

2019

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

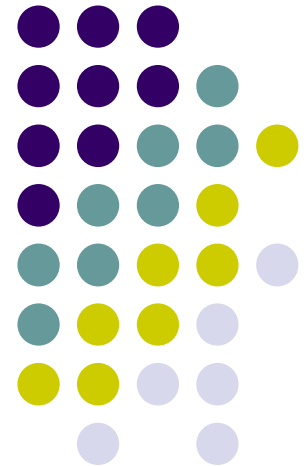
No. 13

GPU Programming (3)

Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp



Comparing OpenMP/OpenACC/CUDA



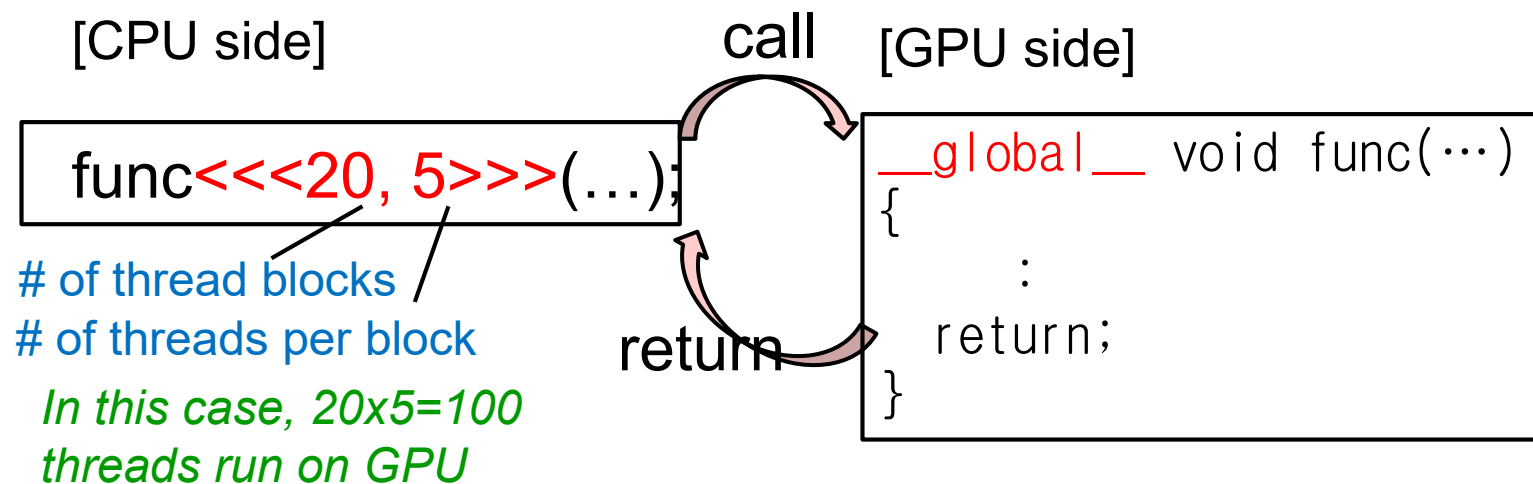
	OpenMP	OpenACC	CUDA
Processors	CPU	CPU+GPU	CPU+GPU
File extension	.c, .cc	.c, .cc	.cu
To start parallel (GPU) region	#pragma omp parallel	#pragma acc kernels	func<<<..., ...>>>()
To specify # of threads	export OMP_NUM_THREADS=...	(num_gangs, vector_length etc)	func<<<..., ...>>>()
Derisable # of threads	# 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”

