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

Part 2: GPU

No 3: Threads in CUDA

May 13, 2024

Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp

Overview of This Course

- Part 0: Introduction
  - 2 classes
- Part 1: OpenMP for shared memory programming
  - 4 classes
- Part 2: GPU programming
  - 4 classes ← We are here (3/4)
  - OpenACC (1.5 classes) and CUDA (2.5 classes)
- Part 3: MPI for distributed memory programming
  - 4 classes

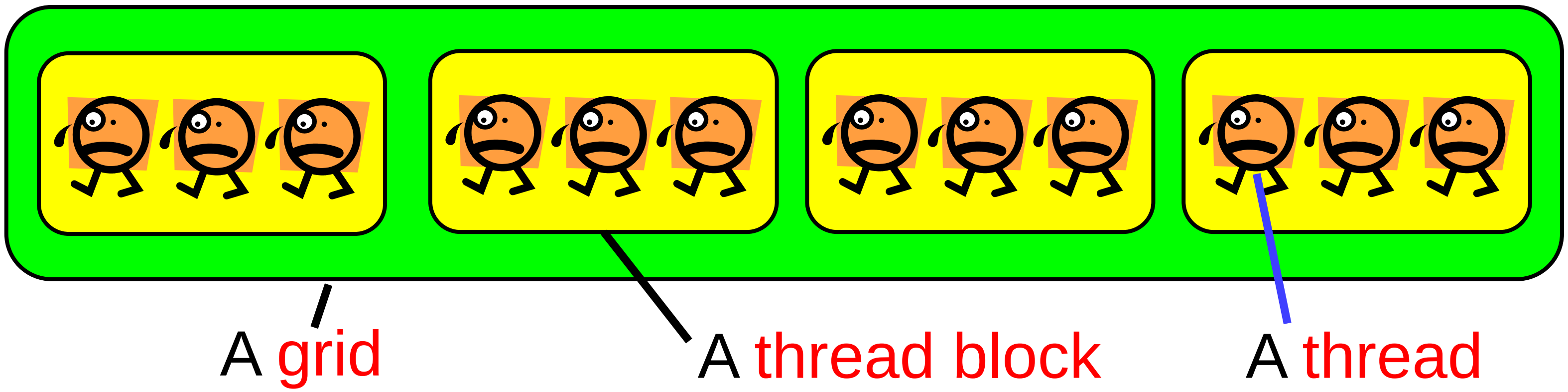
Comparing OpenMP/OpenACC/CUDA

	OpenMP	OpenACC	CUDA
Processors	CPU	CPU+GPU	
File extension	.c, .cc		.cu
To start parallel (GPU) region	#pragma omp parallel	#pragma acc kernels	func<<<…, …>>>()
To specify # of threads	export OMP_NUM _THREADS=…	(num_gangs, vector_length etc)	
Desirable # of threads	# of CPU cores or less	# of GPU cores or “more”	
To get thread ID	omp_thread_num()	-	blockIdx, threadIdx
Parallel for loop	#pragma omp for	#pragma acc loop	-
Task parallel	#pragma omp task	-	-
To allocate device memory	-	#pragma acc data	cudaMalloc()
To copy to/from device memory	-	#pragma acc data #pragma acc update	cudaMemcpy()
Functions on GPU	-	#pragma acc routine	__global__, __device__

※ “# of XXX” = “The number of XXX”

Threads in CUDA

Specify 2 numbers (at least) for number of threads, when calling a GPU kernel function



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

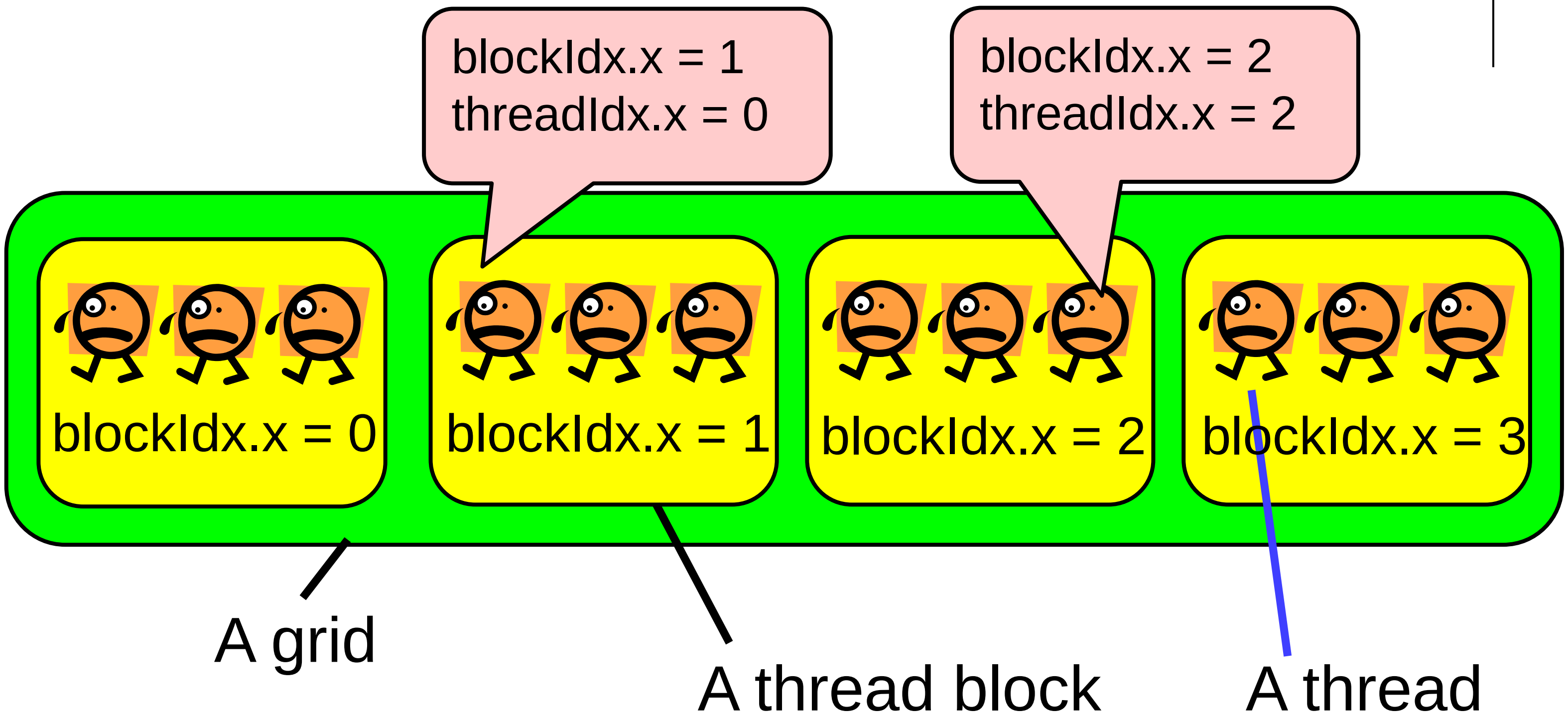
Number of thread blocks = gridDim

Number of threads per block = blockDim

The reason is related to GPU hardware

Thread block ⇔ SMX, Thread ⇔ CUDA core

Thread Block ID, Thread ID



For every thread, gridDim.x = 4, blockDim.x = 3

Note: In order to see the entire sequential ID, we should compute
blockIdx.x * blockDim.x + threadIdx.x

