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

Part2: GPU (3)
June 4, 2020

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
  - 3 classes

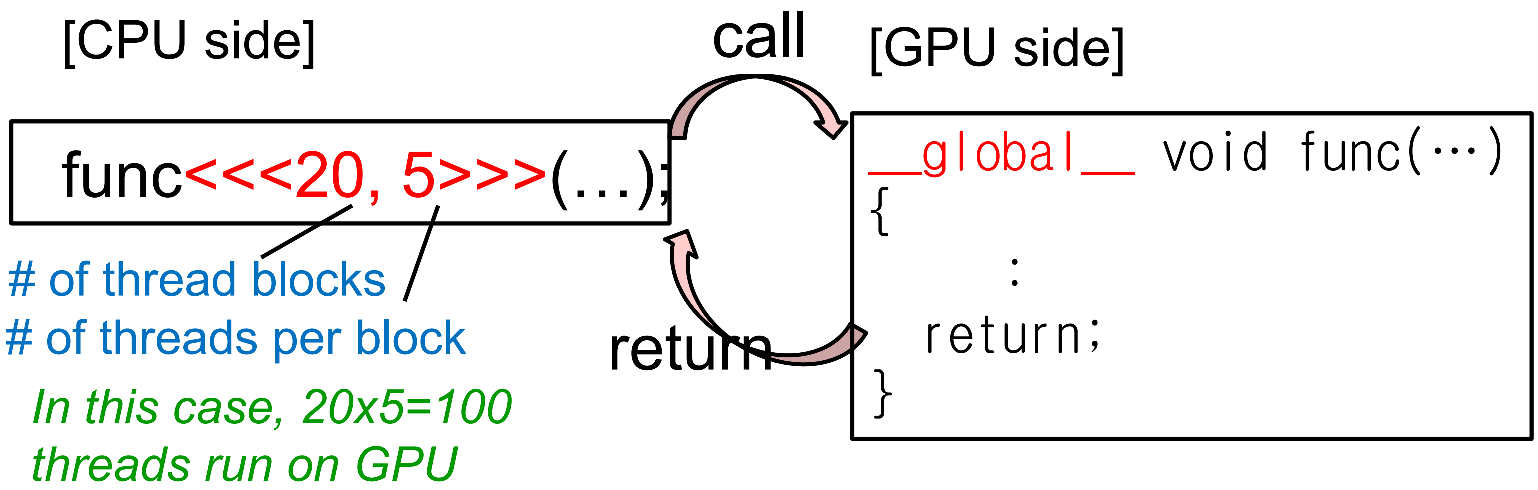
Comparing OpenMP/OpenACC/CUDA

<table>
<tr><th></th><th>OpenMP</th><th>OpenACC</th><th>CUDA</th></tr>
<tr><td>Processors</td><td>CPU</td><td colspan="2">CPU+GPU</td></tr>
<tr><td>File extension</td><td colspan="2">.c, .cc</td><td>.cu</td></tr>
<tr><td>To start parallel (GPU) region</td><td>#pragma omp parallel</td><td>#pragma acc kernels</td><td rowspan="2">func<<<…, …>>>()</td></tr>
<tr><td>To specify # of threads</td><td>export OMP_NUM _THREADS=…</td><td>(num_gangs, vector_length etc)</td></tr>
<tr><td>Desirable # of threads</td><td># of CPU cores or less</td><td colspan="2"># of GPU cores or “more”</td></tr>
<tr><td>To get thread ID</td><td>omp_thread_num()</td><td>-</td><td>blockIdx, threadIdx</td></tr>
<tr><td>Parallel for loop</td><td>#pragma omp for</td><td>#pragma acc loop</td><td>-</td></tr>
<tr><td>Task parallel</td><td>#pragma omp task</td><td>-</td><td>-</td></tr>
<tr><td>To allocate device memory</td><td>-</td><td>#pragma acc data</td><td>cudaMalloc()</td></tr>
<tr><td>To copy to/from device memory</td><td>-</td><td>#pragma acc data
#pragma acc update</td><td>cudaMemcpy()</td></tr>
<tr><td>Function on GPU</td><td>-</td><td>#pragma acc routine</td><td>__global__,__device__</td></tr>
</table>