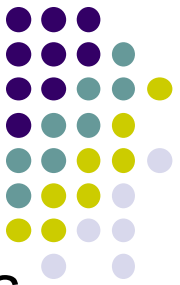
Calling A GPU Kernel Function from CPU



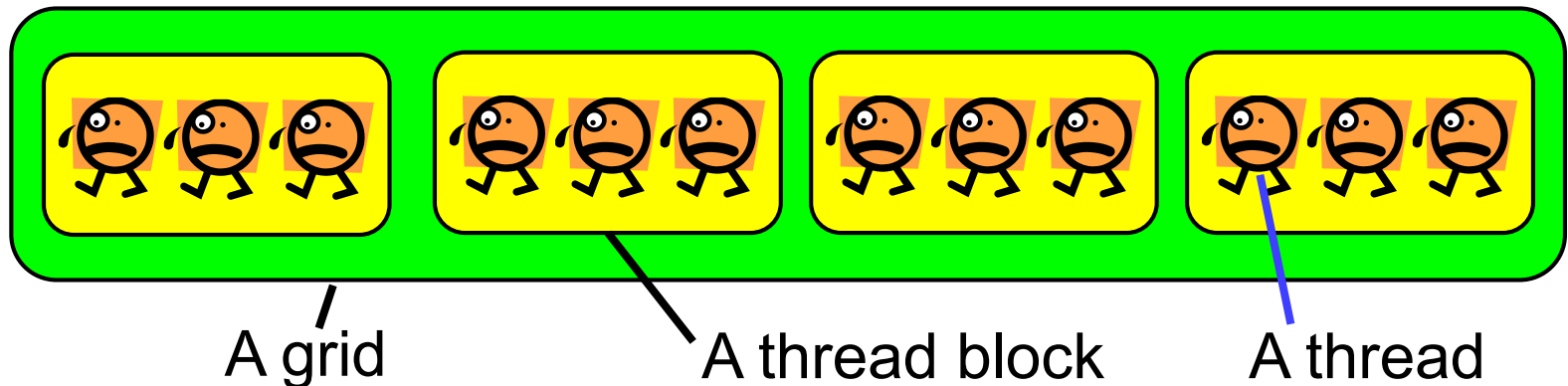
- A region executed by GPU must be a distinct function
 - called a GPU kernel function



Threads in CUDA



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 \Leftrightarrow SMX, Thread \Leftrightarrow CUDA core

