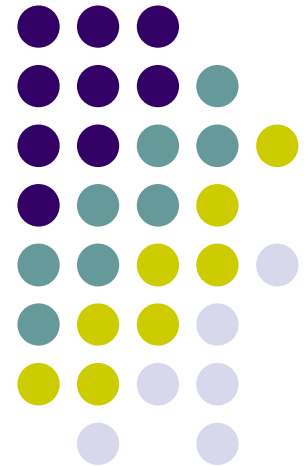


# Practical Parallel Computing (実践的並列コンピューティング)

2025 Class No.9  
[CUDA Part] (1)  
Introduction to CUDA

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





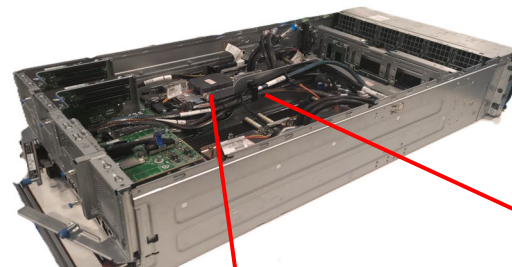
# Overview of This Course

- Introduction Part
    - 2 classes
  - OpenMP (OMP) Part
    - 4 classes
    - Report (required)
  - OpenACC (ACC) Part
    - 2 classes
    - Report (required)
  - CUDA Part
    - 3 classes
    - Report (elective)
  - MPI Part
    - 3 classes
    - Report (elective)
- ← We are here (1/3)



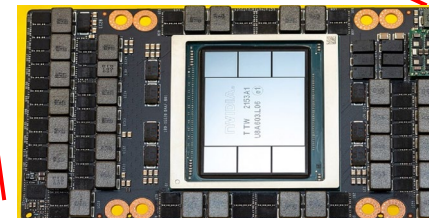
# OpenACC and CUDA for GPUs

- **OpenACC**
    - C/Fortran + directives (`#pragma acc ...`), Easier programming
    - Basically for data parallel programs with for-loops
  - **CUDA**
    - Most popular and suitable for higher performance
- Programming is harder, but more general



TSUBAME4  
node

GPU

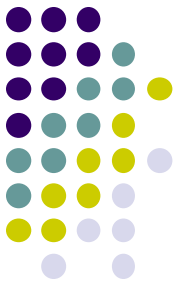




# Why is CUDA Harder?

- Like **OpenACC**,
  - We program code for GPUs and CPUs
  - We program data copying
- Unlike **OpenACC**,
  - We must specify number of threads (& thread blocks)
  - No “**#pragma acc kernel**” → we write **kernel functions**
  - No “**#pragma acc loop**” → we write code for “what each thread does”
  - No “**#pragma acc data**” → we must use different arrays/pointers on CPUs and GPUs

:



# An OpenACC Program 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



# A CUDA Program Look Like

Sample:

[ppcomp-ex/cuda/add/](#)

```
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*)

We have to separate code regions executed on CPU and GPU



# Using cuda/add Sample

```
[make sure that you are at a interactive node (rXn11) ]  
module load nvhpc [Do once after login]  
[please go to your ppcomp-ex directory]  
cd cuda/add  
make  
[An executable file "add" is created]  
./add
```

Meaning of the program is very simple:

```
for (i = 0; i < 100; i++) { A[i] += B[i]; }
```



# Notes on “module load”

- Either is ok for CUDA programming
  - `module load nvhpc` → Both OpenACC and CUDA (12.6) are ok
  - `module load cuda` → CUDA (12.8) is ok

- If “module load cuda” fails:

```
Loading cuda/12.8.0
ERROR: Module cannot be loaded due to a conflict.
HINT: Might try "module unload nvhpc/25.1 cuda12.6" first.
```

→ On TSUBAME4, “nvhpc” and “cuda” cannot be used together

- Try “`module unload nvhpc`” and then “module load cuda”
- Another method: “`module purge`” unload all modules (in the same shell)

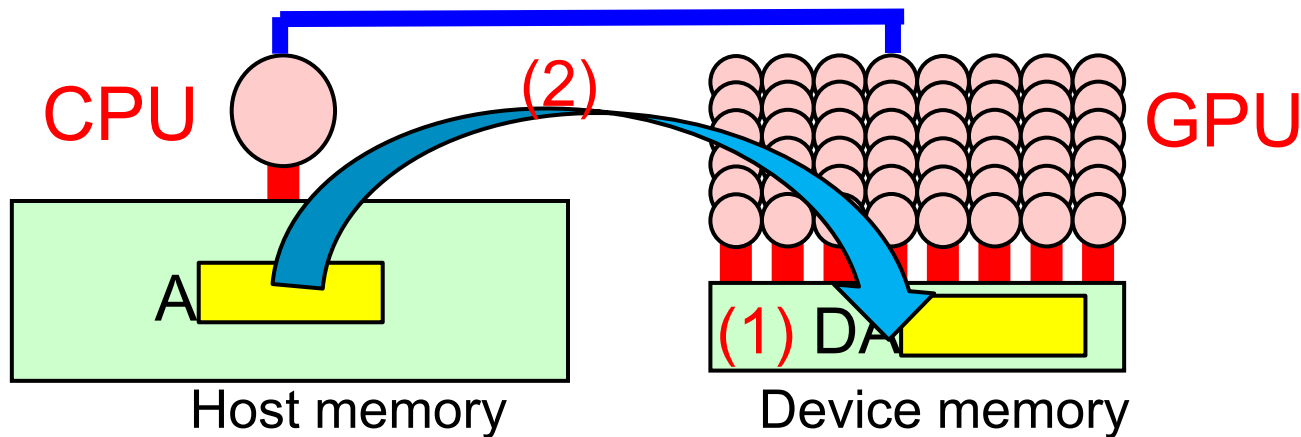


# Preparing Data on Device Memory



Before computation on GPU, we need to prepare data on device memory

- Allocate a region on device memory (1)  
cf) `cudaMalloc((void**)&DA, size);` → DA is a pointer on device memory
- Copy data from host to device (2)  
cf) `cudaMemcpy(DA, A, size, cudaMemcpyDefault);`



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



# In CUDA, No “acc data”

## OpenACC

Both allocation and copy are done by **acc 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]; ← on CPU
int *DA; ← on GPU
cudaMalloc(&DA, ...);
cudaMemcpy(DA, A, ..., ...);
// Here CPU cannot access DA[i]

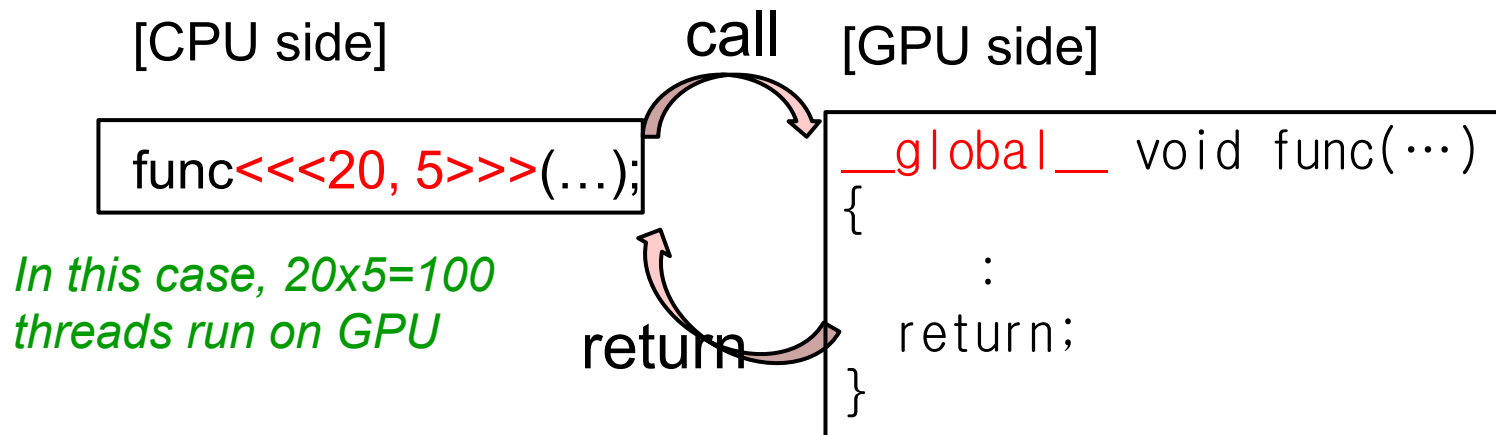
func<<<..., ...>>>(DA, ...);
```