The Case of add-cuda Sample

- /gs/bs/tga-ppcomp/24/add-cuda/
- We want to do `for (i = 0; i < 100; i++) { DA[i] += DB[i]; }`

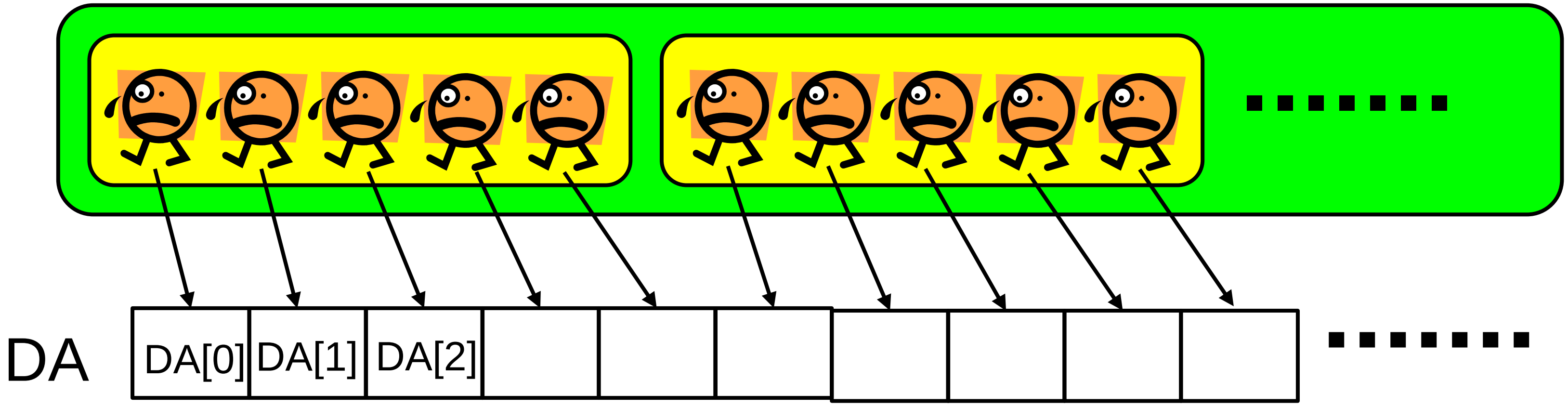
[CPU side]

```
add<<<20, 5>>>(…);
```

20x5=100 threads
will execute add function

[GPU side]

```
__global__ void add(int *DA, int *DB)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    DA[i] += DB[i];
    return;
}
```



How is Number of Threads Designed? (1)

On CUDA, a different strategy is required from on OpenMP

- On OpenMP, number of threads (OMP_NUM_THREADS) should be ≦ CPU cores (or hyper threads)
  - The number is basically determined by hardware
  - ≦48 on an interactive (node_o) node, ≦384 on node_f

- On CUDA, it is better to use number of thread ≧ GPU cores
  - ≧ 7,680 on an interactive (node_o) node with ½ GPU
  - ≧ 16,896 on gpu_1, node_q ...
  - You can use >1,000,000 threads!

How is Number of Threads Designed? (2)

We have to deicide 2 numbers

<<<block number, block size>>>

A better way would be

(1)We decide total number of threads P

(2)We tune each block size BS

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

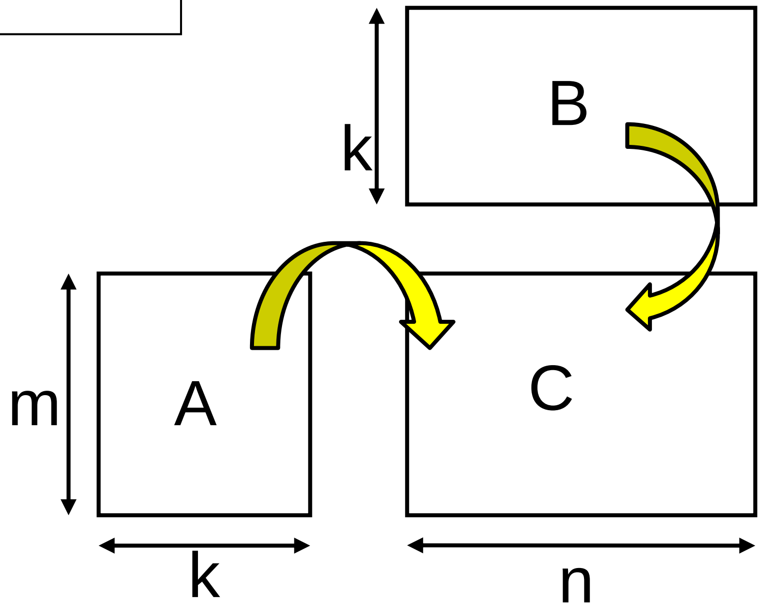
(3)Then block number is P/BS

- We consider indivisible cases later

“mm” sample: Matrix Multiply (related to [G2])

CUDA versions are at
- /gs/bs/tga-ppcomp/24/mm-v1-cuda/
- /gs/bs/tga-ppcomp/24/mm-cuda/

A: a (m × k) matrix, B: a (k × n) matrix
C: a (m × n) matrix
$C \leftarrow A \times B$



- Supports variable matrix size
- Execution:./mm [m] [n] [k]

On CUDA, We need to design
(1) How we parallelize computation
(2) How we put data on host memory & device memory