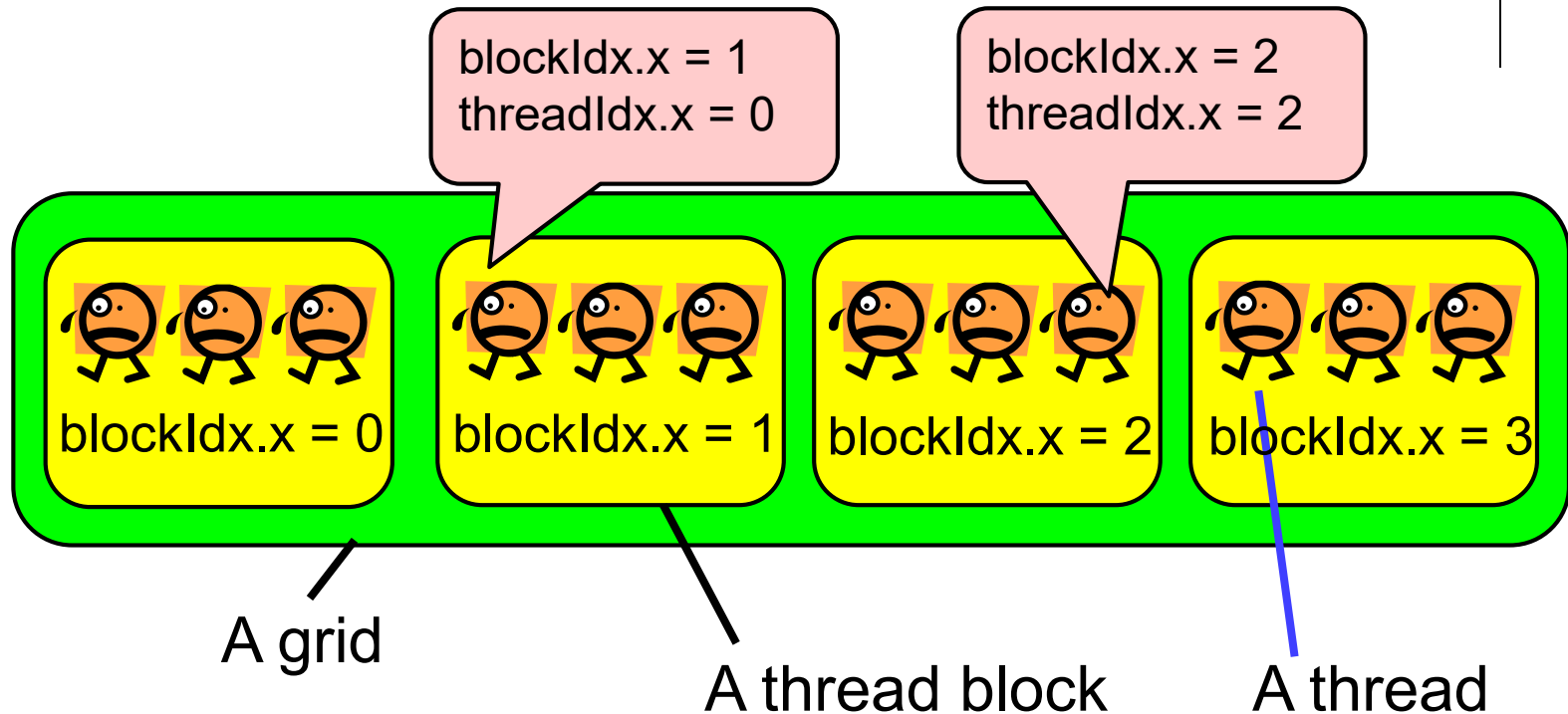


To See Who am I

- 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
 - blockDim.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?



On CUDA, Different strategy is required from on OpenMP

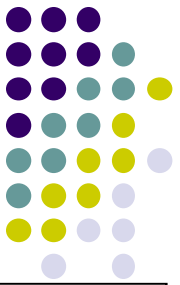
- On OpenMP, number of threads (OMP_NUM_THREADS) should be \leq CPU cores
 - ≤ 4 on q_core node, ≤ 28 on f_node
- On CUDA, it is better to use number of thread \geq GPU cores
 - ≥ 3584 on TSUBAME3's P100 GPU
 - You can use $>1,000,000$ threads!

We have to decide 2 numbers <<<block number, block size>>>

- (1) We decide **total** number of threads P
- (2) We tune each block size BS
 - Good candidates are 16, 32, 64, ... 1024
- (3) Block number is P/BS
 - We consider indivisible cases later



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



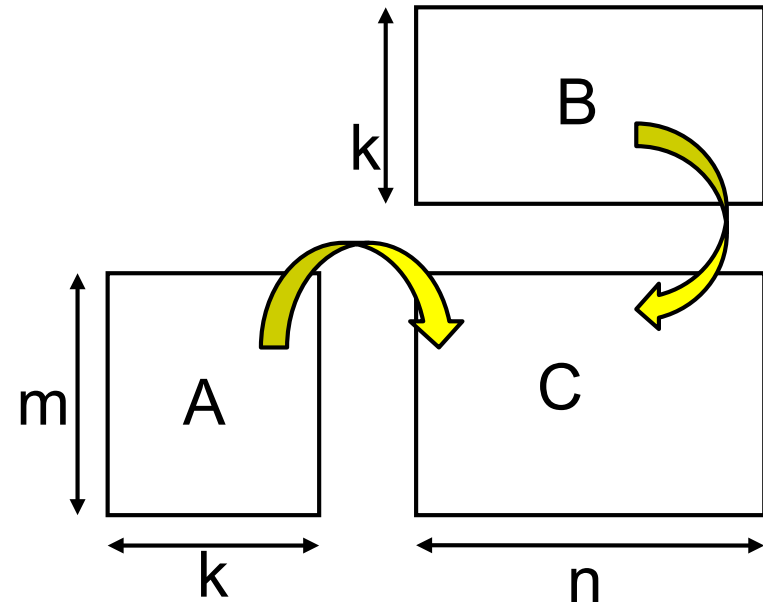
CUDA version available at [~endo-t-ac/ppcomp/18/mm1-cuda/](https://endo-t-ac/ppcomp/18/mm1-cuda/)