※ “# of XXX” = “The number of XXX”

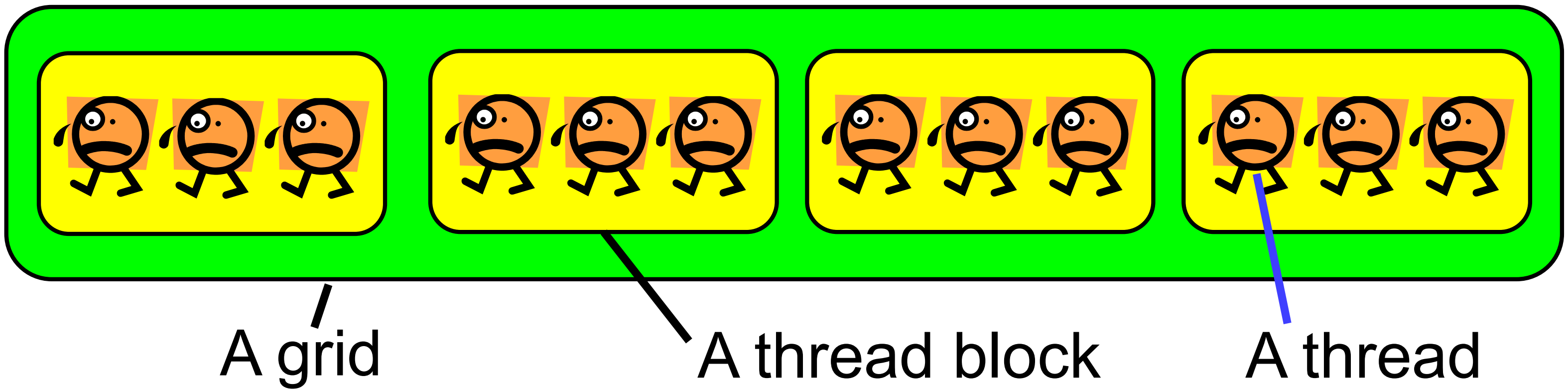
Calling A GPU Kernel Function from CPU

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



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

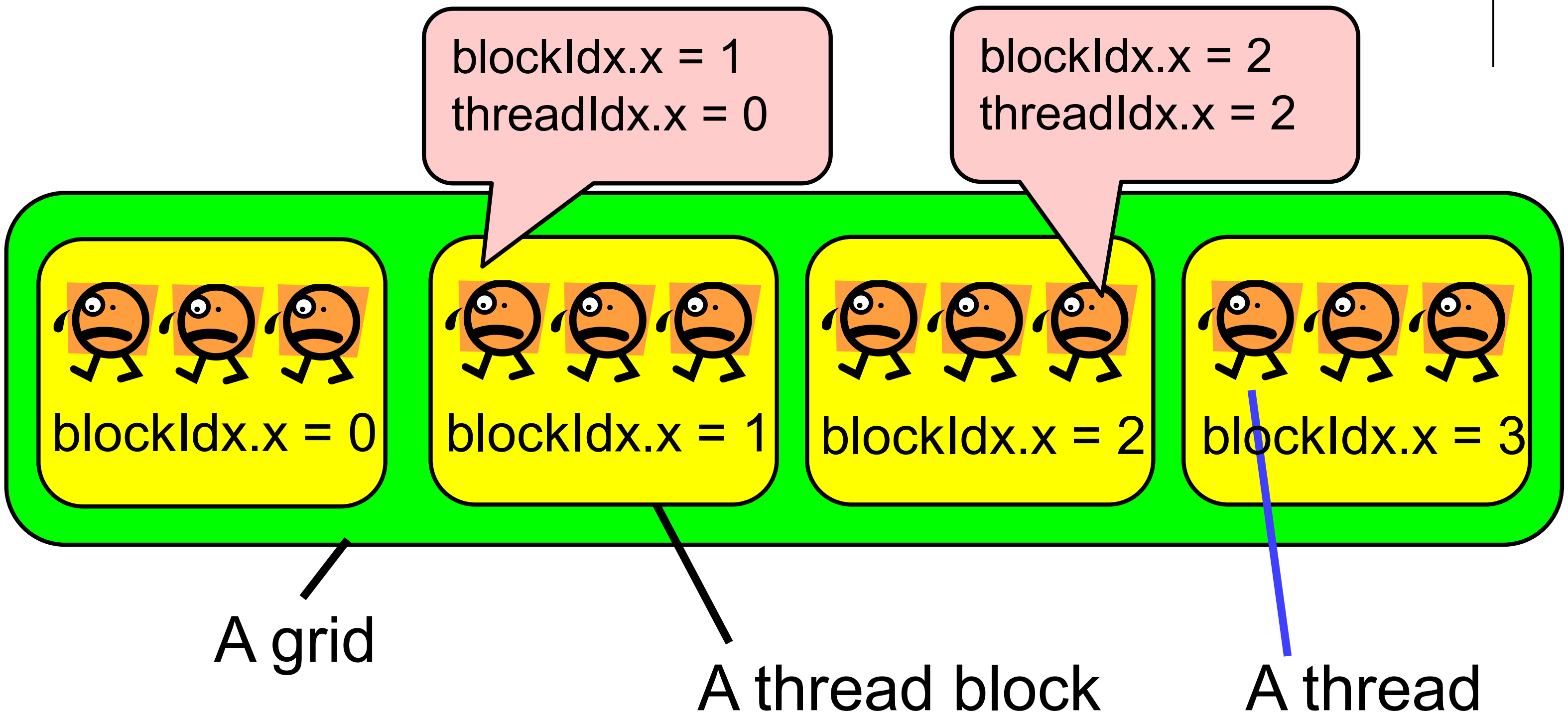
The reason is related to GPU hardware

Thread block ⇔ SMX, Thread ⇔ CUDA core

To See Who am I

- By reading the following special variables, each thread can see its thread ID in GPU kernel function
- 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

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

How Number of Threads is Designed? (1)

On CUDA, Different strategy is required from on OpenMP

