2018 Practical Parallel Computing (実践的並列コンピューティング) No. 13

GPU Programming (3)

Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp

CUDA and OpenACC for GPUs

- OpenACC
  - C/Fortran + directives (#pragma acc …), Easier programming
  - PGI compiler works
    - module load pgi
    - pgcc –acc … XXX.c
  - Basically for data parallel programs with for-loops

  → Less freedom in algorithms ☹
- CUDA
  - Most popular and suitable for higher performance
  - Use “nvcc” command for compile
    - module load cuda
    - nvcc … XXX.cu

  Programming is harder, but more general

Comparing OpenMP/OpenACC/CUDA

	OpenMP	OpenACC	CUDA
Processors	CPU	CPU+GPU	CPU+GPU
File extension	.c, .cc	.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)	func<<<…, …>>>()
Derisable # 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	cudaMemcpy()
Function on GPU	-	#pragma acc routine	__global__, __device__

※ "# of XXX" = "The number of XXX"

OpenACC Programs Look Like

```
   int A[100], B[100];
   int i;
#pragma acc data copy(A,B)
#pragma acc kernels
#pragma acc loop independent
  for (i = 0; i < 100; i++) {
       A[i] += B[i];
  }
```

Executed on GPU in parallel

CUDA Programs Look Like

Sample:
~endo-t-ac/ppcomp/18/add-cuda/

```
int A[100], B[100];
int *DA, *DB;
int i;
cudaMalloc(&DA, sizeof(int)*100);
cudaMalloc(&DB, sizeof(int)*100);
cudaMemcpy(DA,A,sizeof(int)*100,
    cudaMemcpyHostToDevice);
cudaMemcpy(DB,B,sizeof(int)*100,
    cudaMemcpyHostToDevice);

add<<<20, 5>>>(DA, DB);

cudaMemcpy(A,DA,sizeof(int)*100,
    cudaMemcpyDeviceToHost);
```

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

Executed on GPU
(called a *kernel function*)

Compiling CUDA Programs/ Submitting GPU Jobs

- Compile .cu file using the NVIDIA CUDA toolkit
  - module load cuda, and then use nvcc
  - -arch sm_60 option for new GPUs

Also see Makefile in the sample directory

- Job submission method is same as OpenACC version

add-cuda/job.sh

```
#!/bin/sh
#$ -cwd
#$ -l q_node=1
#$ -l h_rt=00:10:00

./add
```

⇨ qsub job.sh

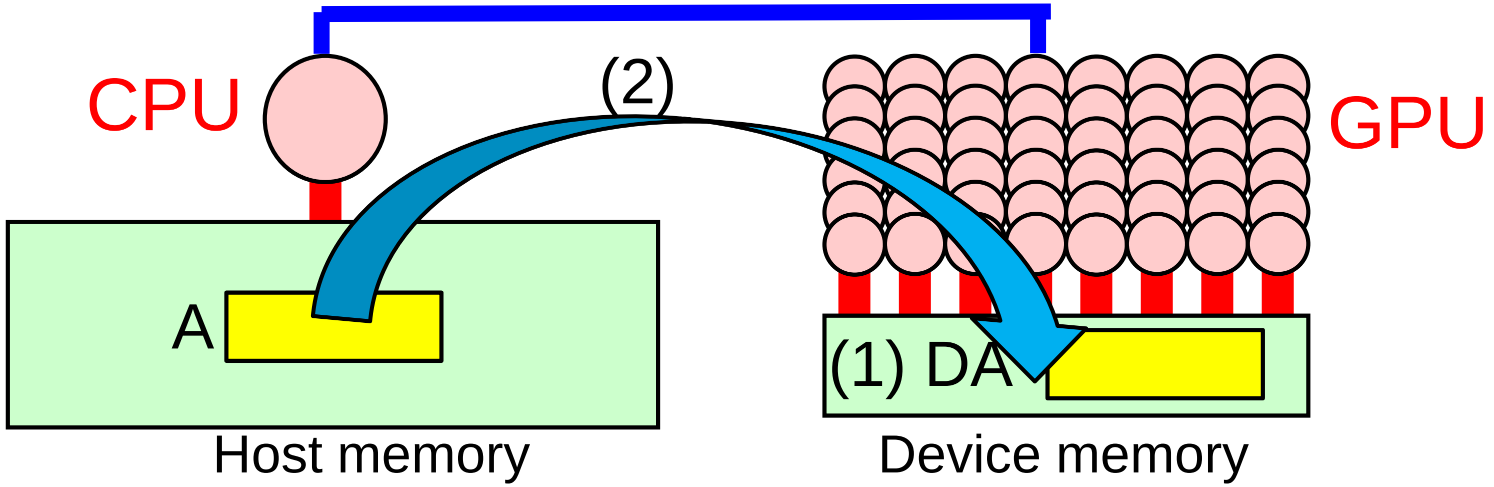
Preparing Data on Device Memory

(1) Allocate a region on device memory

cf) cudaMalloc((void**)&DA, *size*);

(2) Copy data from host to device

cf) cudaMemcpy(DA, A, *size*, cudaMemcpyHostToDevice);



Note: cudaMalloc and cudaMemcpy must be called on CPU, NOT on GPU

Comparing OpenACC and CUDA

OpenACC

Both allocation and copy are done by … data copyin

One variable name A may represent both

- A on host memory
- A on device memory

```
  int A[100]; ← on CPU
#pragma acc data copy(A)
#pragma acc kernels
 {
   … A[i] …
 }          ↖ on GPU
```