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

C: a $(m \times n)$ 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



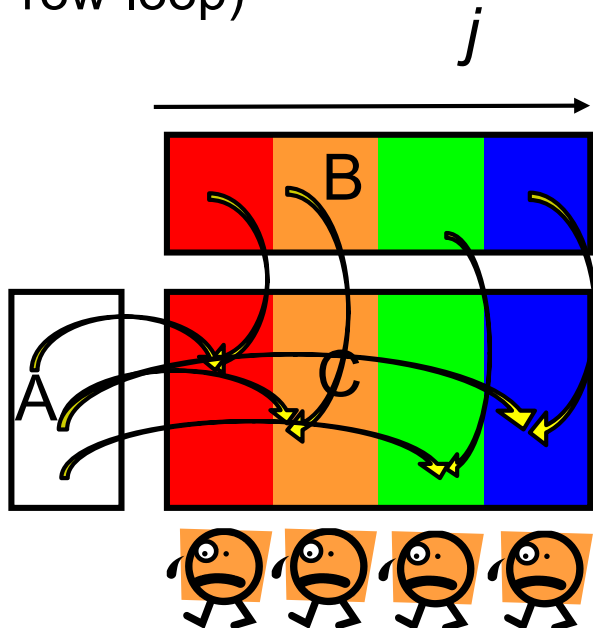
How We Parallelize Computation

In mm, we can compute different C elements in parallel

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

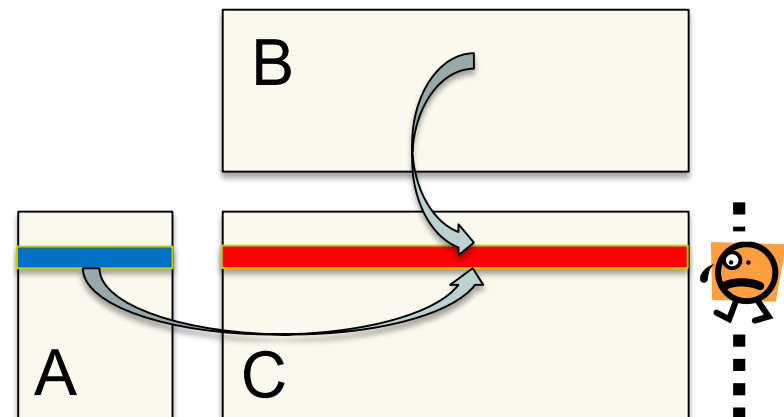
OpenMP

- Parallelize column-loop
(or row-loop)



CUDA (mm1-cuda)

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



✂ This is not the unique way



Parallelism in mm1-cuda

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

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

gridDim

blockDim (BS=16 in this sample)

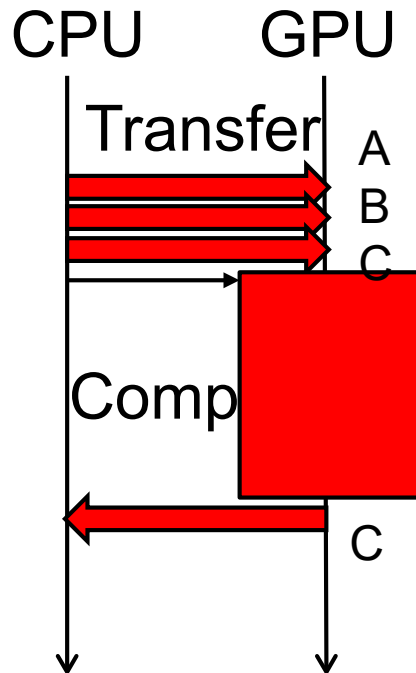
1 element for 1 row \rightarrow No need of “i” loop in this sample

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>>>`
 \rightarrow But # of threads may be larger m . “Extra” threads ($id \geq m$) should not work. See mm1-cuda.c/mm.c



Data Transfer in mm1-cuda



- A, B, C are copied from CPU to GPU before computation
 - `cudaMemcpy(... cudaMemcpyHostToDevice)`
- C is copied from GPU to CPU after computation
 - `cudaMemcpy(... cudaMemcpyDeviceToHost)`



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`



Comparing speed of mm1-cuda

m=n=k	mm-acc	mm1-cuda (thread=row)	mm1-cuda (thread=column)
1000	143(Gflops)	14(Gflops)	14(Gflops)
2000	173	27	24
4000	164	50	29
6000	138	70	31
8000	137	85	32

- Measured with a P100 GPU on TSUBAME3
- CUDA version is compiled with `-arch=sm_60` option, for better speed (see mm1-cuda/Makefile)
- Data transfer costs are included

Discussion on Speed (related to [G2])