# Calling A GPU Kernel Function from CPU



- We need to write functions on GPU and functions on CPU separately

- A functions on GPU is called a GPU kernel function

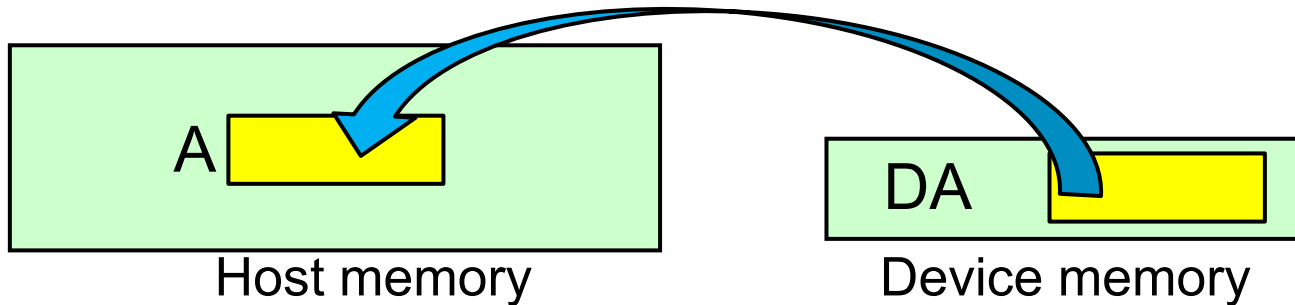


A GPU kernel function (called from CPU)

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

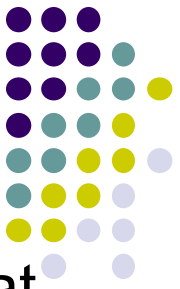


# Copying Back Data from GPU

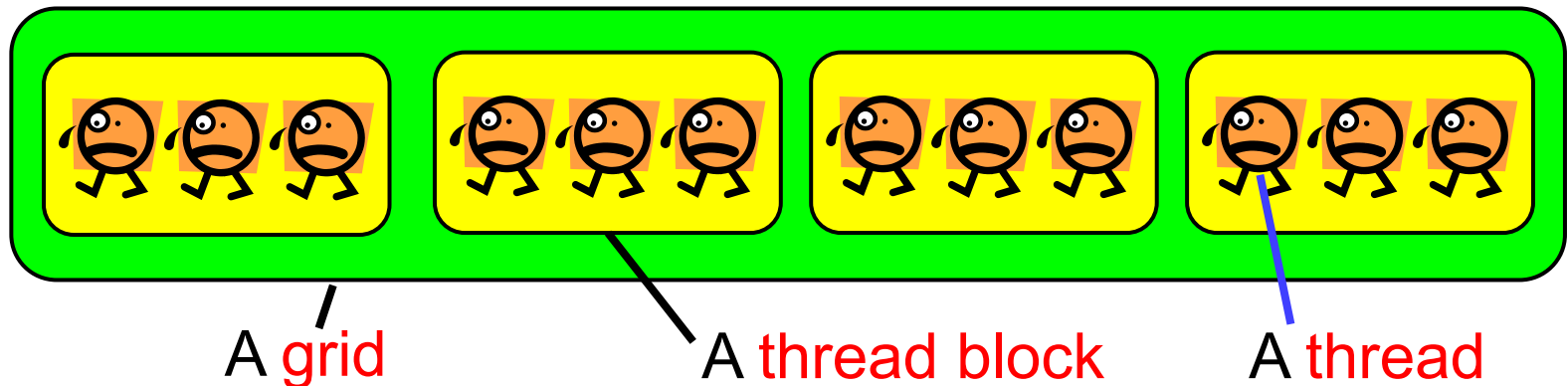


- Copy data using `cudaMemcpy`
  - cf) `cudaMemcpy(A, DA, size, cudaMemcpyDefault);`
  - 4<sup>th</sup> argument is one of
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost`
    - `cudaMemcpyDefault` ← Detect memory type automatically 😊
- When a memory area is unnecessary, free it
  - cf) `cudaFree(DA);`