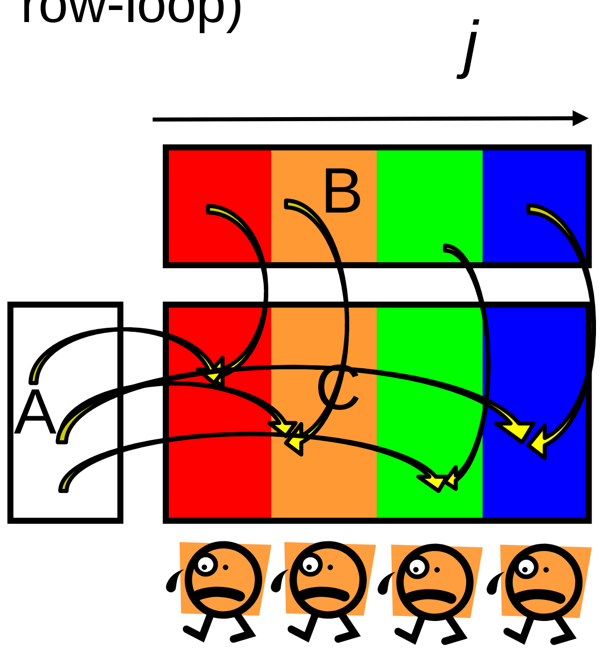
How We Parallelize Computation

In mm, we can compute different C elements in parallel

- On the other hand, it is harder to parallelize dot-product loop

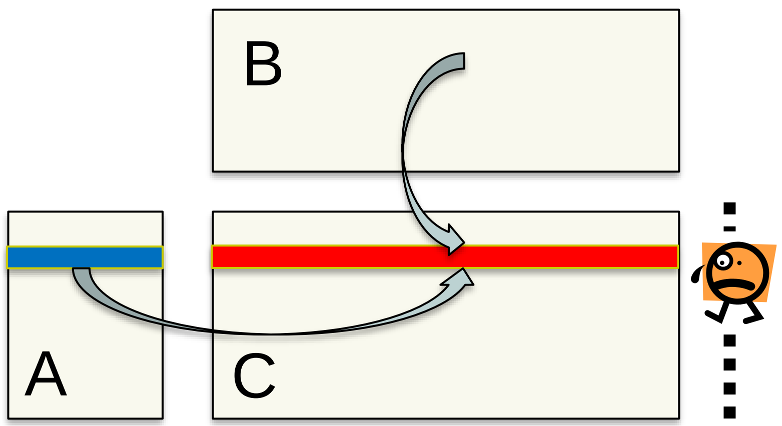
OpenMP

- Parallelize column-loop (or row-loop)



CUDA (mm-v1-cuda)

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



※ This is not the unique way

Parallelism in mm-v1-cuda

- It is ok to make >1000, >10000 threads on CUDA
- We use *m* threads for *m* rows computation

```
matmul_kernel<<<m/BS, BS>>>(.....);
```

gridDim

blockDim (BS=16 in this sample)

1 element for 1 row → No need of “i” loop in this sample

Note: <<<m, 1>>> also works, but speed is not good
<<<1, m>>> causes an error if m>1024 (CUDA’s rule)

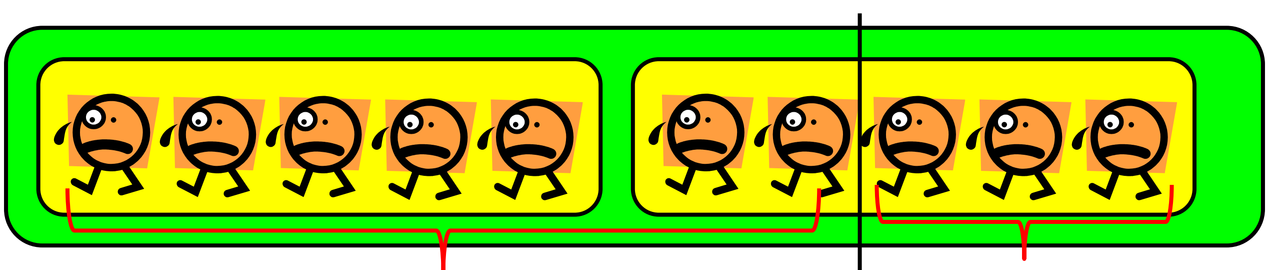
If Number of Threads is Indivisible by BlockDim

- m: the number of threads
- BS: BlockDim
- If m may be indivisible by BS, we should use

  <<<(m+BS-1)/BS, BS>>>

→ But # of threads may be larger m. “Extra” threads (id≧m) <u>should not work</u>.   See mm-v1-cuda/mm.cu

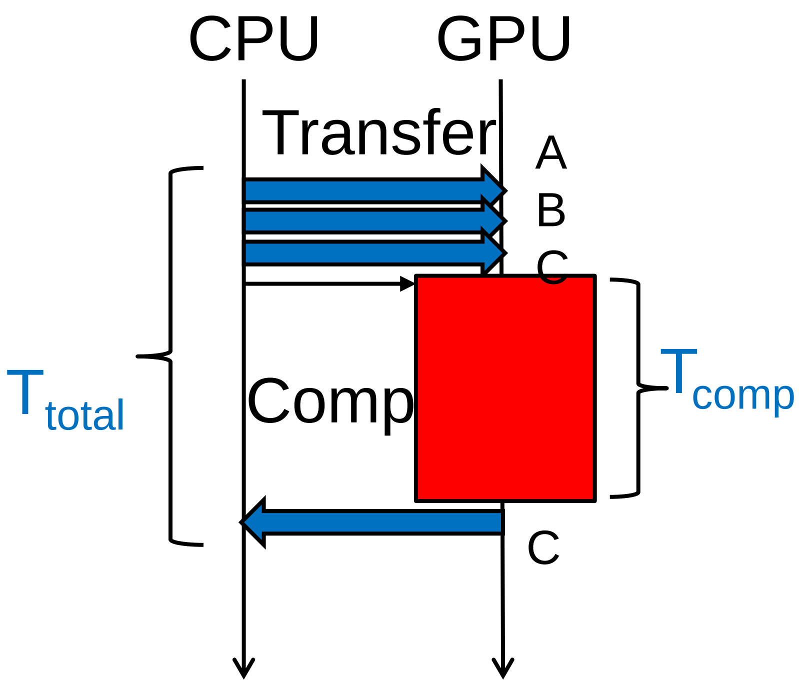
Example: m=7, BS=5 → <<<2,5>>>
10 threads start working, but 3 threads should do nothing



7 threads work

3 threads should return immediately

Data Transfer in mm-v1-cuda



(1) A, B, C are copied from CPU to GPU

- cudaMemcpy(DA, A, …) …

(2) Computation is done on GPU

(3) C is copied from GPU to CPU

- cudaMemcpy(C, DC, …)

Notes in Time Measurement

- clock(), gettimeofday() must be called from CPU
- For accurate measurement, we should call cudaDeviceSynchronize() before measurement
  - Actually GPU kernel function call and cudaMemcpy(HostToDevice) are non-blocking

Discussion on Speed
(related to [G2])

Bad news: mm-v1-cuda is much slower than mm-acc! ☹
(even slower than mm-omp?)

