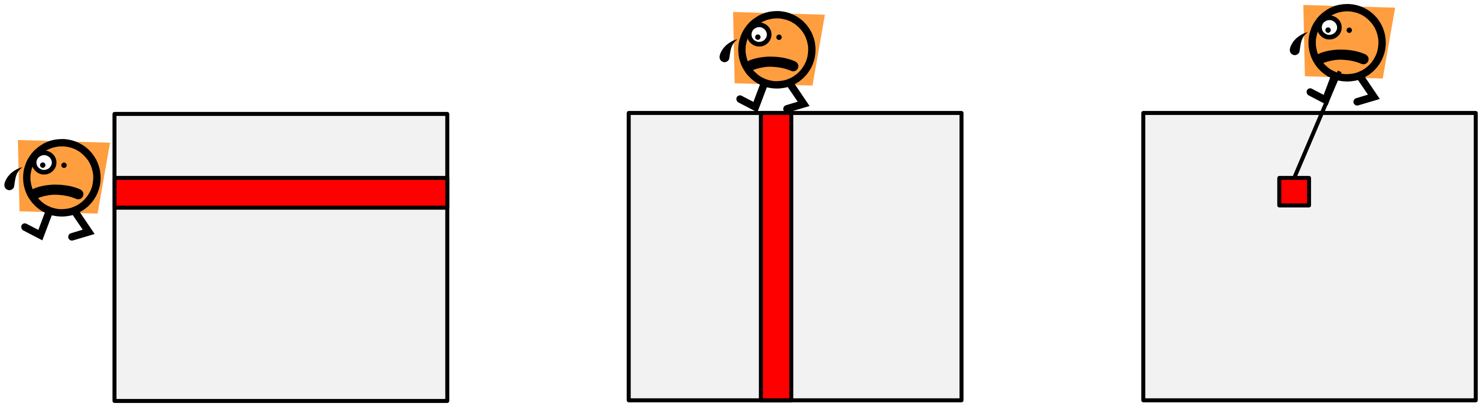
```
__global__ void calc_gpu(...) {
    :   // behavior of each thread
}

int calc(int nt) {
    :
  for (t = 0; t < nt; t++) {
    :
    calc_gpu<<<....>>>(...)
    :
  }
}
```

Calling a kernel function

Considering CUDA Threads

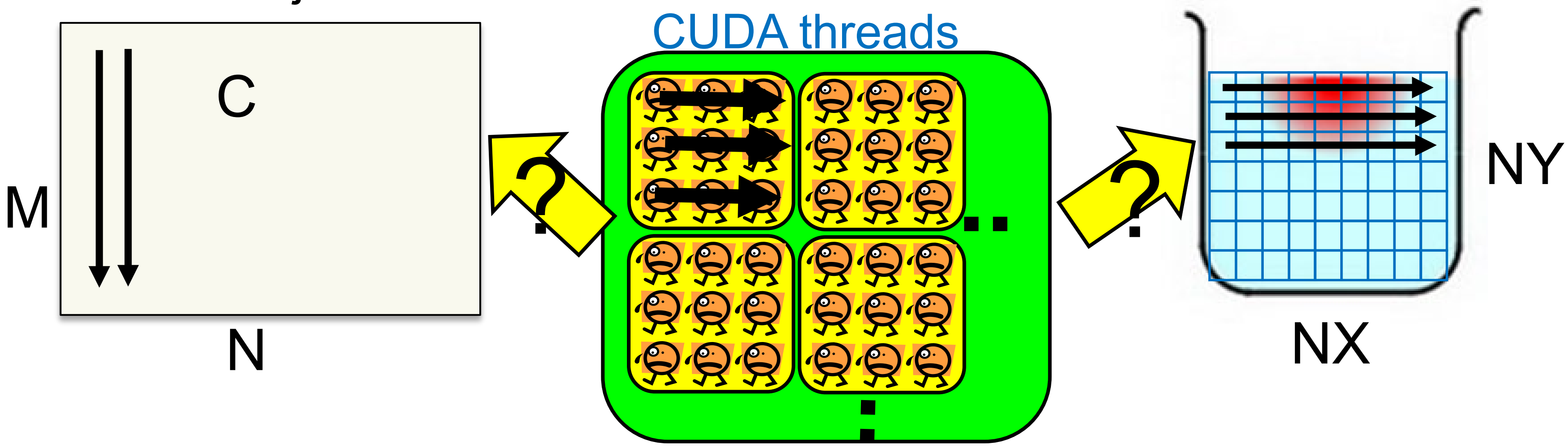
- How do we design threads on CUDA?
- There are several choices
  - 1thread = 1row
    - We use NY threads in total → only x-loop in kernel function
  - 1thread = 1column
    - We use NX threads in total → only y-loop in kernel function
  - 1thread = 1element (Recommended)
    - We use NX x NY threads in total → No loop in kernel function!
    - This will be fast since the number of threads is very large