CUDA

cudaMalloc and cudaMemcpy are separated

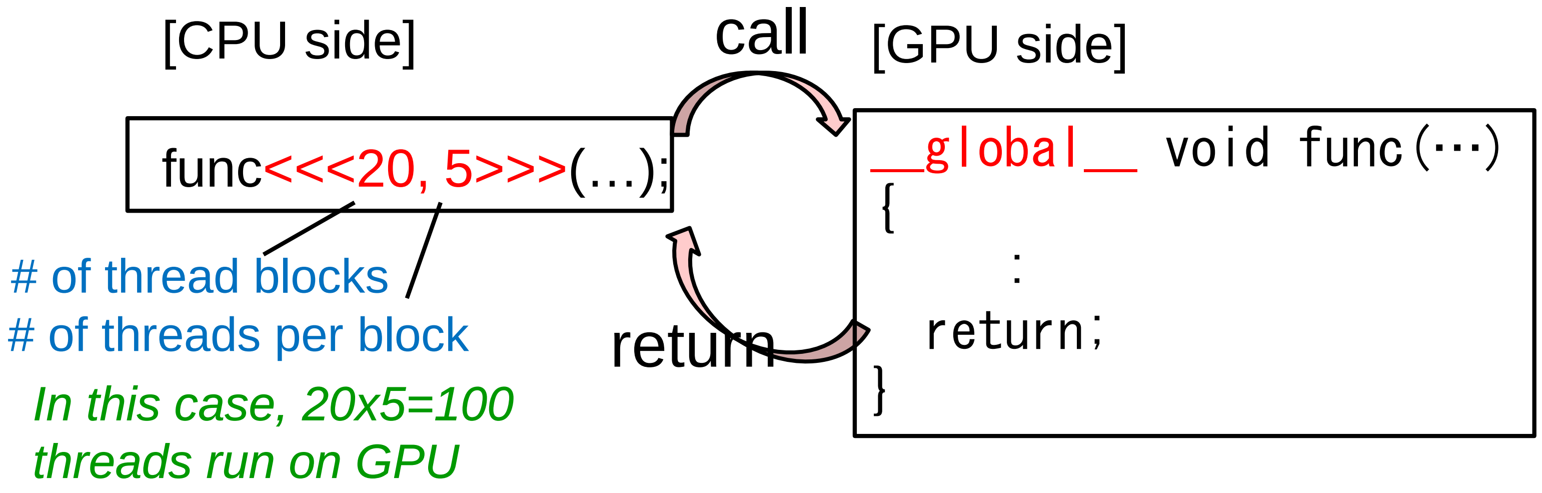
Programmer have to prepare two pointers, such as A and DA

```
int A[100];
int *DA;
cudaMalloc(&DA, …);
cudaMemcpy(DA, A, …, …);
// Here CPU cannot access DA[i]

func<<<…, …>>>(DA, …);
```

Calling A GPU Kernel Function from CPU

- A region executed by GPU must be a distinct function
    - called a GPU kernel function



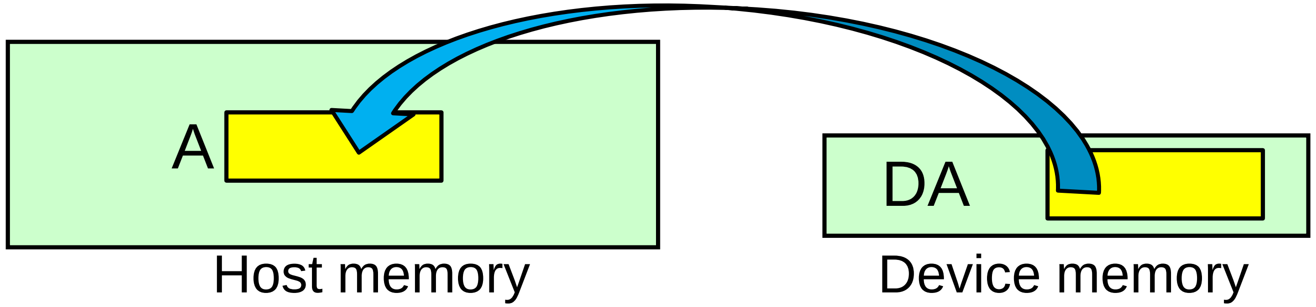
```
func<<<20, 5>>>(...);
```

```
__global__ void func(…)
{
      :
  return;
}
```

A GPU kernel function (called from CPU)

- needs __global__ keyword
- can take parameters
- can NOT return value; return type must be void

Copying Back Data from GPU

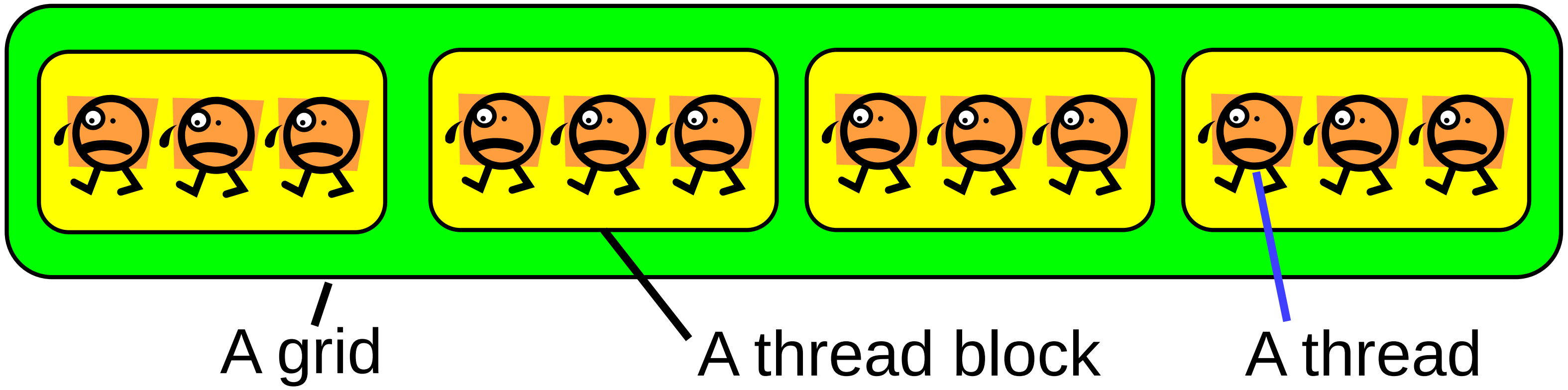


- Copy data using cudaMemcpy
  - cf) cudaMemcpy(A, DA, *size*, cudaMemcpyDeviceToHost);
  - 4th argument is one of
    - cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice, cudaMemcpyHostToHost
    - cudaMemcpyDefault ← Detect memory type automatically ☺