- In mm-acc, i-loop and j-loop has “loop independent”
  → $m \times n$ elements are computed in parallel

- In mm-v1-cuda, we use m threads in total
  → We should use <u>more threads</u> on a GPU!
  - At least, ≧ 7,680 = number of CUDA cores
  - We see m=1000~8000 threads are still insufficient, and slow

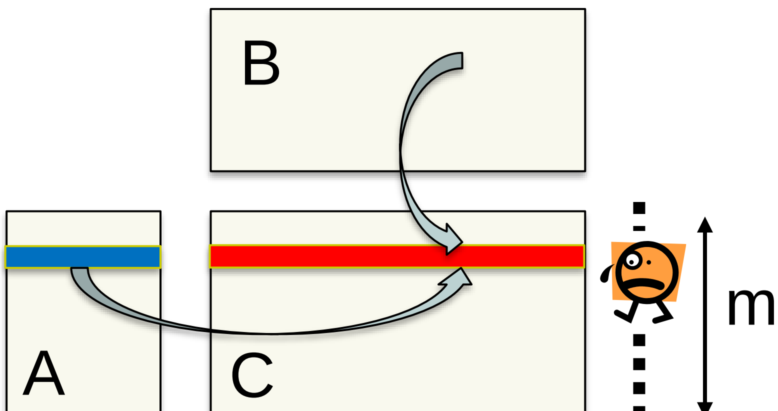
Improvement:
How to Use More Threads

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

mm-v1-cuda

- 1 thread computes 1 row

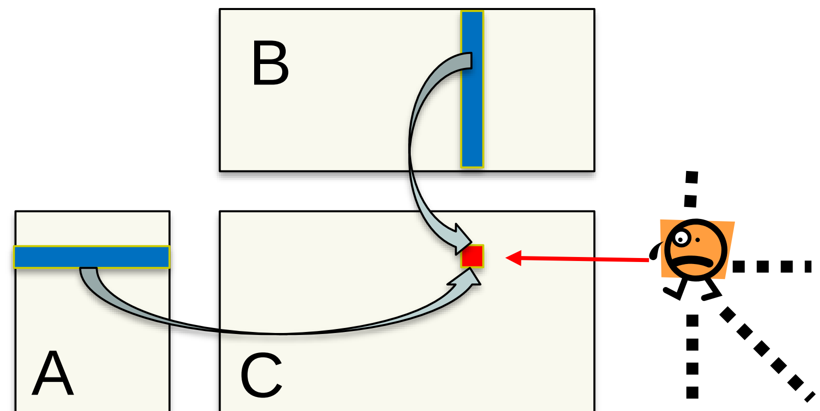
  ➔ We use m threads



mm-cuda

- 1 thread computes 1 element

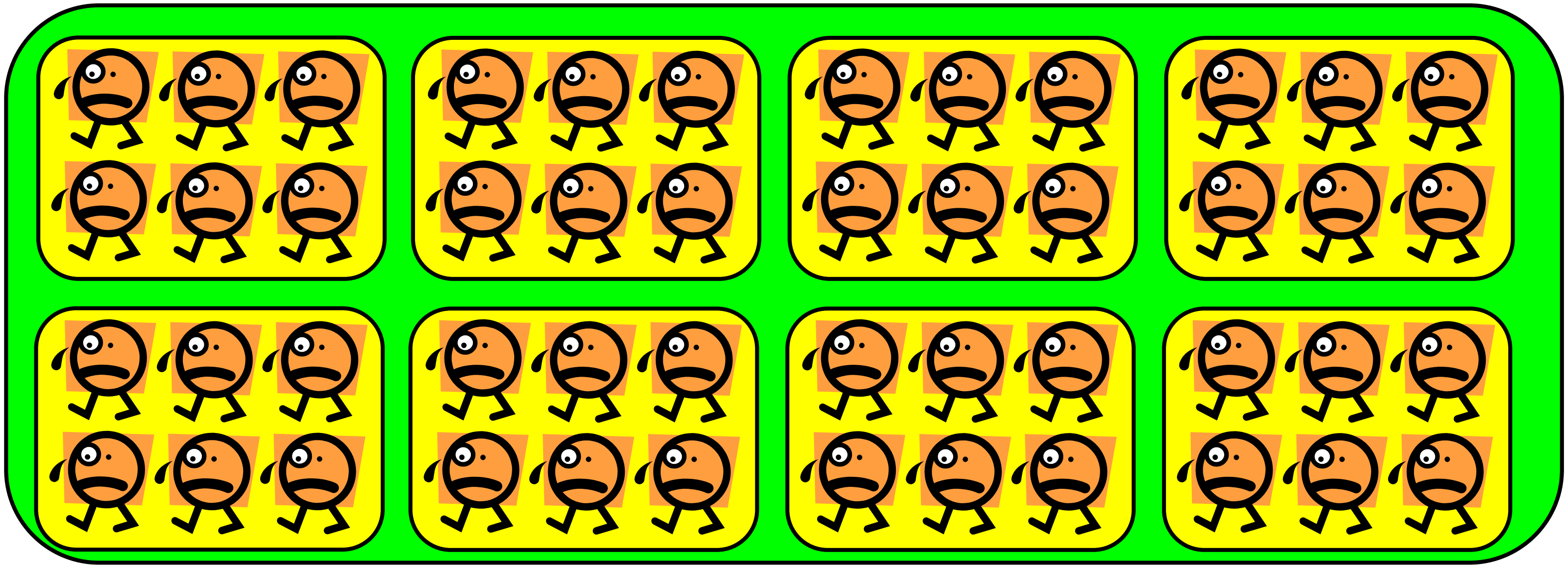
  ➔ We use m × n threads !!