Mapping between Threads and Data

mm:
Matrices has
column-major format

diffusion:
2D array has
row-major format



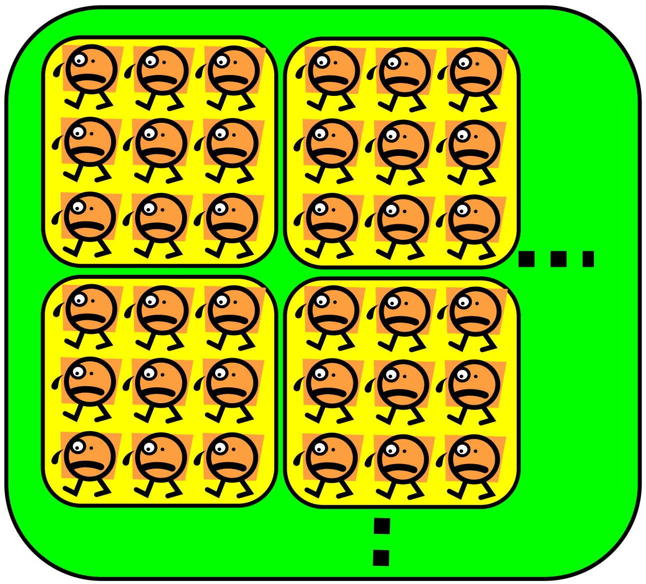

```
j = blockIdx.y * blockDim.y +
threadIdx.y;
i = blockIdx.x * blockDim.x +
threadIdx.x;
 : This thread computes Cij
```

```
y = blockIdx.y * blockDim.y +
threadIdx.y;
x = blockIdx.x * blockDim.x +
threadIdx.x;
 : This thread computes[y][x]
```

[Q] What if the dimensions are exchanged?

Considering gridDim/blockDim (1)

$$\text{func} <<< \underbrace{\text{dim3 (?, ?, ?)}}_{\text{gridDim}}, \underbrace{\text{dim3 (?, ?, ?)}}_{\text{blockDim}} >>>(\ldots);$$



(1) We decide total number of threads

➔ (NX, NY, 1) threads

- See notes on the next page

(2) We tune each block size (blockDim)

➔ Good candidates are (4, 4, 1), (8, 8, 1), (16, 16, 1), (32, 32, 1)

- The number must be ≦ 1024
- How about non-square blocks?

(3) Then block number (gridDim) is determined

We should consider indivisible cases

Considering gridDim/blockDim (2)

- In diffusion, Points [1, NX-1) × [1, NY-1), <u>excluded boundary</u>, should be computed

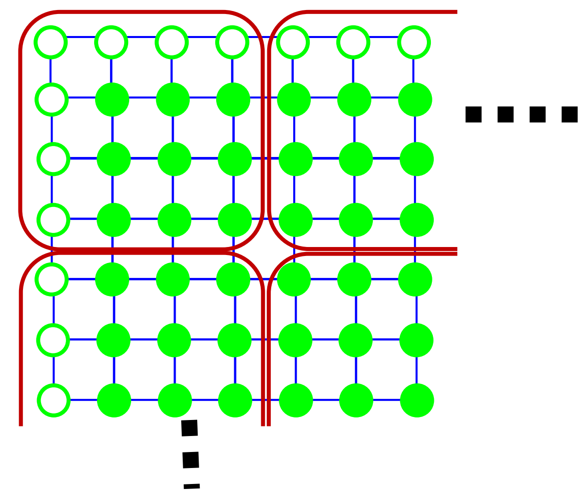
  There are choices:

  (A) Create NX x NY threads
  - Thread (x,y) computes (x,y)
  - Threads with below IDs do nothing
    - x == 0 or y == 0 or x ≧ NX-1 or y ≧ NY-1

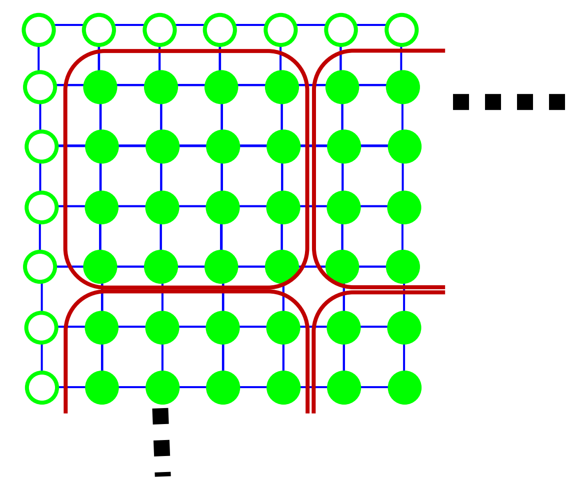
  (B) Create (NX-2) x (NY-2) threads
  - Thread (x,y) computes (x+1,y+1)
  - Threads with below IDs do nothing
    - x ≧ NX-2 or y ≧ NY-2

Either is ok ☺

(A)



(B)



Discussion on Data Transfer of Diffusion

Both codes will work, but how about speeds?

```
[Data transfer from CPU to GPU]
for (t = 0; t < nt; t++) {
                :

  for (y = 1; y < NY-1; y++) {
    for (x = 1; x < NX-1; x++) {
                :
    }
  }

}
[Data transfer from GPU to CPU]
```

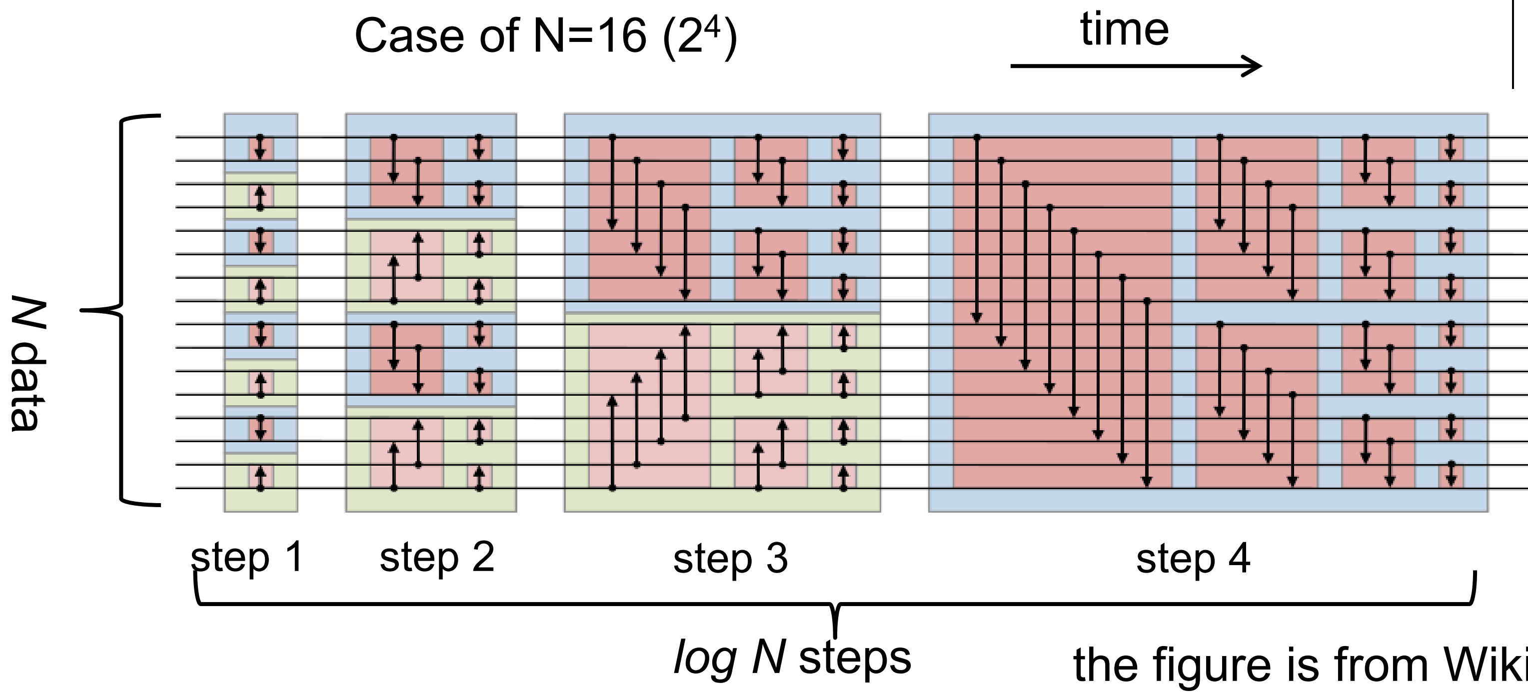
Computation: O(NX NY nt)
Transfer: O(NX NY)

```
for (t = 0; t < nt; t++) {
                :
  [Data transfer from CPU to GPU]
  for (y = 1; y < NY-1; y++) {
    for (x = 1; x < NX-1; x++) {
                :
    }
  }
  [Data transfer from GPU to CPU]
}
```