- On OpenMP, number of threads (OMP_NUM_THREADS) should be ≦ CPU cores
  - The number is basically determined by hardware
  - ≦7 on q_node node, ≦28 on f_node
- On CUDA, it is better to use number of thread ≧ GPU cores
  - ≧ 3584 on TSUBAME3's P100 GPU
  - You can use >1,000,000 threads!

How Number of Threads is 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

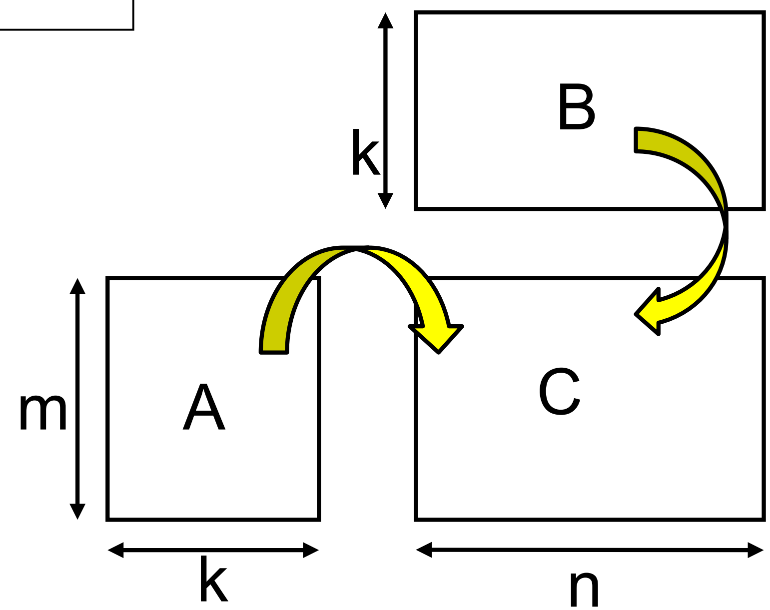
(3)Then block number is P/BS

- We consider indivisible cases later

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

CUDA versions are at
- /gs/hs1/tga-ppcomp/20/mm-v1-cuda/
- /gs/hs1/tga-ppcomp/20/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