- When a memory area is unnecessary, free it
  - cf) cudaFree(DA);

Threads in CUDA

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

OpenACC	-	Gang	Worker	Vector lane
CUDA	Grid	Thread block	(Warp)	Thread
Hardware	GPU	SMX	(Warp)	CUDA core

To See Who am I

- By reading the following special variables, each thread can see its thread ID, etc.
- My ID
  - blockIdx.x: Index of the block the thread belong to (≧0)
  - threadIdx.x: Index of the thread (inside the block) (≧0)
- Number of thread/blocks
  - gridDim.x: How many blocks are running
  - blockDim.x: How many threads (per block) are running

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

Parallelism in add sample

- It is ok to make >1000, >10000 threads on CUDA
- We use N threads for N elements computation

```
add<<<N/BS, BS>>>(.....);
```

gridDim → N/BS

blockDim (=5 in this sample) → BS

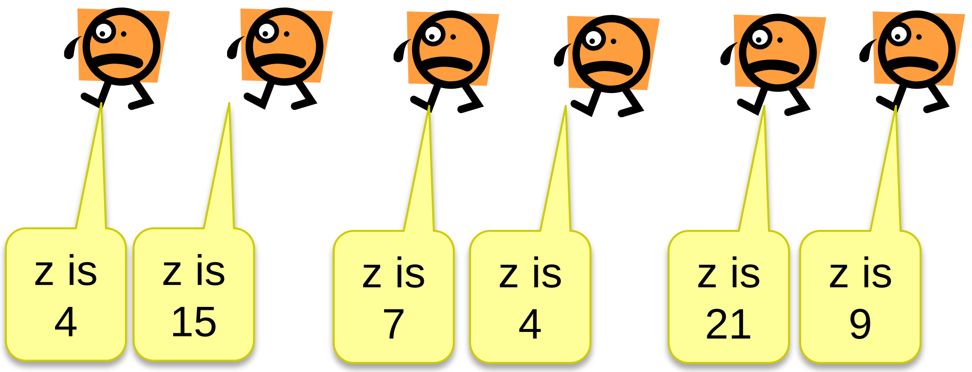
1 element for 1 thread → No need of "for" loop in this sample

Note1: <<<N, 1>>> or <<<1, N>>> also works, but speed is not good

Note2: To support the case N is indivisible by BS, we should use
<<<(N+BS-1)/BS, BS>>>
→But # of threads may be larger N. "Extra" threads (id≧N) should not work. See add-cuda/add2.cu

Rules for Memory/Variables

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

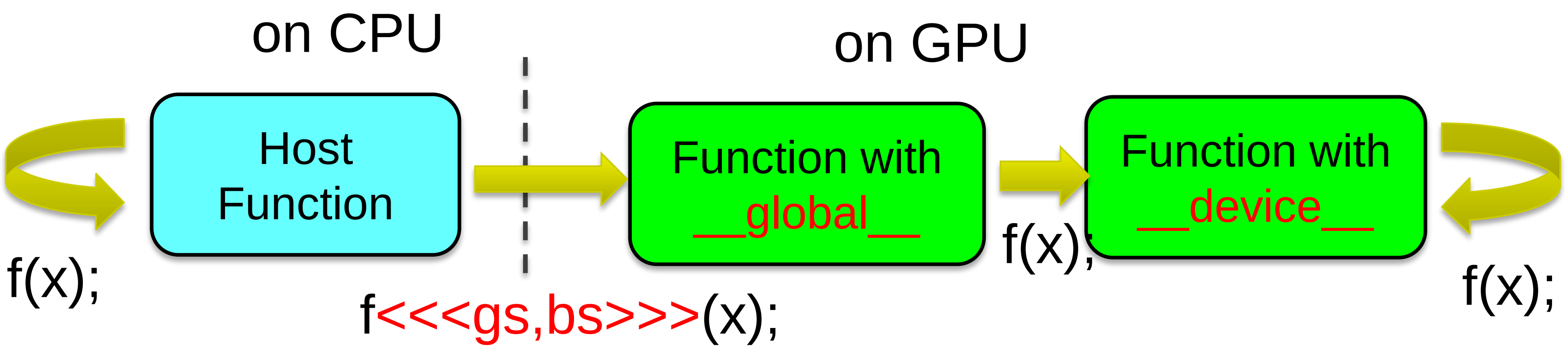


- Device memory is shared by all CUDA threads
  - Be careful to avoid race condition problem (multiple threads write same address)
  - Reading same address is ok
- Do not forget host memory and device memory are distributed

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
    - like OpenACC
  - Exceptionally, printf() is OK
    - unlike OpenACC ☺
  - Calling malloc()/free() on GPU is OK, if the size is small
    - If we need large regions on device memory, call cudaMalloc() from CPU