Computation: O(NX NY nt)
Transfer: O(<u>NX NY nt</u>)

“bsort” Sample Program

Target of [C2], details are in ppcomp25-4



the figure is from Wikipedia

Base version: ppcomp-ex/base/bsort/
You can use ppcomp-ex/cuda/bsort/

```
cd ppcomp-ex/cuda/bsort
module load nvhpc  // if not yet
make
./bsort 1000000 // number of elements to be sorted
```

How Can We Parallelize bsort?

- k loops can be parallelized
- i, j loops cannot be parallelized

```
for (i = 1; (1<<i) <= N2; i++) { // step loop
  for (j = i-1; j >= 0; j--) { // sub-step loop
    :
    for (k = 0; k < N2; k++) {
        // compare 2 data and swap
    }
  }
}
```

GPU computation must be a distinct function (GPU kernel function)

Next, where should the cudaMemcpy be?

- Please consider reducing data copy cost

Preparing GPU Kernel Function for bsort

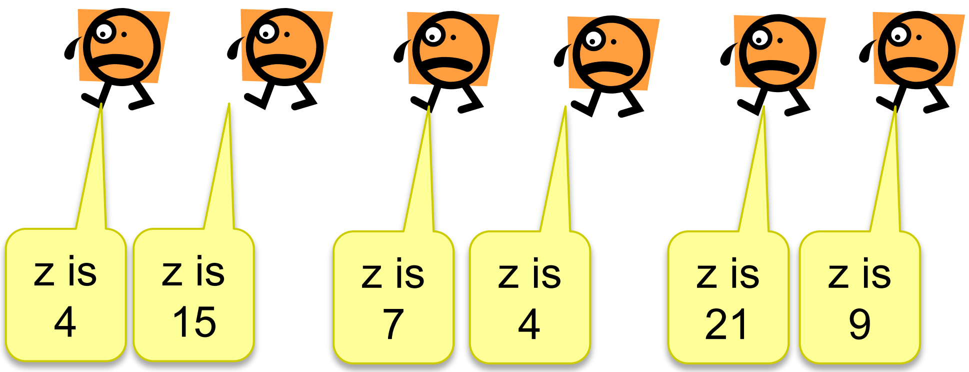
```
__global__ void sort_kernel (...) {
   int k = blockIdx.x * blockDim.x
       + threadIdx.x;
     :
   // behavior of each thread
}

int sort(...) {
     :
   for (i = 1 ...) {
     for (j = i-1 ...) {
       :
       sort_kernel<<<....>>>(...)
       :
     } ...
```