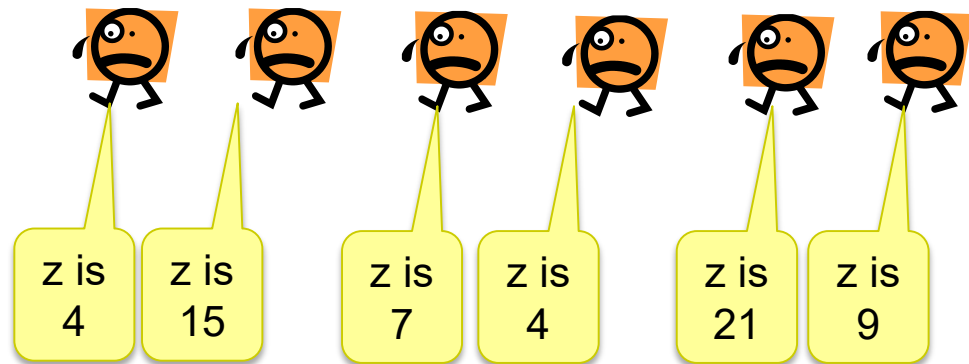


- mm1-cuda is slower than mm-acc
 - In mm-acc, i-loop and j-loop has “loop independent”
→ $m \times n$ elements are computed in parallel
- In mm1-cuda, we use m (or n) threads are used
→ We need more parallelism on a GPU!
 - We see 4000 or 6000 threads are still insufficient
 - Will be improved in the next class
- (thread=row) and (thread=column) have different speed
 - Due to “coalesced memory access”, explained in the next class