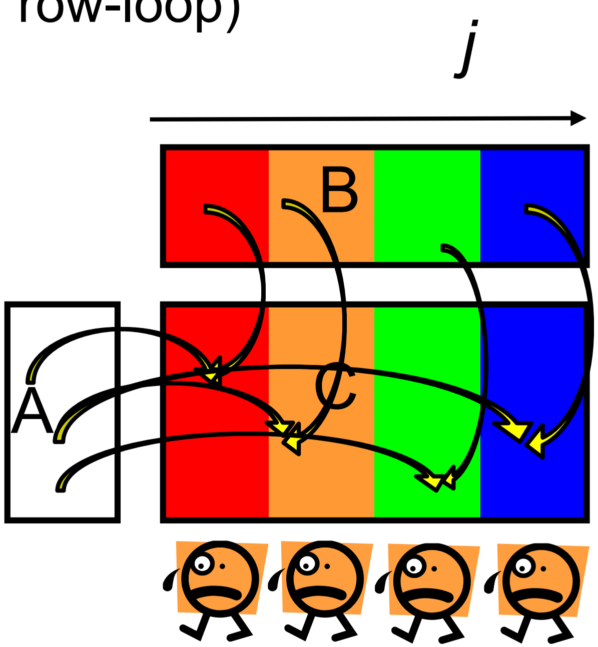
How We Parallelize Computation

In mm, we can compute different C elements in parallel

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

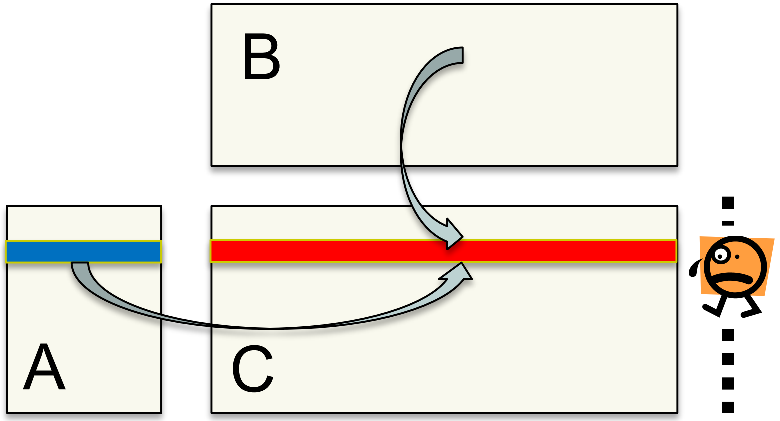
OpenMP

- 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

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

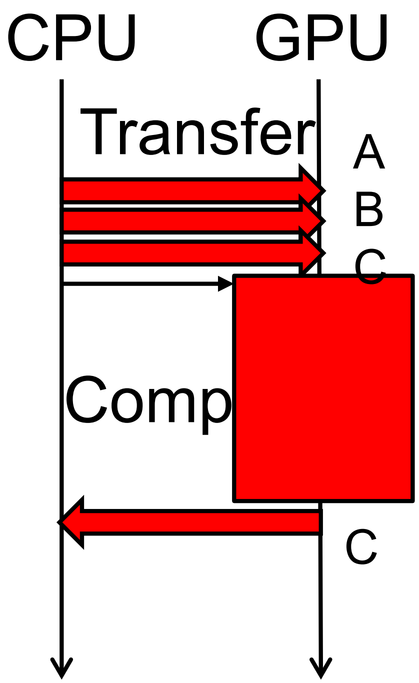
gridDim → m/BS; blockDim (BS=16 in this sample) → BS

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

Note1: <<<m, 1>>> also works, but speed is not good
<<<1, m>>> causes an error if m>1024 (CUDA's rule)