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, gridDim and blockDim may have "dim3" type, 3D vector structure with x, y, z fields

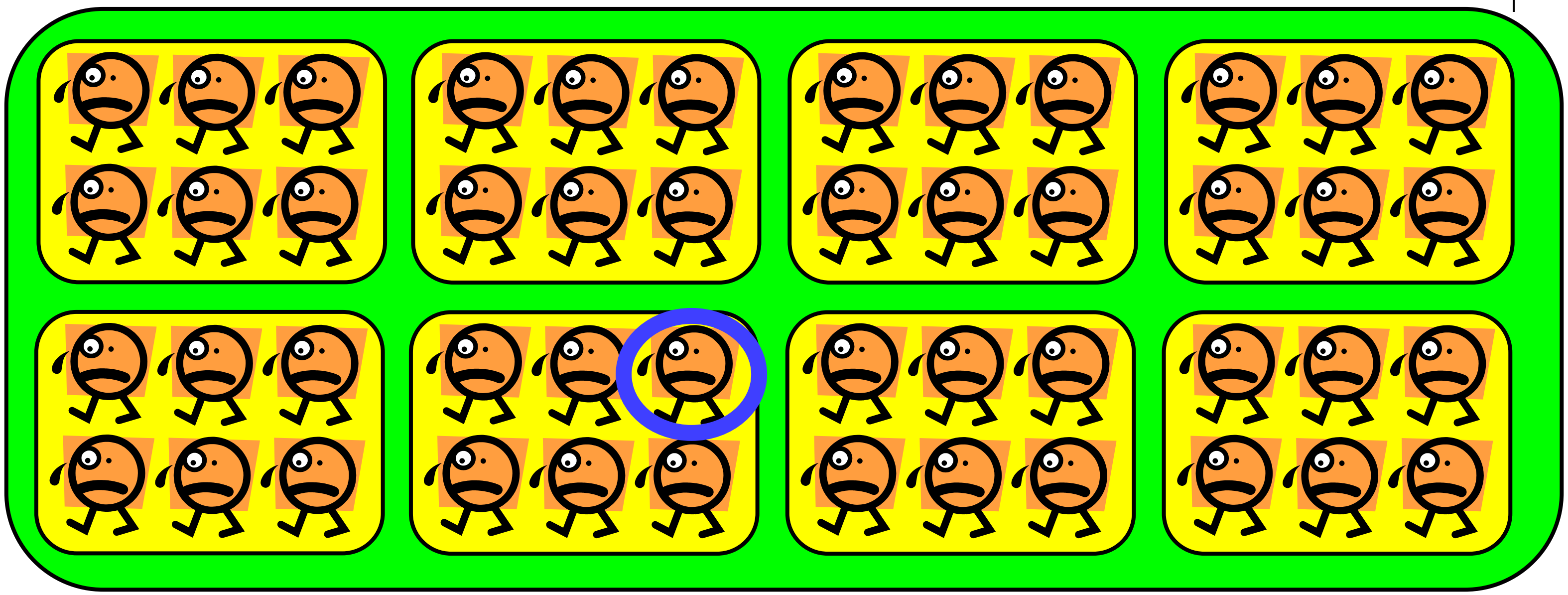
cf) func <<< dim3(4,2,1), dim3(3,2,1) >>> (); → 48 threads



※ 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

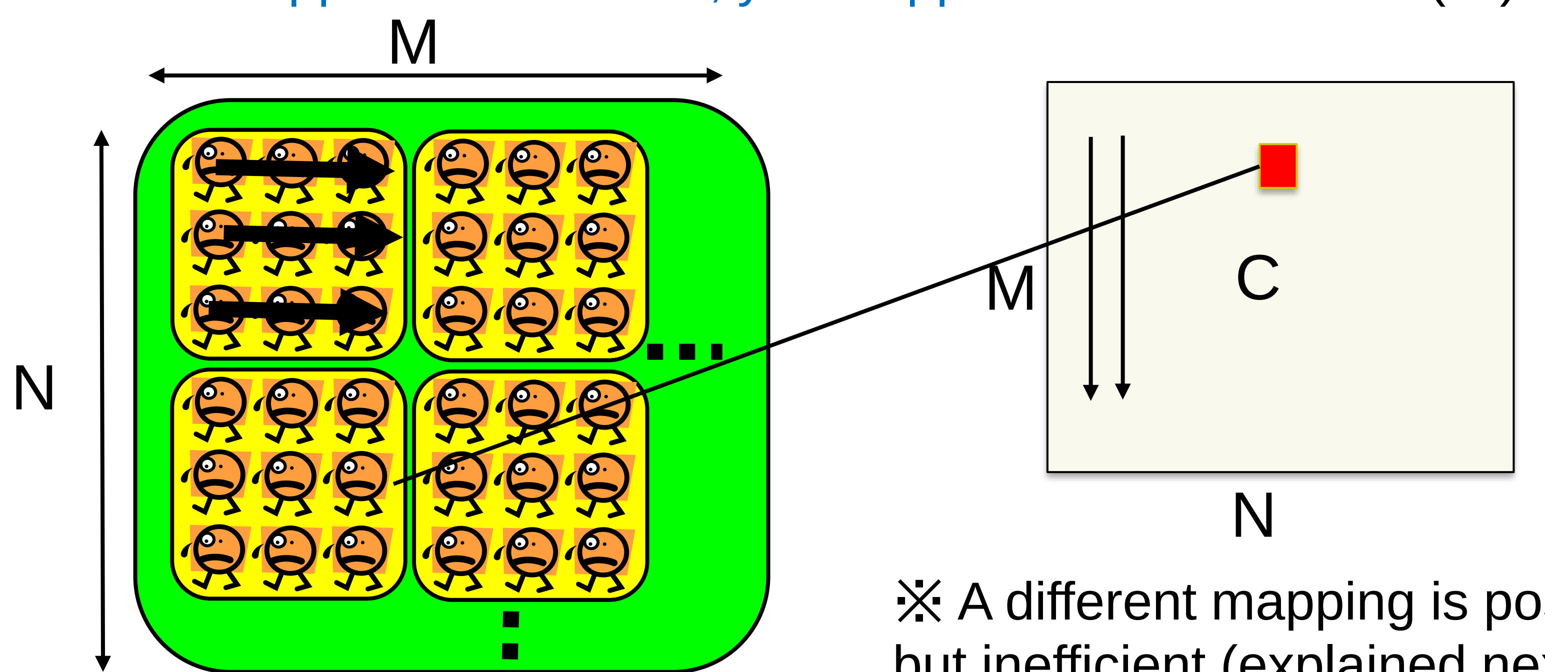
- For the thread with blue mark,

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

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

Threads in mm-cuda Sample

- The total number of threads are m*n
- How do we determine gridDim, blockDim?
  - <<<m, n>>> does not work for constraints explained later ☹
- 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 inefficient (explained next time)