# Threads in CUDA



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



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

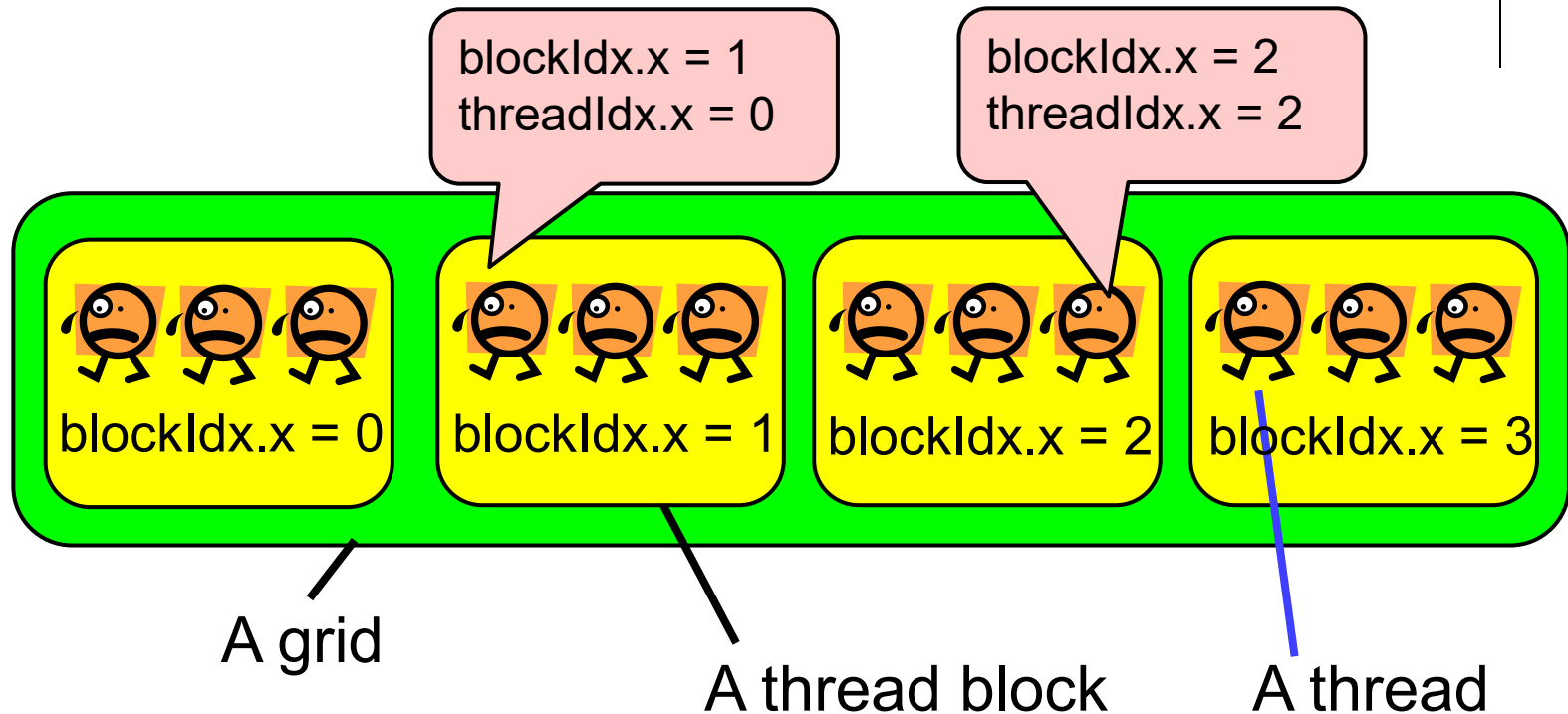


# To See Who am I

- By reading the following special variables, each thread can see its thread ID in GPU kernel function
- My ID
  - blockIdx.x: Index of the block the thread belong to ( $\geq 0$ )
  - threadIdx.x: Index of the thread (**inside the block**) ( $\geq 0$ )
- Number of thread/blocks
  - 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`**

# The Case of cuda/add Sample



- [ppcomp-ex/cuda/add](#)
- We want to do

```
for (i = 0; i < 100; i++) { DA[i] += DB[i]; }
```

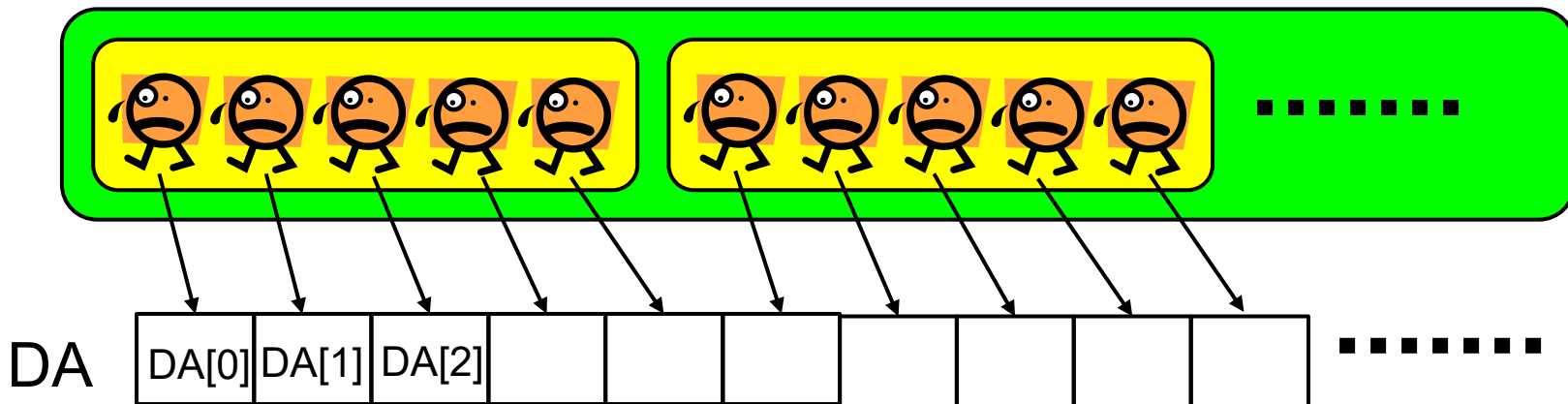
[CPU side]

```
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;
}
```