Note2: To support the case m is indivisible by BS, we should use
<<<(m+BS-1)/BS, BS>>>
→But # of threads may be larger m. "Extra" threads (id≧m) should not work. See mm-v1-cuda/mm.c

Data Transfer in mm-v1-cuda



- A, B, C are copied from CPU to GPU before computation
  - cudaMemcpy(DA, A, …) …
- C is copied from GPU to CPU after computation
  - 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

Speed of mm-v1-cuda

•Measured with a P100 GPU on TSUBAME3

m=n=k	mm-acc	mm-v1-cuda
1000	143(Gflops)	14(Gflops)
2000	173	27
4000	164	50
6000	138	70
8000	137	85

- The program outputs 2 speeds
  - Speed with data transfer costs → shown on the above table
  - Speed without data transfer costs

Discussion on Speed
(related to [G2])

- mm-v1-cuda is slower than mm-acc
  - In mm-acc, i-loop and j-loop has "loop independent"

  → m n elements are computed in parallel
- In mm-v1-cuda, we use m threads are used

→ We need more parallelism on a GPU!
  - We see 4000 or 6000 threads are still insufficient

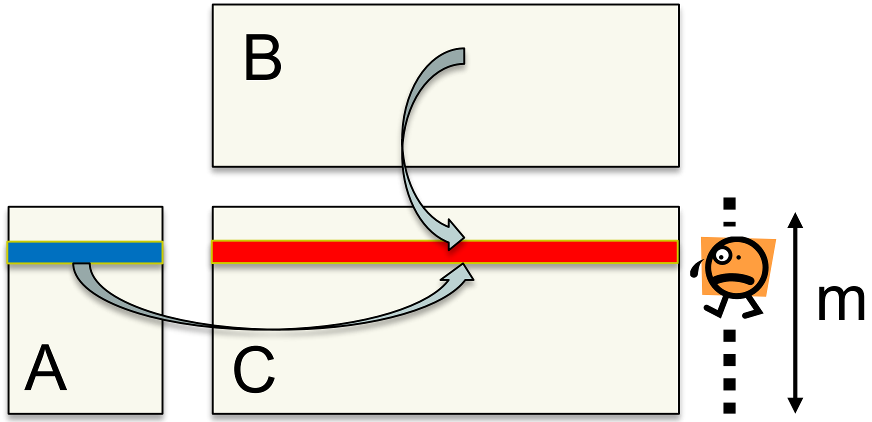
- (1thread=1row) and (1thread=1column) have different speed
  - Due to "coalesced memory access", explained in the next class

Parallelization of mm Sample (related to [G2])

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

CUDA (mm-v1-cuda)

- 1 thread computes 1 row
  - We use m threads



※ This is not the unique way

We have seen that this is slower than OpenACC version ☹ -- Why?