Code in mm-cuda

gridDim blockDim

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

BS=16 in this sample
Actually, we use rounding up

In matmul_kernel function,

```
        :
  j = blockIdx.y * blockDim.y + threadIdx.y;
  i = blockIdx.x * blockDim.x + threadIdx.x;
        :   This thread computes Cij ← Only l loop (dot prod)
```

About Programming Efforts

- So far, mm-cuda with (m * n) threads has been explained
  - How fast is it? Please measure it (Related to [G2])

- On the other hand, codes are more complex than mm-acc

  Programmer have to
  - Call cudaMalloc, cudaMemcpy
  - Determine the number of blocks, threads
  - Determine the structure of for-loop
    - We have to change code largely when parallelization method is changed
    - In OpenACC, adding “#pragma acc loop independent” works

    :

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)>>> (...);

func<<<$\underbrace{\text{dim3(gx, gy, gz)}}_{\text{gridDim}}$, $\underbrace{\text{dim3(bx, by, bz)}}_{\text{blockDim}}$>>> (...);

gx $\leqq 2^{31}-1$

gy, gz $\leqq$ 65535

bx, by $\leqq$ 1024

bz $\leqq$64

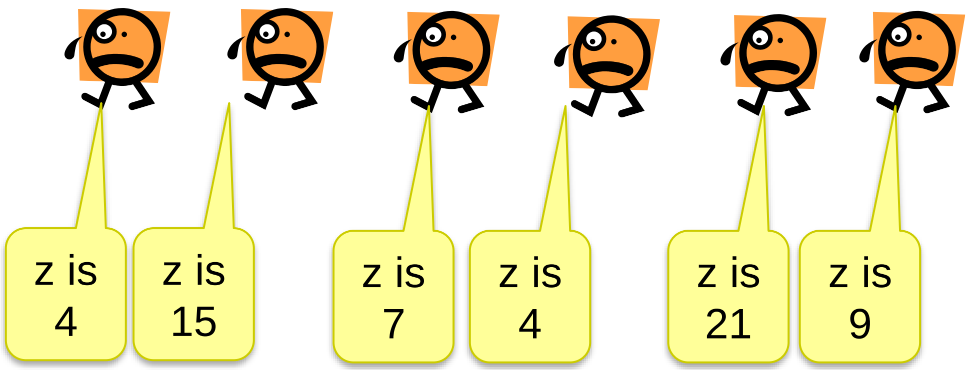
Also, bx*by*bz must be $\leqq$1024

Size of BlockDim has severe limitation ☹

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

Rules for Memory/Variables

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

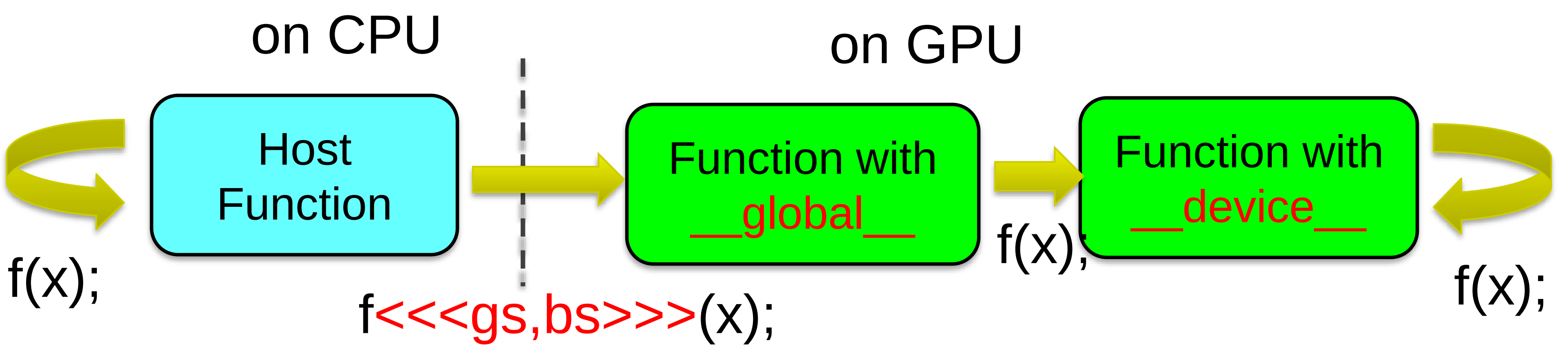


- Device memory is shared by all CUDA threads
  - Be careful to avoid race condition problem (multiple threads write same address)
  - Reading same address is ok
- Do not forget host memory and device memory are separated

Two Types of GPU Kernel Functions

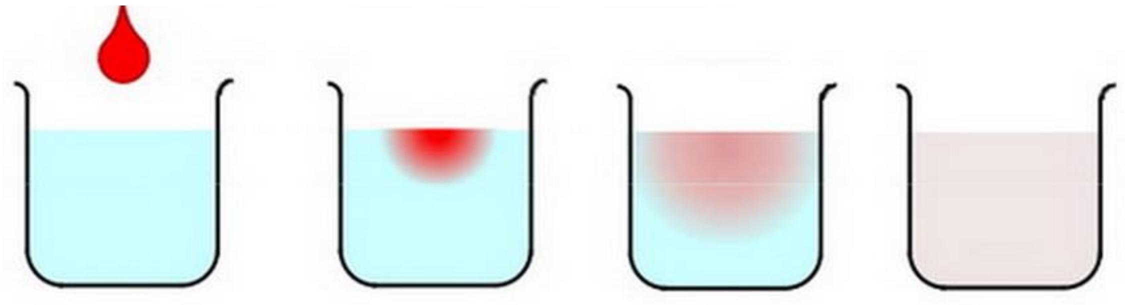
1) Functions with __global__ keyword
- "Gateway" from CPU
- Return value type must be "void"

2) Function with __device__ keyword → In OpenACC, #pragma acc routine
- Callable only from GPU
- Can have return values
- Recursive call is OK



“diffusion” Sample Program
related to [G1]

An example of diffusion phenomena:

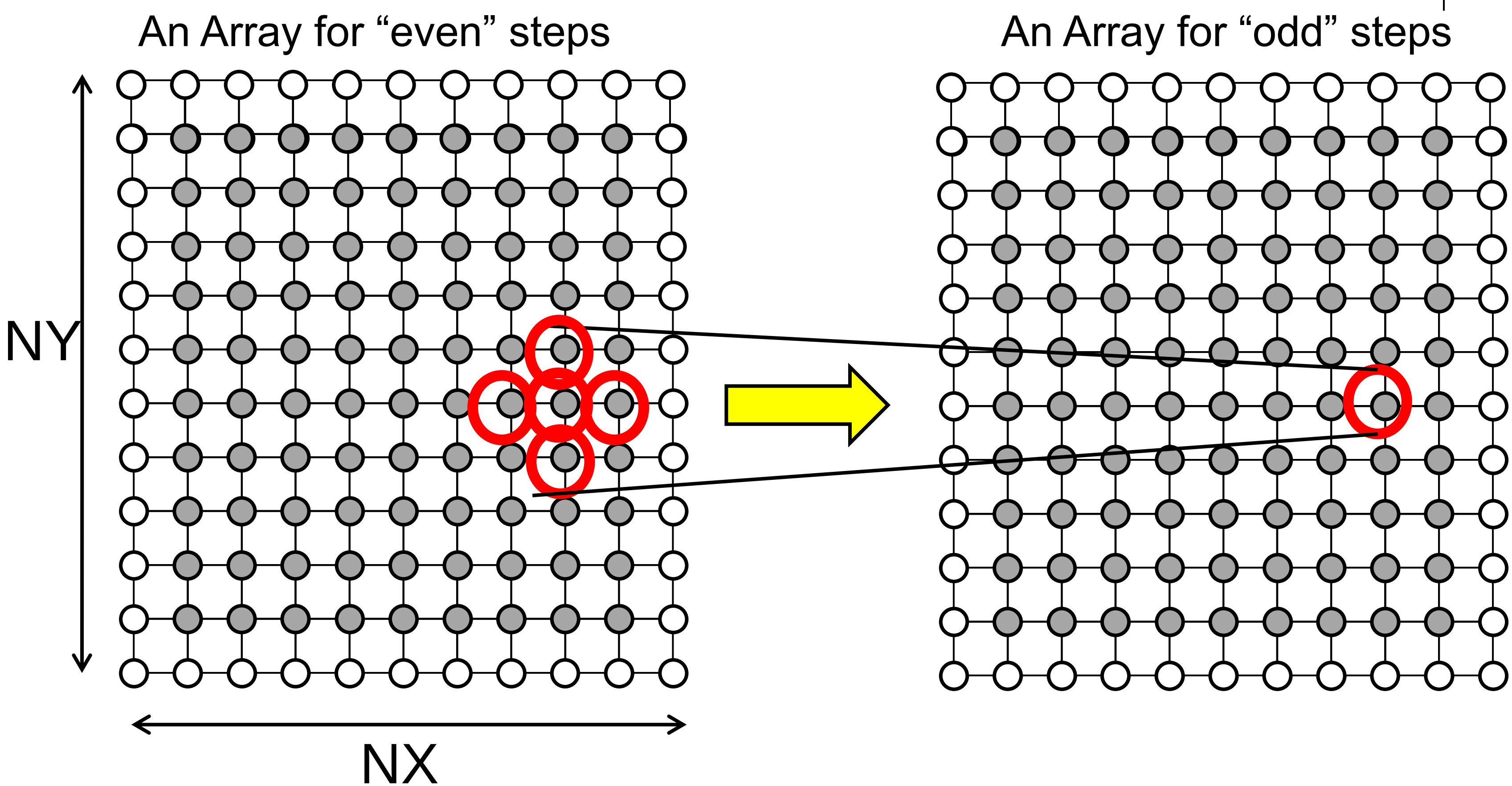


The ink spreads gradually, and finally the density becomes uniform   (Figure by Prof. T. Aoki)

Available at /gs/bs/tga-ppcomp/24/diffusion/
You can use /gs/bs/tga-ppcomp/24/diffusion-cuda/

- Execution：./diffusion [nt]
    - nt: 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 with CUDA
related to [G1]

- 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

related to [G1]