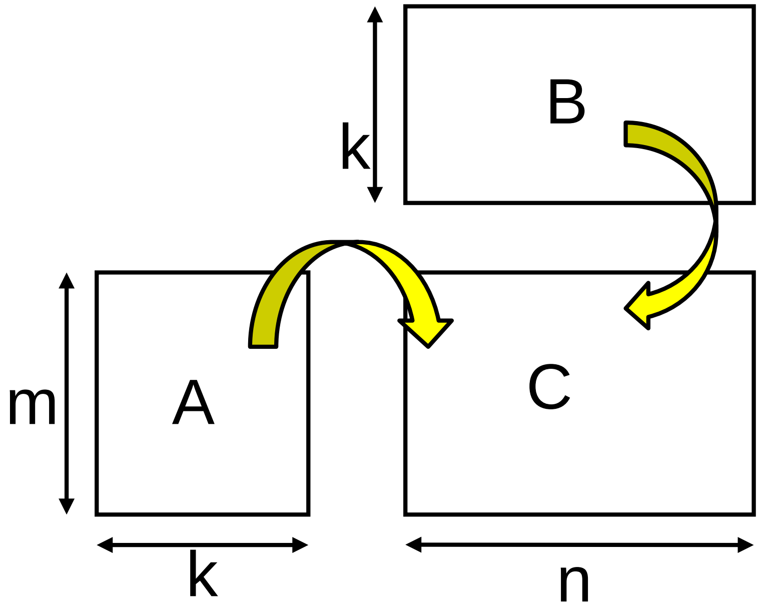
"mm" sample: Matrix Multiply (Revisited, related to [G2])

CUDA version available at ~endo-t-ac/ppcomp/18/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.
  - Each matrix is expressed as a 1D array by *column-major* format
- 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

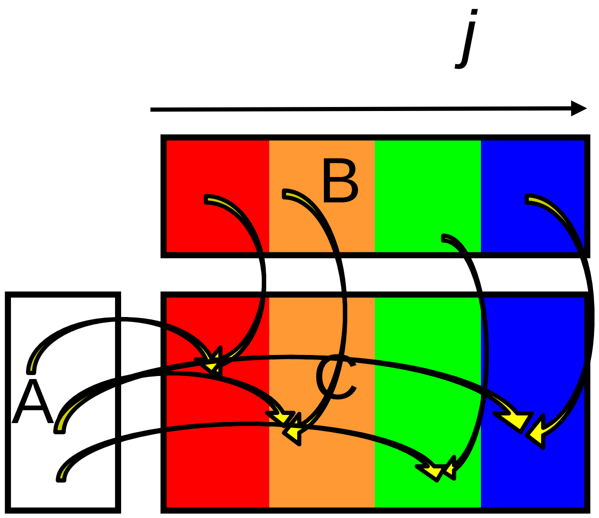
How We Parallelize Computation

In mm, we can compute different C elements in parallel

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

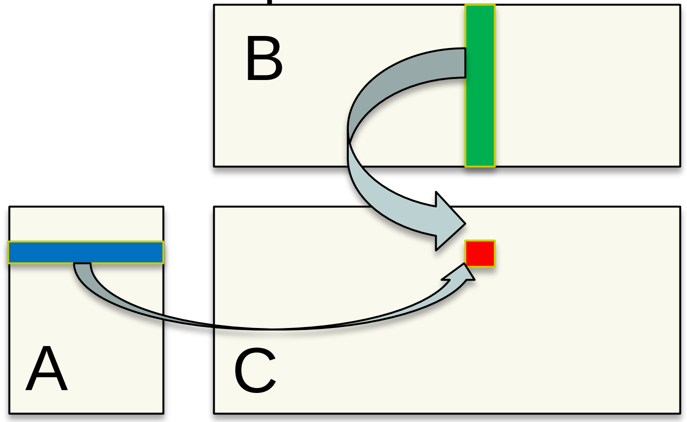
OpenMP

- Parallelize column-loop (or row-loop)



CUDA

- We can create too many threads
  → M x N threads are ok!!
- Parallelize row&column of C
- 1 thread computes 1 element

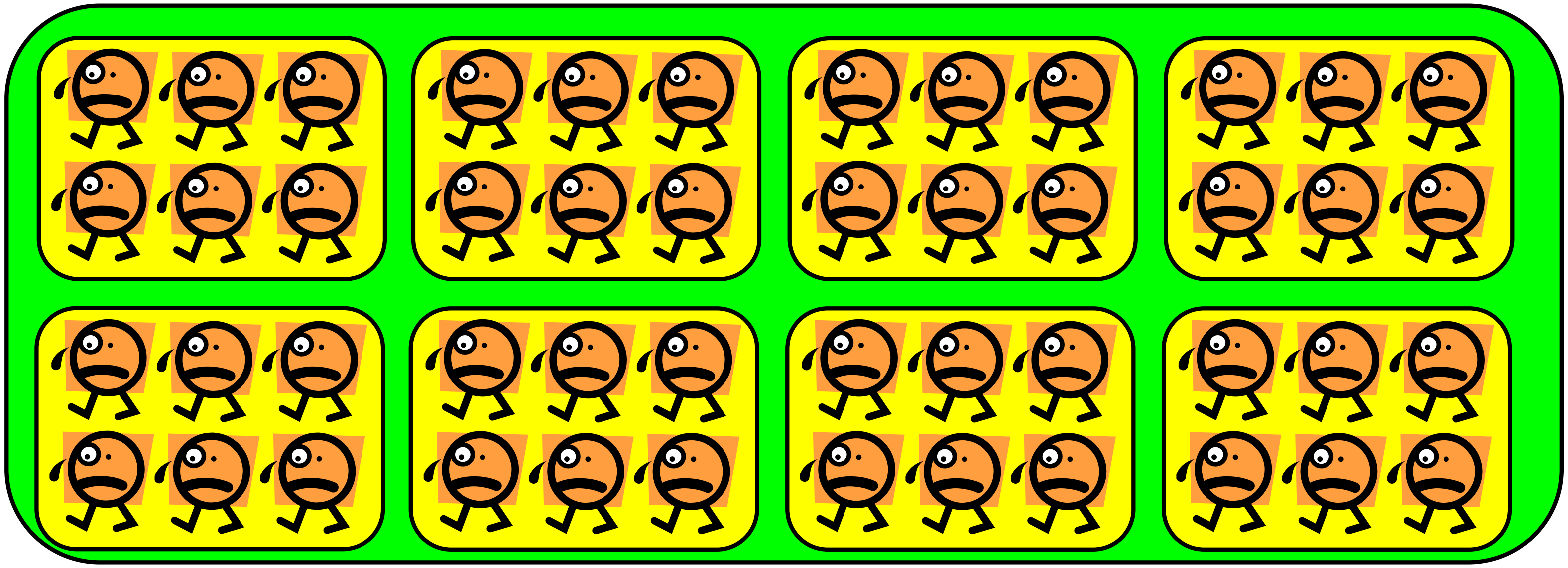


※ This is not the unique way

Creating Many Threads

- Now we want to make M*N (may be >1,000,000) threads
  - <<<(M*N)/BS, BS>>> is ok, but…
- On CUDA, gridDim and blockDim may have "dim3" type (3D vector structure with x, y, z fields)