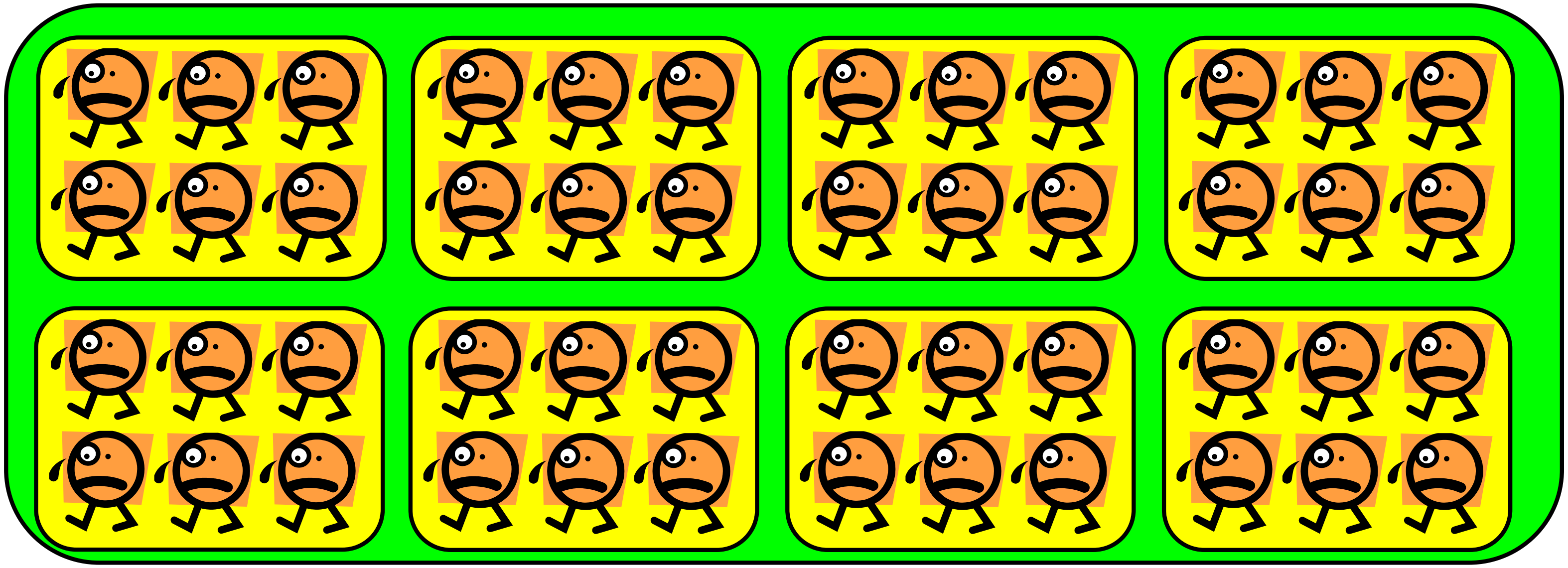
- The number of threads (m) is still insufficient on GPUs
- If (1thread = 1element), we can use m*n threads

How can we do that on CUDA?

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

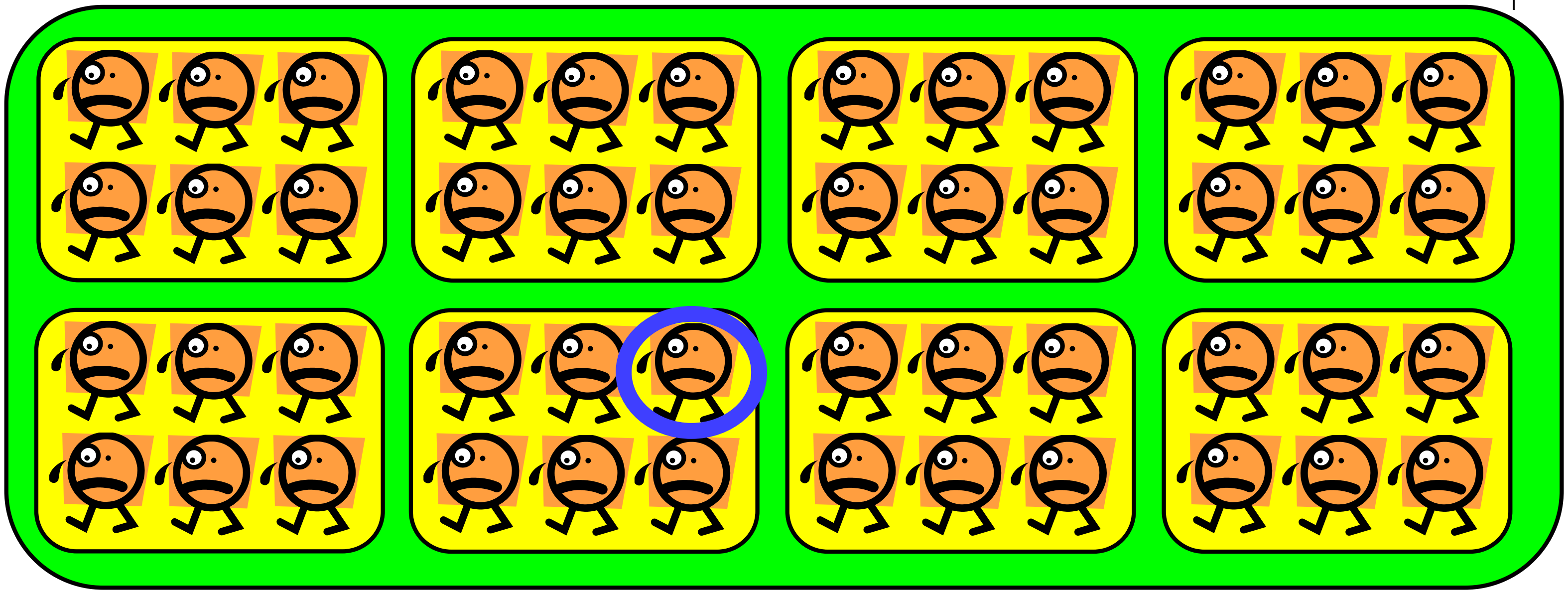
cf) func <<< dim3(4,2,1), dim3(3,2,1) >>> (); → 48 threads