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

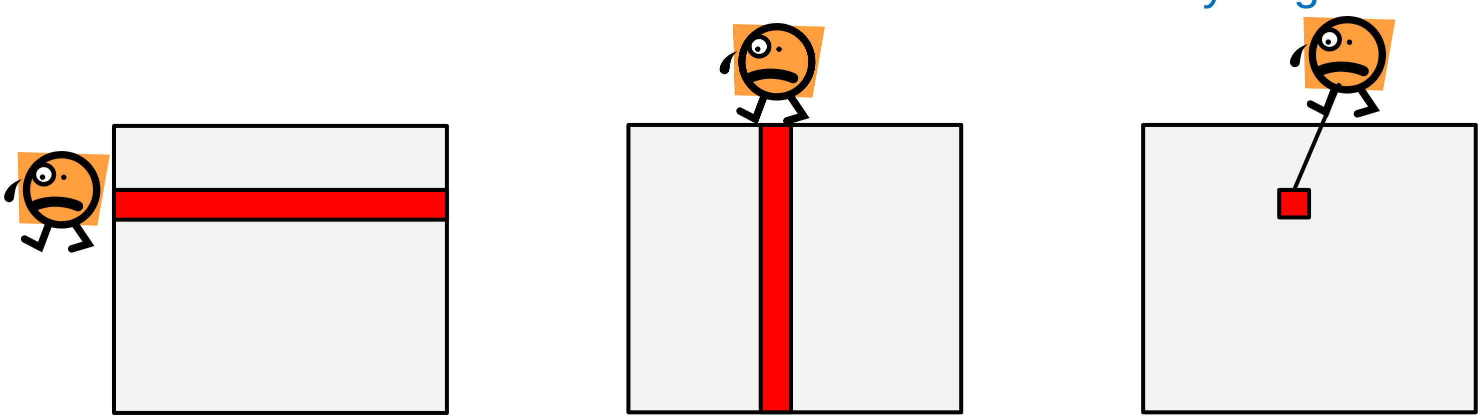
```
__global__ void calc_gpu(...) {
    :   // behavior of each thread
}

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

Calling a kernel function

Considering CUDA Threads

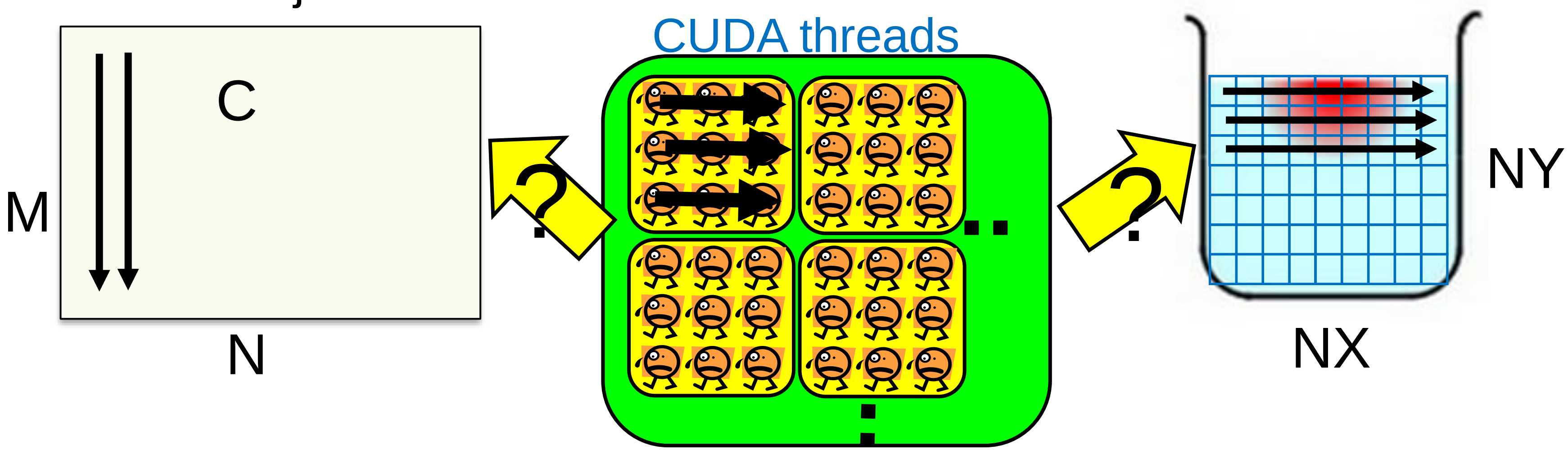
- How do we design threads on CUDA?
- There are several choices in [G1]
  - 1thread = 1row
    - We use NY threads in total → only x-loop in kernel function
  - 1thread = 1column
    - We use NX threads in total → only y-loop in kernel function
  - 1thread = 1element
    - We use NX x NY threads in total → No loop in kernel function!
    - This looks fast since the number of threads is very large



Mapping between Threads and Data

mm-cuda:
Matrices has
column-major format

diffusion:
2D array has
row-major format



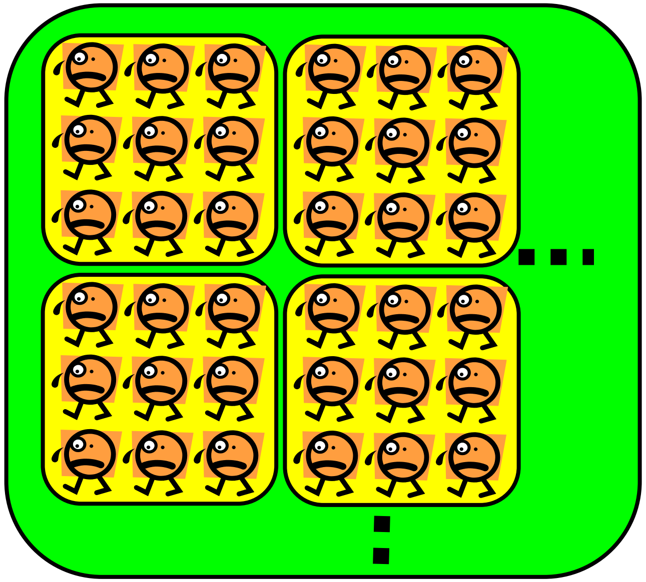

```
j = blockIdx.y * blockDim.y +
threadIdx.y;
i = blockIdx.x * blockDim.x +
threadIdx.x;
 : This thread computes Cij
```

```
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)

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



(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)

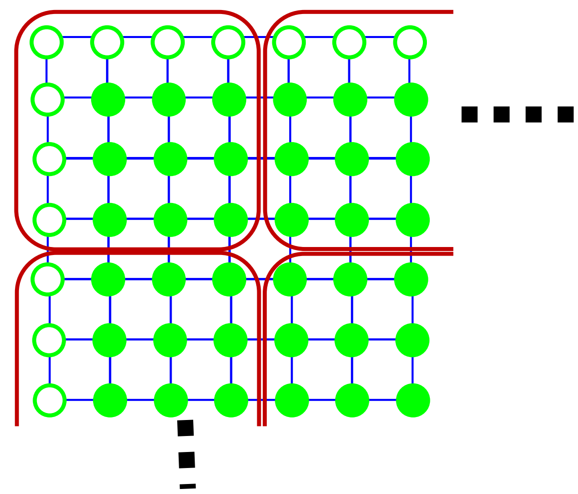
- In diffusion, Points [1, NX-1) × [1, NY-1), <u>excluded boundary</u>, should be computed

  There are choices:

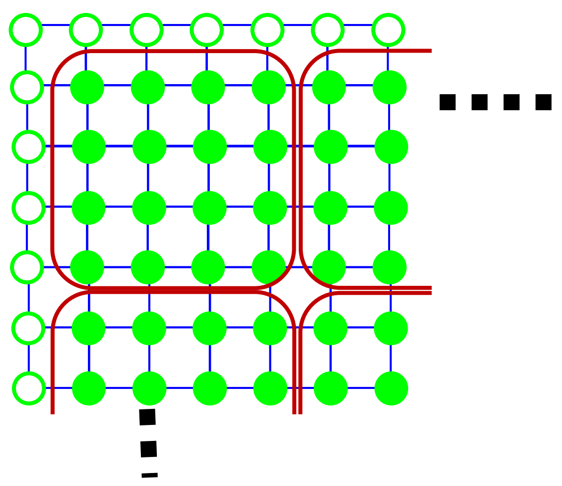
  (A) Create NX x NY threads
  - Thread (x,y) computes (x,y)
  - Threads with below IDs do nothing
    - x == 0 or y == 0 or x ≧ NX-1 or y ≧ NY-1

  (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

(A)



(B)



Either is ok ☺

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>)

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 GPU Part (Abstract)

Choose one of [G1]—[G3], and submit a report

Due date: May 30 (Thursday)

[G1] Parallelize "diffusion" sample program by OpenACC or CUDA

[G2] Evaluate speed of "mm-acc" or "mm-cuda" in detail

※ In OpenACC case, mm-meas-acc sample is useful

[G3] (Freestyle) Parallelize *any* program by OpenACC or CUDA.

For more detail, please see ppcomp-2-1 slides

Next Class:

- GPU Programming (4)
  - Discussion on speed and GPU hardware

- Schedule
  - Thu, May 16: No classes (cancelled/休講)
  - Mon, May 20: GPU (4)