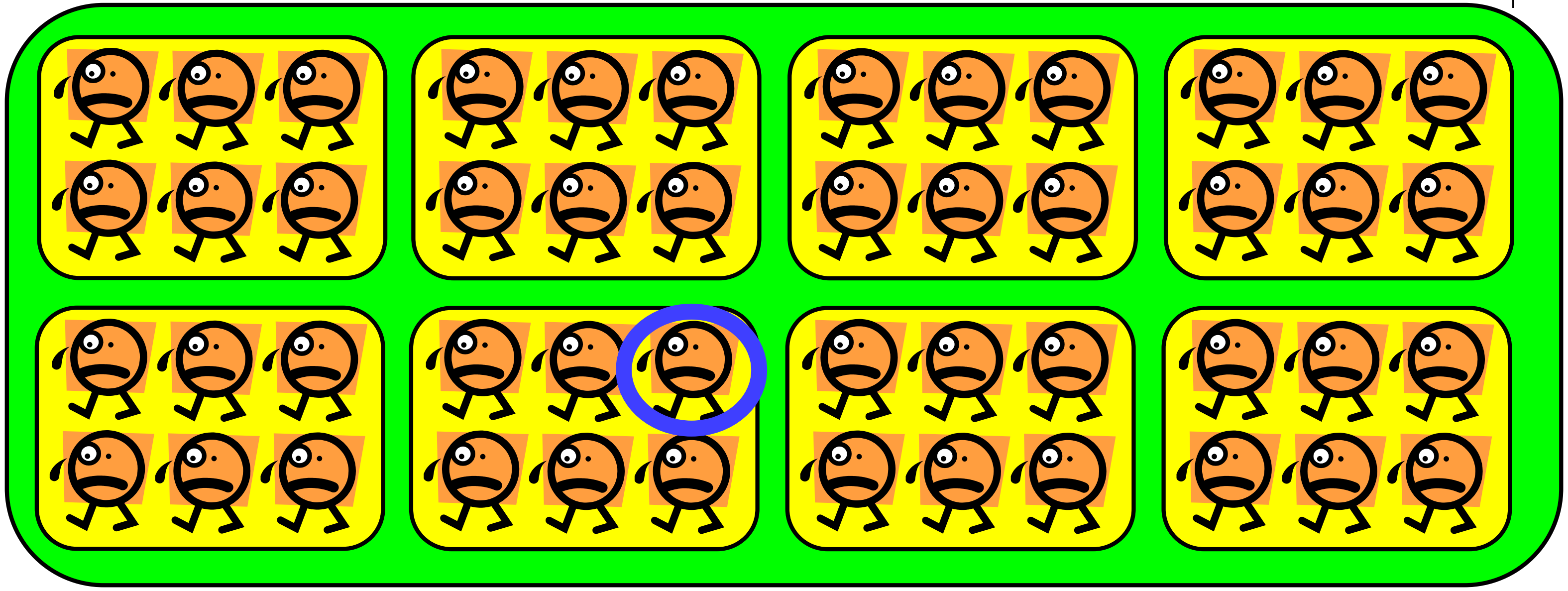
cf) func <<< dim3(4,2,1), dim3(3,2,1) >>> (); → 48 threads



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

Thread IDs in multi-dimensional cases

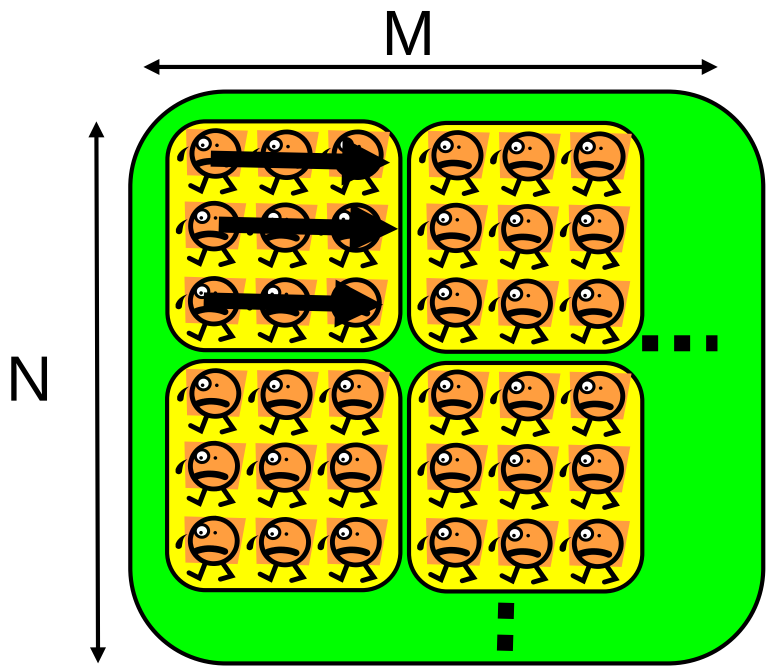
In the case of func <<< dim3(4,2,1), dim3(3,2,1) >>> ();