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

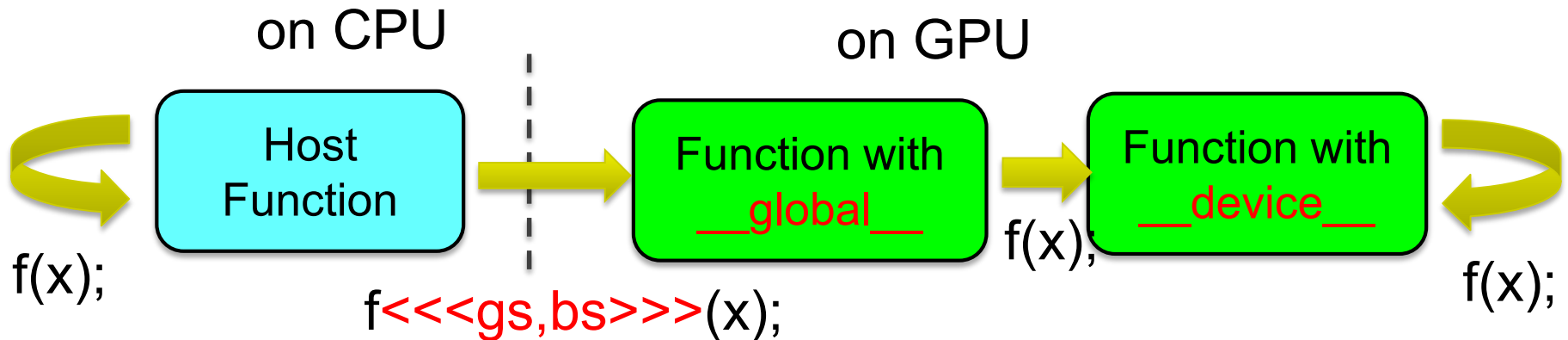
1) Functions with `__global__` keyword

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

2) Function with `__device__` keyword

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

→ In OpenACC, `#pragma acc routine`

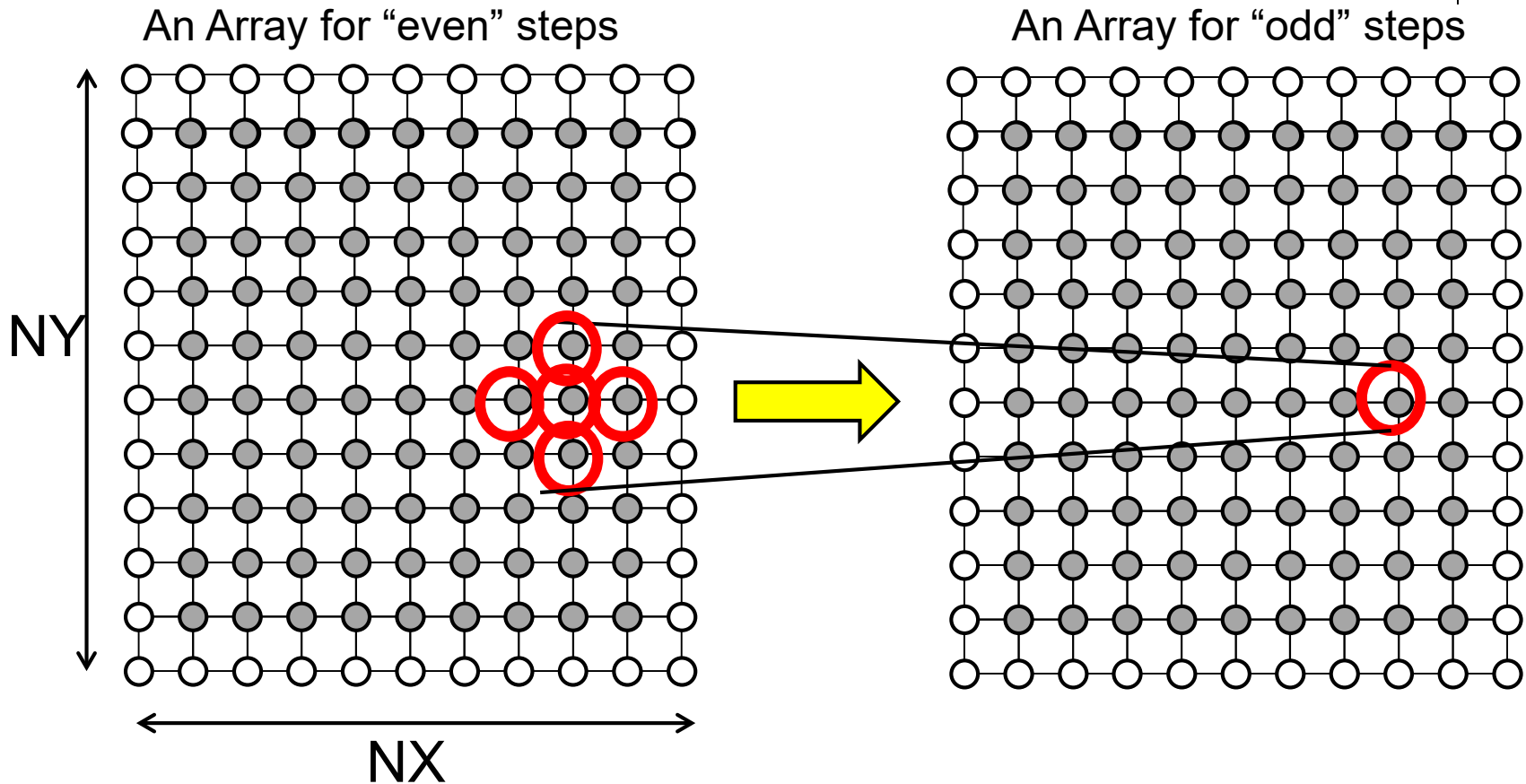


What Can be Done in GPU Functions?



- 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

Discussion on diffusion sample (related to [G1])



CUDA Parallelization of diffusion



- t-loop **cannot** be parallelized (as usual)
- Computation of one time step should be a GPU kernel
- How do we design threads on CUDA?
 - 1thread = 1row
 - We use NY threads in total → only x-loop in the kernel
 - 1thread = 1column
 - We use NX threads in total → only y-loop in the kernel
 - 1thread = 1element (optional in [G1])
 - We use NX NY threads in total → No loop in kernel !
 - Discussed in next class

Assignments in GPU Part (Abstract)



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

Due date: June 17 (Monday)

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

[G2] Evaluate speed of “mm-acc” or “mm-cuda” ([mm1-cuda](#) and [mm2-cuda](#)) in detail

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



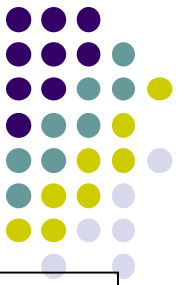
Notes in Submission

- 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:

- June 3: TSUBAME3.0 tour
 - Please come to this room (W242)
 - Then a staff will bring you to GSIC building
- June 6: GPU Programming (4)



Information

Lecture

- Slides are uploaded in OCW
 - www.ocw.titech.ac.jp → search “2019 practical parallel computing”
- Assignments information/submission site are in OCW-i
 - Login portal.titech.ac.jp → OCW/OCW-i
- Inquiry
 - ppcomp@el.gsic.titech.ac.jp
- Sample programs
 - Login TSUBAME, and see `~endo-t-ac/ppcomp/19/` directory

TSUBAME

- Official web including Users guide
 - www.t3.gsic.titech.ac.jp
- Your account information
 - Login portal.titech.ac.jp → TSUBAME portal