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

Thread IDs in multi-dimensional cases

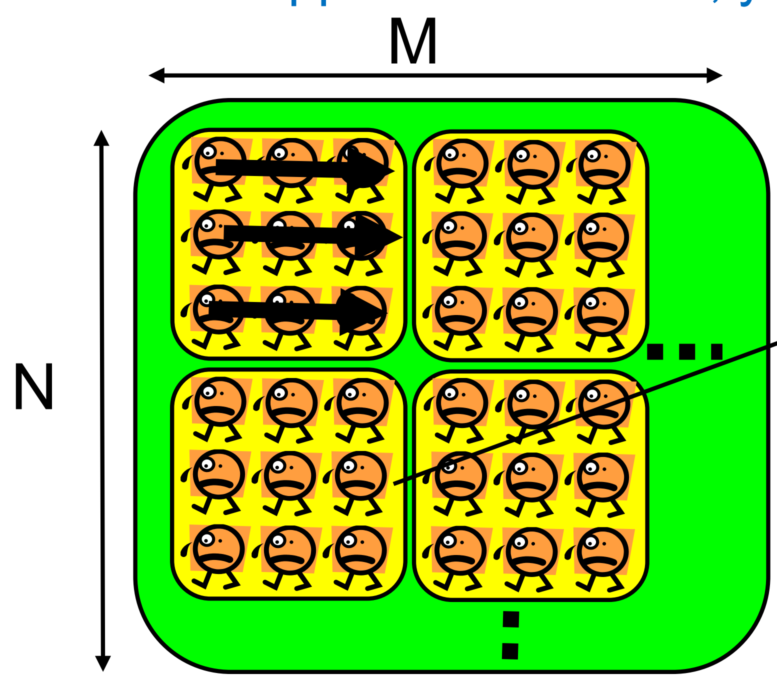
In the case of func <<< dim3(4,2,1), dim3(3,2,1) >>> ();

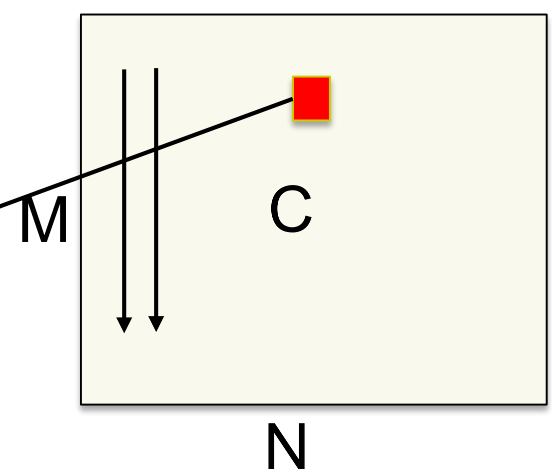


- For every thread,
  gridDim.x=4, gridDim.y=2, gridDim.z=1
  blockDim.x=3, blockDim.y=2, blockDim.z=1
- For the thread with blue mark,
  blockIdx.x=1, blockIdx.y=1, blockIdx.z=0
  threadIdx.x=2, threadIdx.y=0, threadIdx.z=0