# “mm” sample: Matrix Multiply (related to [C3])



CUDA versions are at

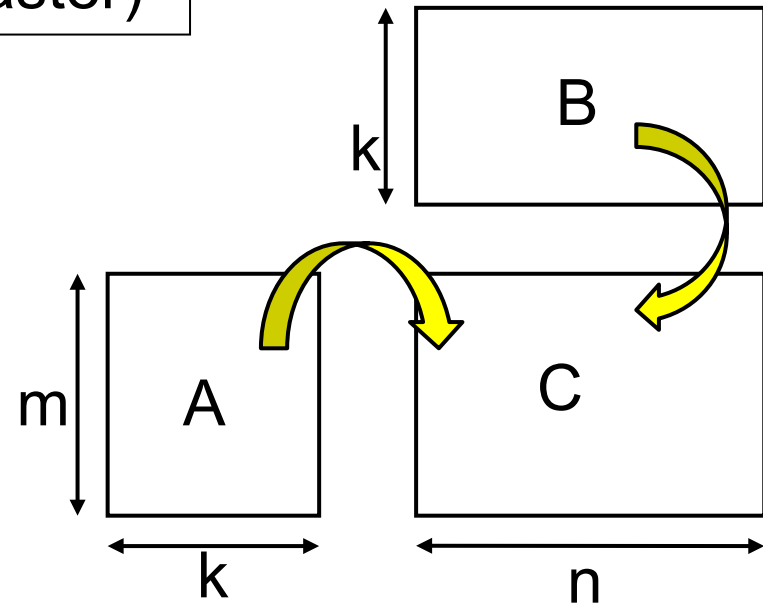
- [ppcomp-ex/cuda/mm-1dpar/](#) (slow)
- [ppcomp-ex/cuda/mm/](#) (faster)

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
- Execution: `./mm [m] [n] [k]`