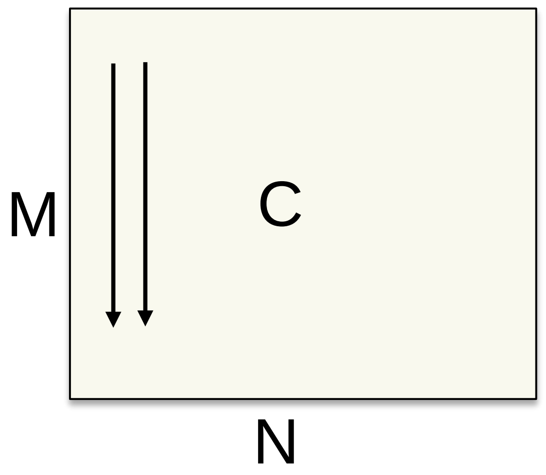


- For every thread,
  gridDim.x=4, gridDim.y=2, gridDim.z=1
  blockDim.x=3, blockDim.y=2, blockDim.z=1
- For the thread with blue mark,
  blockIdx.x=1, blockIdx.y=1, blockIdx.z=0
  threadIdx.x=2, threadIdx.y=0, threadIdx.z=0

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 use fixed blockDim (x=16, y=16 → 256 threads per block)
  - gridDim is computed from M, N
- x is mapped to column index, y is mapped to row index (※)





※ A different mapping is possible, but inefficient (in the next class)