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 use fixed blockDim (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 later)

Code in mm2-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 k loop
```

CUDA Rules on Number of Threads

func<<<gs, bs>>> (...); is interpreted as
func<<<dim3(gs,1,1), dim3(bs,1,1)>>> (...);

gridDim blockDim

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

gx: ≦ 2^{31}-1
gy, gz: ≦ 65535
bx, by: ≦ 1024
bz: ≦64

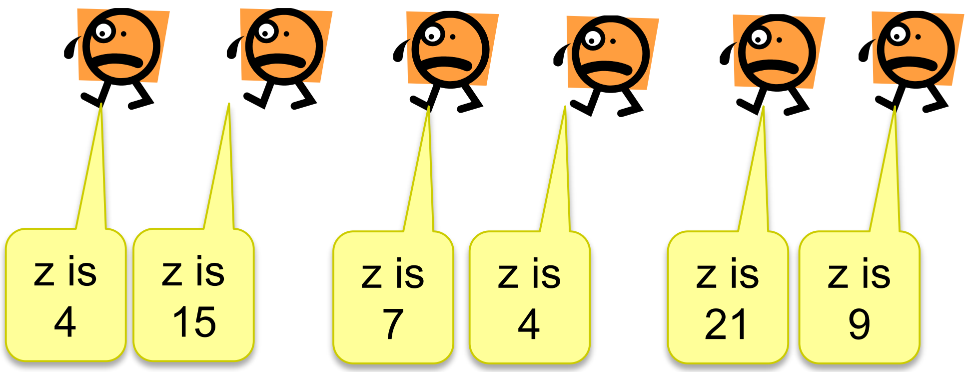
Also, bx*by*bz must be ≦1024

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"

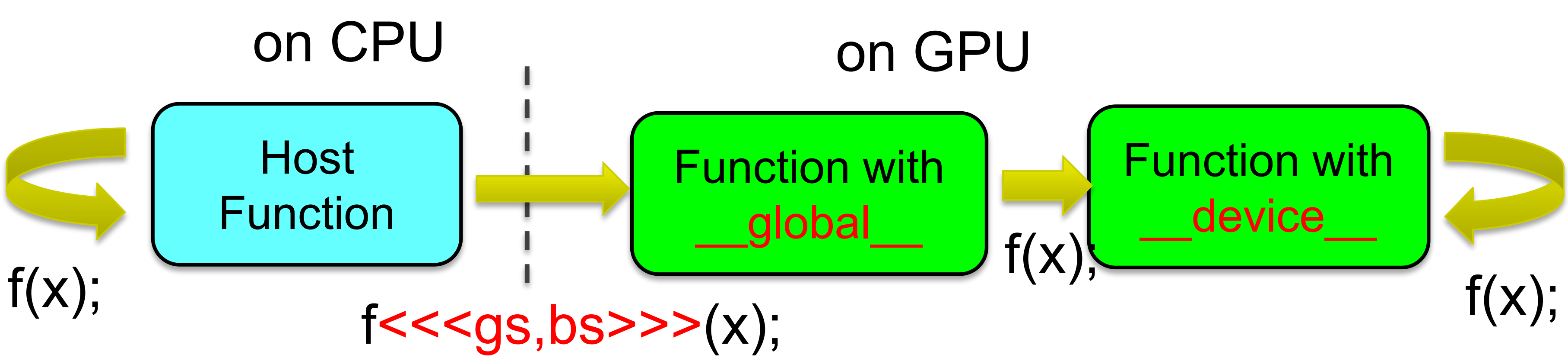


- Device memory is shared by all CUDA threads
  - Be careful to avoid race condition problem (multiple threads write same address)
  - Reading same address is ok
- Do not forget host memory and device memory are distributed

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

Assignments in GPU Part (Abstract)

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

Due date: June 18 (Thursday)

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

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

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

Next Class:

- GPU Programming (4)
  - Discussion on diffusion
  - Some techniques for speed