# Using cuda/mm-1dpar Sample

*[make sure that you are at a interactive node (rXn11) ]*

**module load nvhpc** *[Do once after login]*

*[please go to your ppcomp-ex directory]*

**cd cuda/mm-1dpar**

**make**

*[An executable file “mm” is created]*

**./mm 2000 2000 2000**

When you try cuda/mm,  
replace **cuda/mm-1dpar** to **cuda/mm**



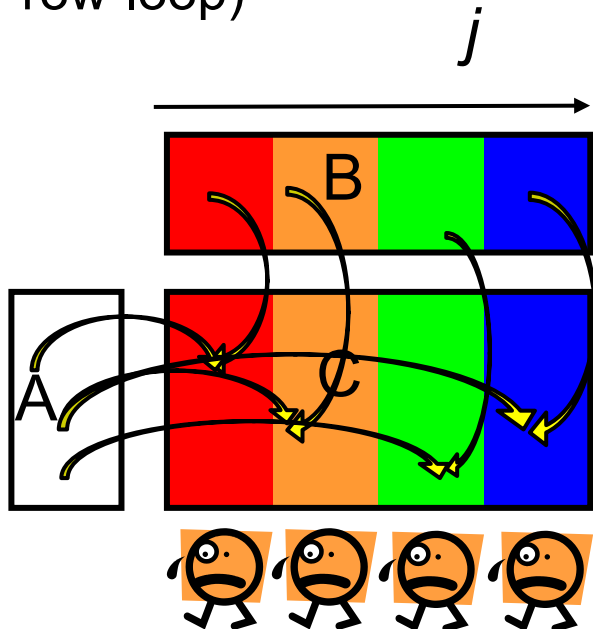
# How We Parallelize Computation

In mm, we can compute different C elements in parallel

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

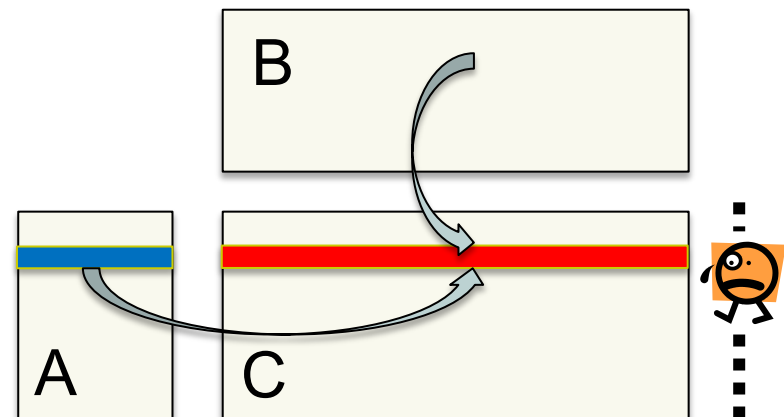
## OpenMP

- Parallelize column-loop  
(or row-loop)



## CUDA (mm-1dpar)

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



✘ This is not the unique way



# Parallelism in cuda/mm-1dpar

- 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=64 in  
this sample)

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

Note:

`<<<m, 1>>>` also works, but speed is even slower ☹

`<<<1, m>>>` causes an error if  $m > 1024$  (CUDA's rule)

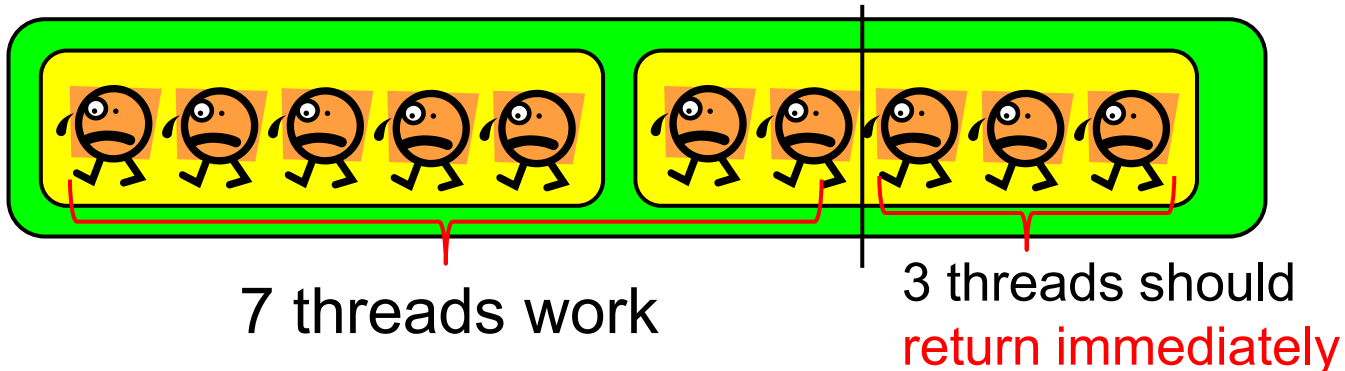
# If Number of Threads is Indivisible by BlockDim



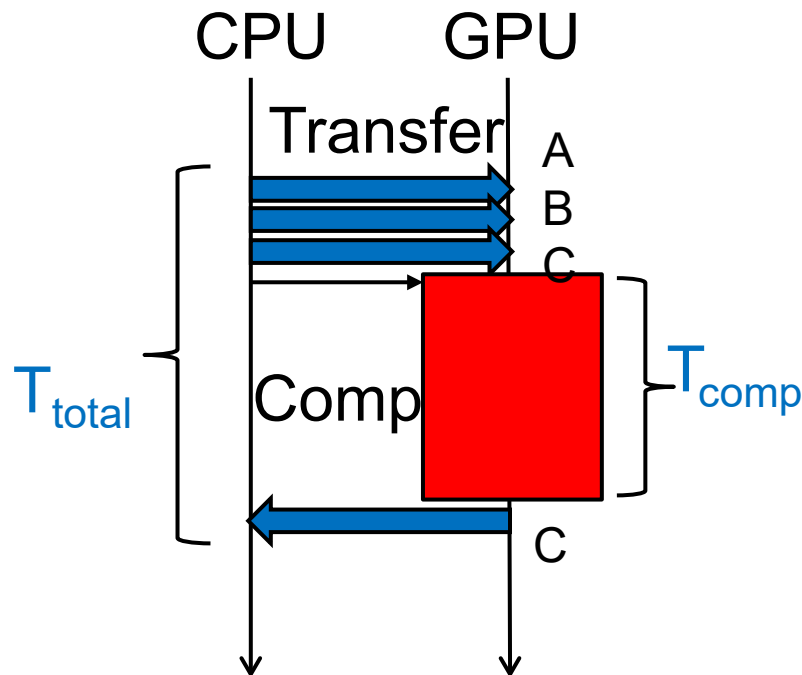
- $m$ : the number of threads
- $BS$ : blockDim
- If  $m$  may be indivisible by  $BS$ , we should use  $\lll(m+BS-1)/BS, BS\ggg$
- But # of threads may be larger  $m$ . “Extra” threads ( $id \geq m$ ) should not work. See [cuda/mm-1dpar/mm.cu](http://cuda-mm-1dpar/mm.cu)