Calling a kernel function
The total number of threads should be N2

Rules for Memory/Variables

- Variables declared in GPU kernel functions are "thread private"



- Device memory is shared by all CUDA threads
  - Be careful to avoid race condition problem (multiple threads write same address)
  - Reading same address is ok
- Do not forget host memory and device memory are separated

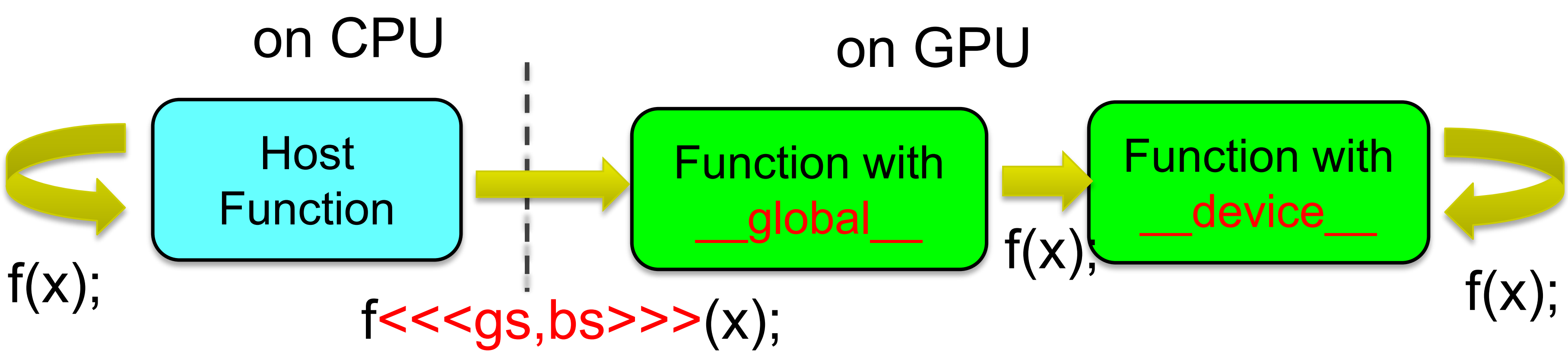
Two Types of GPU Kernel Functions

1) Functions with __global__ keyword

- “Gateway” from CPU
- Return value type must be “void”

2) Function with __device__ keyword → In OpenACC, #pragma acc routine

- Callable only from GPU
- Can have return values
- Recursive call is OK



What Can be Done in GPU Functions?

- Basic computations (+, -, *, /, %, &&, ||...) are OK
- if, for, while, return are OK
- Device memory access is OK
- Host memory access is NG
- Calling host functions is NG
- Calling most of functions in libc or other libraries for CPUs are NG
  - Several mathematical functions, sin(), sqrt()… are OK
  - printf() is OK
  - Calling malloc()/free() on GPU is OK, if the size must be small
    - Usually, use cudaMalloc() on <u>CPU</u>

Assignments in CUDA Part

Choose one of [C1]—[C4], and submit a report

Due date: May 26 (Monday)

[C1] Parallelize "diffusion" sample program by CUDA

[C2] Parallelize "bsort" sample program by CUDA

[C3] Evaluate speed of "acc/mm" sample in detail

[C4] (Freestyle) Parallelize *any* program by CUDA

For more details, please see ppcomp25-9 slides

Plan of CUDA Part

- Class #9
  - Introduction to CUDA, kernel functions
- Class #10 (Today)
  - Characteristics of grid, thread blocks, threads
- Class #11
  - Performance improvement on GPU

NOTICE:

- Due date for OpenACC report is changed from May 12 to May 15
  - Machine trouble on TSUBAME interactive nodes happens