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
```

Limitations on Number of Threads

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

≦ 2^{31}-1

≦ 65535

≦ 1024

≦64

Also, bx*by*bz must be ≦1024

BlockDim has severe limitation ☹
That is why mm-cuda uses fixed BlockDim (16x16x1)

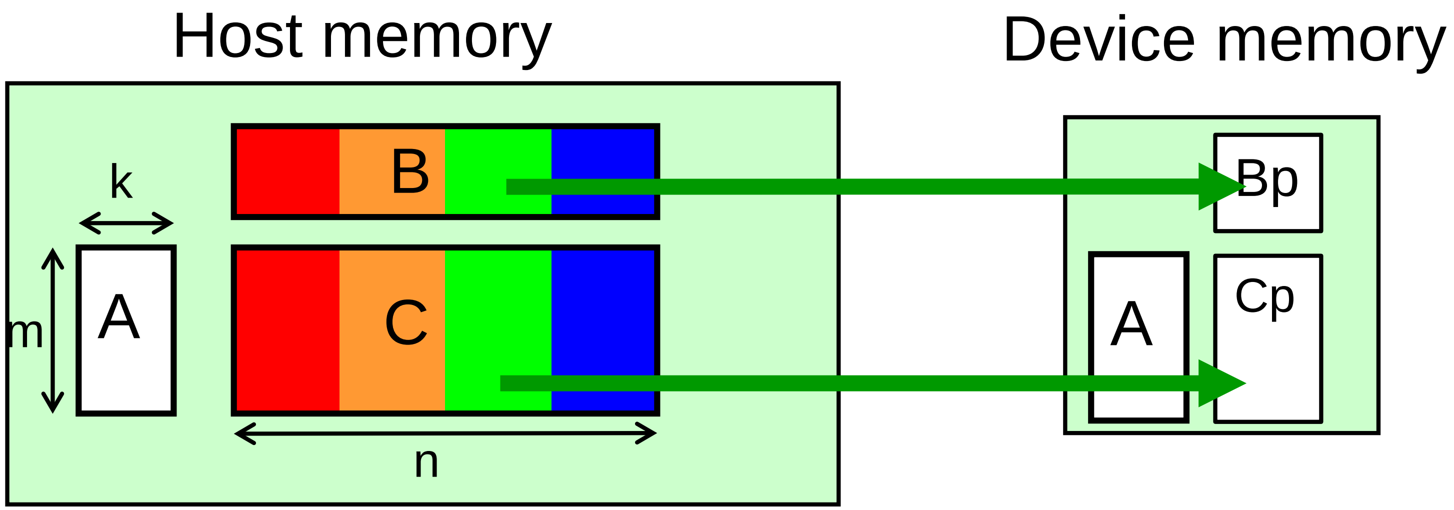
Notes in Time Measurement

- 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
        - “non-blocking” like MPI_Isend, MPI_Irecv

Larger Matrix Multiply (Concept, Related to [G2])

mm fails with too large m, n, k, since cudaMalloc fails

- such as ./mm 2000 600000 2000

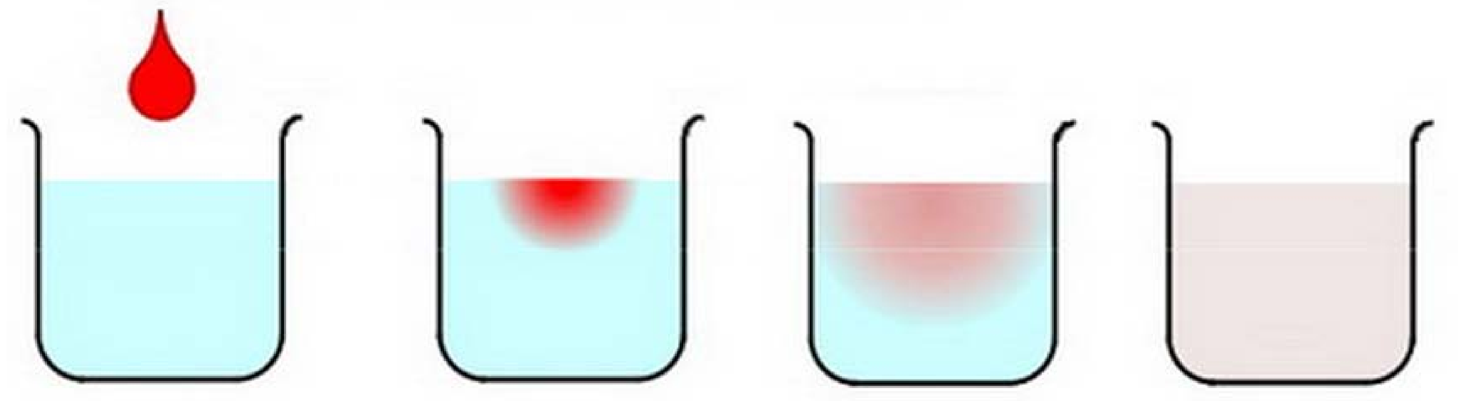


- Dividing large matrices will solve the issue
  - Do we need to transfer A each step?
  - We do not need Bp/Cp on host

“diffusion” Sample Program (1) (Revisited, related to [G1])

An example of diffusion phenomena:

•Pour a drop of ink into a water glass



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

- Density of ink in each point vary according to time → Simulated by computers
- Stencil computation

How Do We Parallelize “diffusion” Sample?

Parallelization method with OpenMP :

[Algorithm] Parallelize spatial (Y or X) for-loop

- “1 parallel region = 1 time step” is easier
- Each thread computes its part in the space
- Time (T) for-loop cannot be parallelized, due to dependency

[Data] Data structure is same as sequential version

With CUDA:

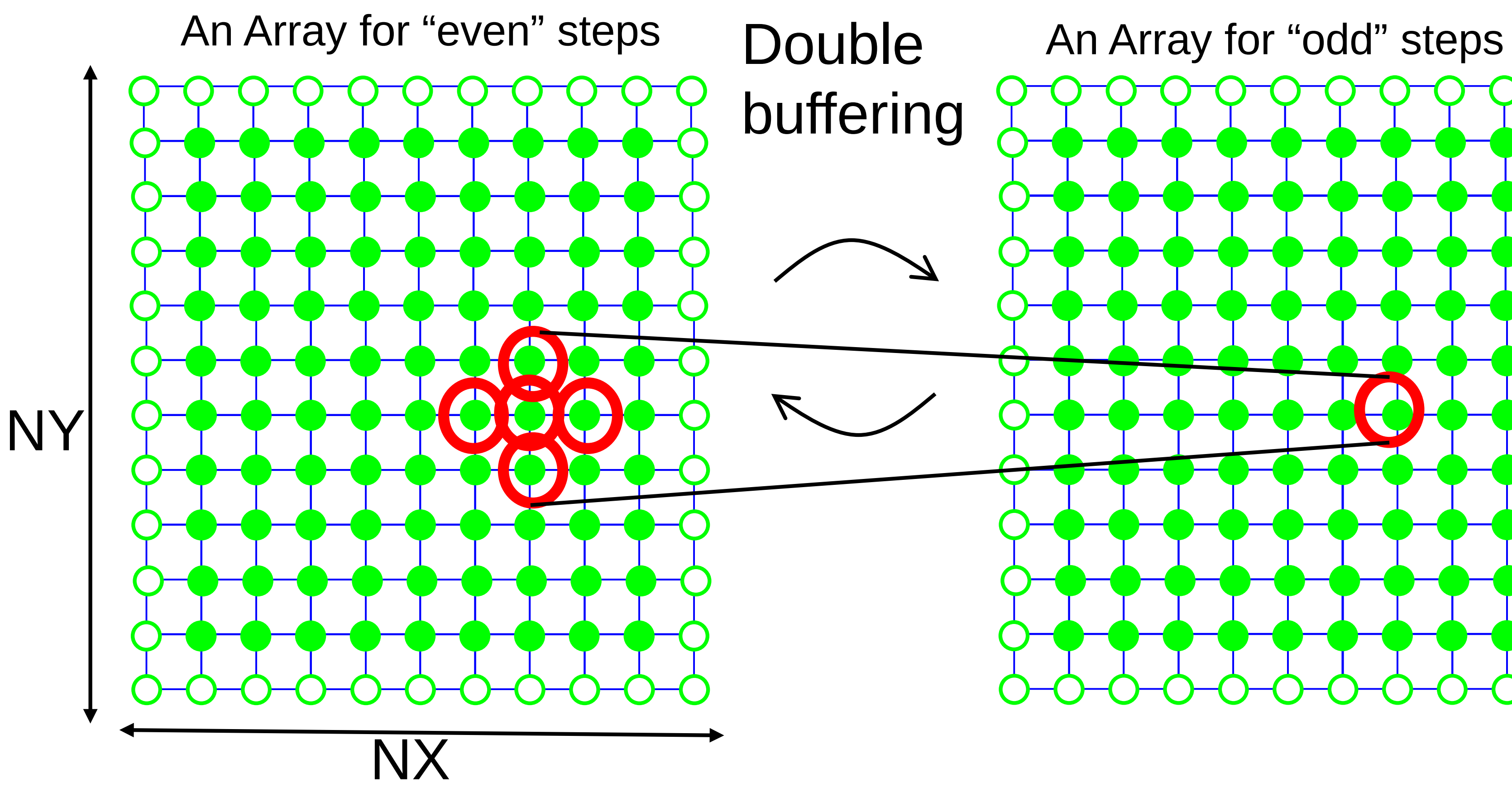
[Algorithm] Similar policy as OpenMP version

- “1 GPU kernel function call = 1 time step” is easier
- Unlike OpenMP, “1 thread = 1 point” policy is ok

[Data] Data structure is same as sequential version, but…

- When should we do cudaMemcpy?

Parallelize "diffusion" Sample



- In diffusion, computation of a new point requires 5 old points (5-point stencil)
- Points on boundary are exceptional. In this sample, no computation is done

Considering gridDim/blockDim

- Points [1, NX-1) × [1, NY-1), excluded boundary, should be computed.

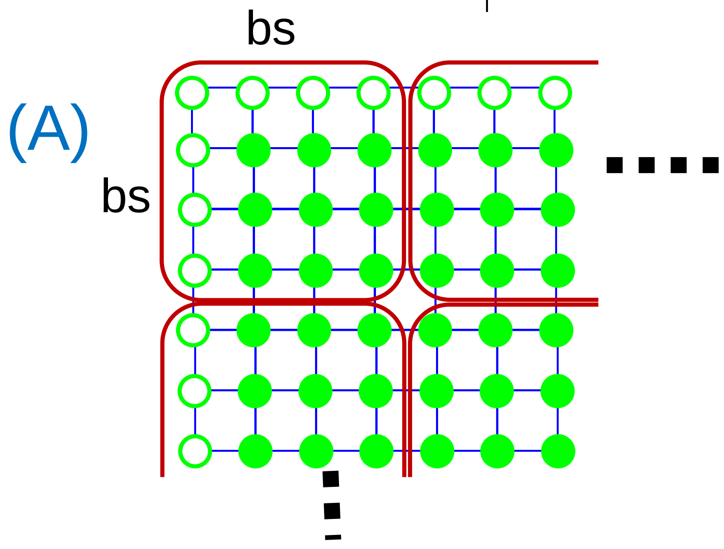
  There are choices:

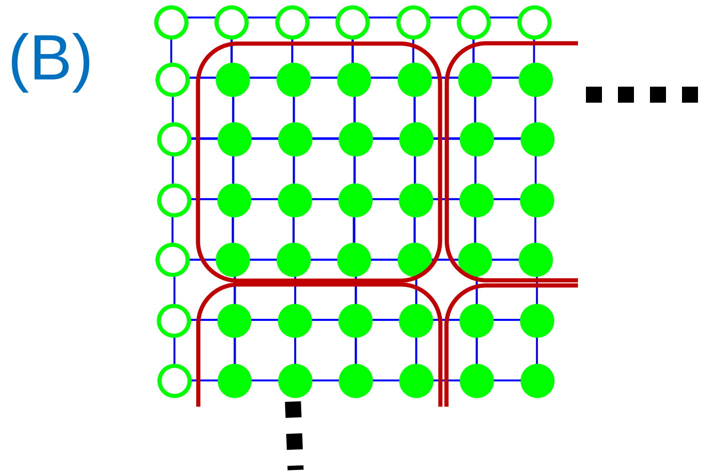
  (A) Create NX x NY threads

  (B) Create (NX-2) x (NY-2) threads
- For gridDim/blockDim, using "dim3" type would be a good idea