Example:  $m=7$ ,  $BS=5 \rightarrow \lll 2, 5 \ggg$

10 threads start working, but 3 threads should do nothing



# Data Transfer in cuda/mm-1dpar



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

- `cudaMemcpy(DA, A, ...)`
- `cudaMemcpy(DB, B, ...)`
- `cudaMemcpy(DC, C, ...)`

(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

Bad news: cuda/mm-1dpar is much slower than OpenACC! ☹️  
(even slower than OpenMP?)

- In acc/mm, i-loop and j-loop has “loop independent”  
→  $m \times n$  elements are computed in parallel
  - In mm-1dpar, we use  $m$  threads  
→ We should use more threads on a GPU!
    - We see  $m=1000 \sim 8000$  threads are still insufficient, and slow
- cuda/mm uses  $m \times n$  threads. Explained in next class



# How is Number of Threads Determined? (1)



## Difference between OpenMP and CUDA

- On OpenMP, number of threads (OMP\_NUM\_THREADS) should be  $\leq$  CPU cores (or hyper threads)
  - The number is basically determined by hardware
  - $\leq 48$  on an interactive (node\_o) node,  $\leq 384$  on node\_f
- On CUDA, it is better to use number of thread  $\gg$  GPU cores
  - $\gg 7,680$  on an interactive (node\_o) node with  $\frac{1}{2}$  GPU
  - $\gg 16,896$  on gpu\_1, node\_q ...
  - You can use  $>1,000,000$  threads!

# How is Number of Threads Determined? (2)



We have to decide 2 numbers in kernel call  
<<<number of blocks, 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 32, 64, 128, ... 1024
- (3) Then block number is  $P/BS$ 
  - For indivisible cases,  $(P+BS-1)/P$



# Comparing OpenMP/OpenACC/CUDA



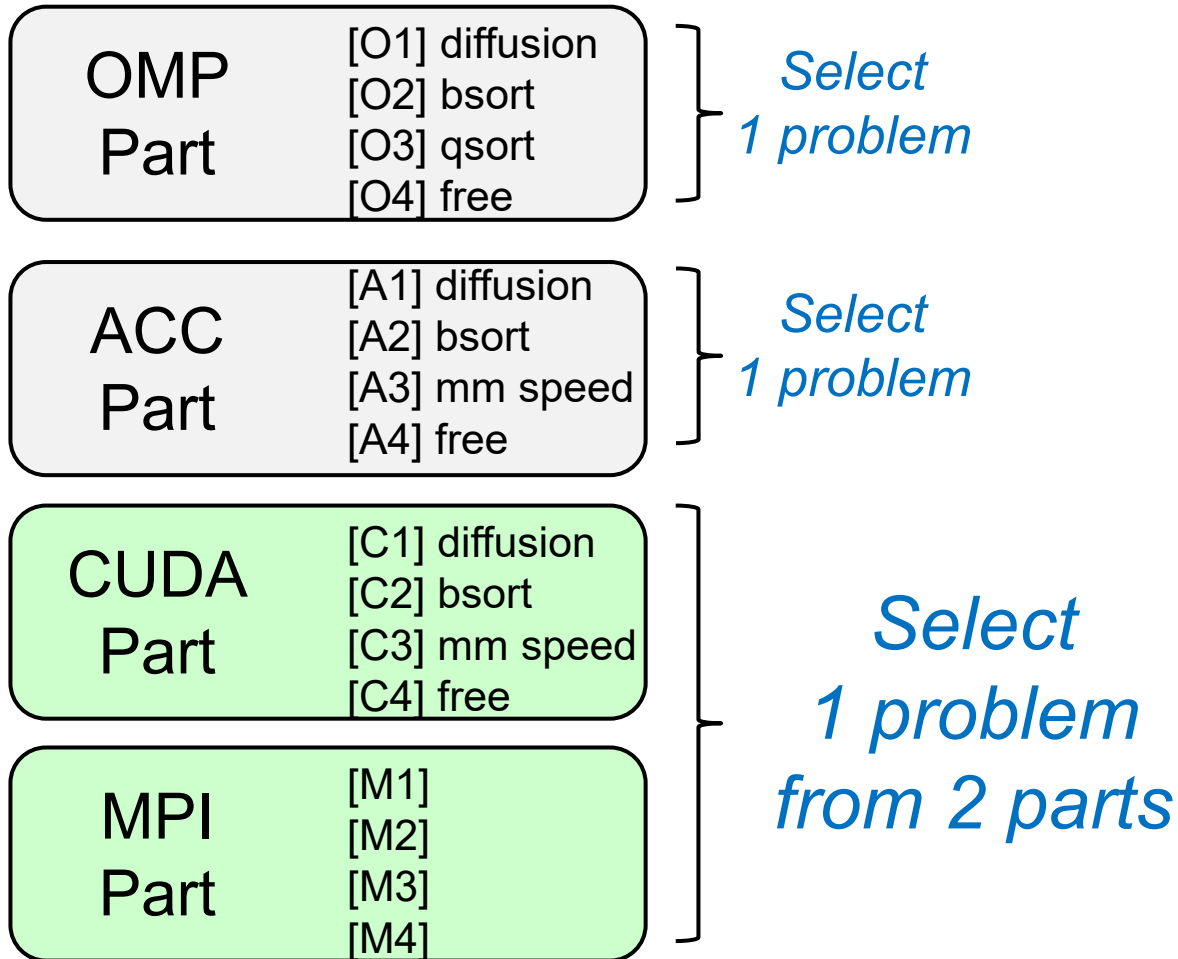
	OpenMP	OpenACC	CUDA
Processors	CPU	CPU+GPU	
File extension	.c, .cc		.cu
To start parallel (GPU) region	#pragma omp parallel	#pragma acc kernels	func<<<..., ...>>>()
To specify # of threads	export OMP_NUM_THREADS=...	(num_gangs, vector_length etc)	
Desirable # of threads	# of CPU cores or less	# of GPU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

# Assignments in this Course



- There is homework for each part.





# Assignments in CUDA Part (1)

If you choose this part,  
choose one of [C1]—[C4], and submit a report  
Due date: May 26 (Monday)

## [C1] Parallelize “diffusion” sample program by CUDA

- You can start from [/ppcomp-ex/cuda/diffusion](http://ppcomp-ex/cuda/diffusion)

Optional:

- To make array sizes variable parameters
- To compare CUDA vs OpenMP (vs OpenACC?)
- To improve performance further



## Assignments in CUDA Part (2)

**[C2]** Parallelize “bsort” sample program by CUDA

- You can start from </ppcomp-ex/cuda/bsort>

Optional:

- Comparison with other sort algorithms
  - Quick sort (qsort), Heap sort, Merge sort, ...
- Comparison with OpenMP or OpenACC...



# Assignments in CUDA Part (3)

**[C3]** Evaluate speed of “cuda/mm” sample in detail  
[ppcomp-ex/cuda/mm/](#)

- Compare speed of cuda/mm and cuda/mm-1dpar
- Use various matrices sizes
- Evaluate effects of data transfer cost

Optional :

- Compare with OpenMP or OpenACC
- To use different loop orders
- To change/improve the program
  - Cache blocking?



# Assignments in CUDA Part (4)

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

- cf) A problem related to your research
- Challenging one for parallelization is better
  - cf) Partial computations have dependency with each other



# Notes in Report Submission (1)



- Submit the followings via **LMS**
  - (1) **A report document**
    - PDF, MS-Word or text file
    - 2 pages or more
    - in English or Japanese (日本語もok)
  - (2) **Source code files** of your program
    - Try “zip” to submit multiple files

# Notes in Report Submission (2)



The report document should include:

- Which problem you have chosen
- How you parallelized
  - It is even better if you mention efforts for high performance or new functions
  - [CUDA] How many threads? What computations do threads do?
- Performance evaluation on TSUBAME
  - With varying number of threads
  - With varying problem sizes
  - Discussion with your findings
  - Other machines than TSUBAME are ok, if available



# Plan of CUDA Part

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