```
int bs =16
…<<< dim3(NX/bs, NY/bs, 1),
dim3(bs,bs,1)>>>…
```

  - Actually, we need rounding up and excluding extra threads
  - "mm-cuda" sample is a hint
- On the other hand, <<<NX, NY>>> is not good ☹
  - BS must be 1024 or less

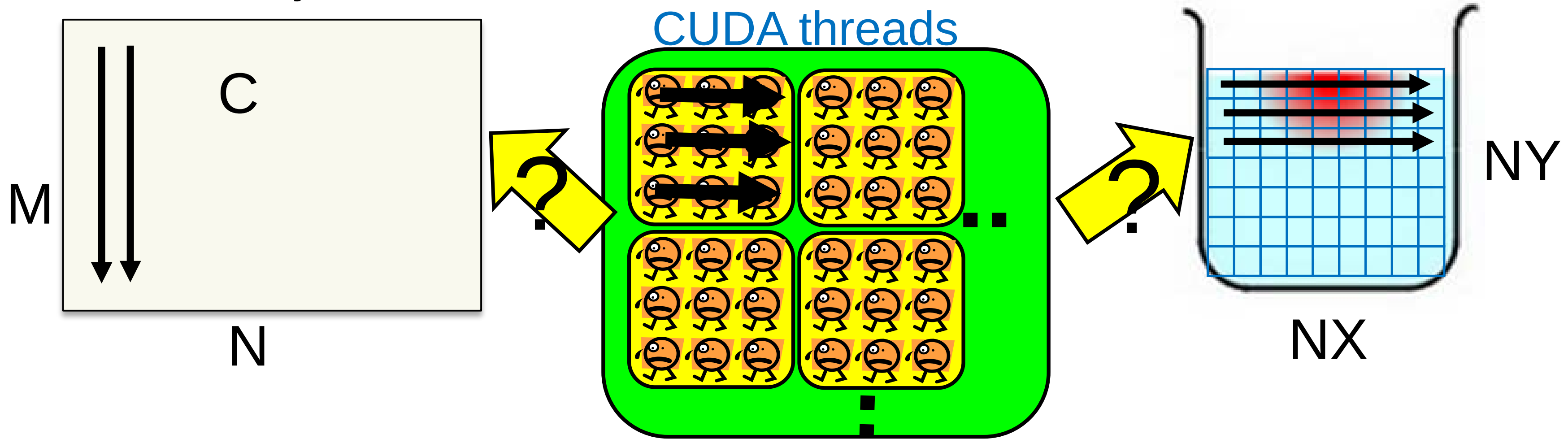




Mapping between Threads and Data

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?

Assignments in this Course

- There is homework for each part. Submissions of reports for 2 parts are required
- Also attendances will be considered

Part 1 OpenMP	[O1] diffusion [O2] sort [O3] free	*Select 1 problem*
Part 2 MPI	[M1] diffusion [M2] mm [M3] free	*Select 1 problem*
Part 3 GPU	[G1] diffusion [G2] mm [G3] free	*Select 1 problem*

Select 2 parts

Assignments in GPU Part (Abstract)

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

Due date: June 14 (Thursay)

[G1] Parallelize “diffusion” sample program by OpenACC or CUDA

[G2] Improve “mm-acc” or “mm-cuda” to support larger matrices

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

Notes in Submission

- Submit the followings via OCW-i

  (1) A report document

  - A PDF or MS-Word file, 2 pages or more
  - in English or Japanese (日本語もok)

  (2) Source code files of your program

  - If you use multiple files, you can use ".zip" or ".tgz"

- Report should include:
  - Which problem you have chosen
  - How you parallelized
    - It is even better if you mention efforts for high performance or new functions
  - Performance evaluation on TSUBAME
    - With varying number of processor cores
    - With varying problem sizes
    - Discussion with your findings
    - Other machines than TSUBAME are ok, if available

Next Class:

- GPU Programming (4)
  - Performance of GPU programs (OpenACC/CUDA)