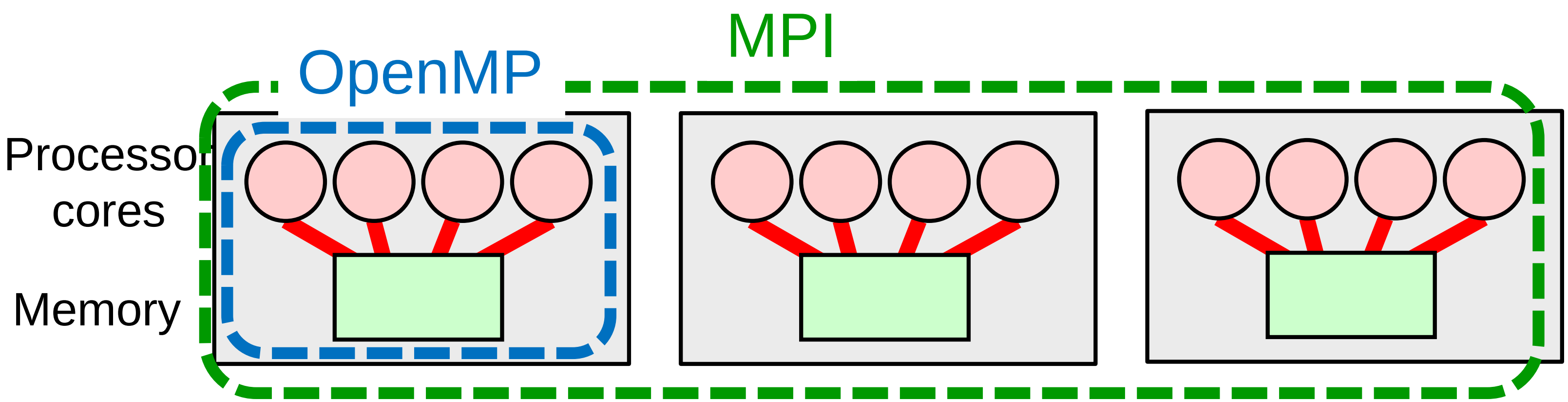
2017
Practical Parallel Computing
(実践的並列コンピューティング)
No. 12

GPU Programming with CUDA (1)

Toshio Endo
School of Computing & GSIC
endo@is.titech.ac.jp

Parallel Programming using CPUs

- Both OpenMP and MPI uses multiple processor cores in CPUs
  - OpenMP: cores in a single node
  - MPI: we can use cores in multiple nodes



In Part 3, we use other processors than CPUs → GPU

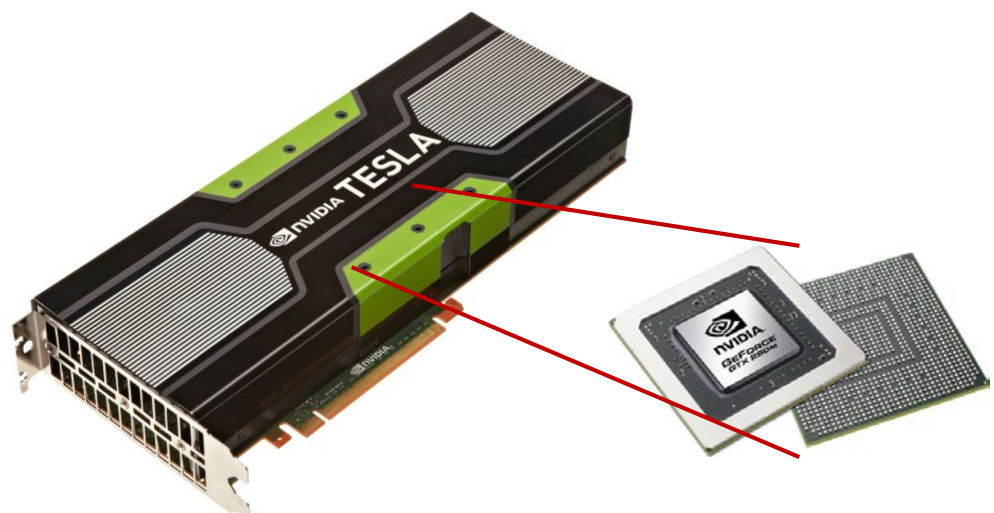
GPU Computing

- Graphic processing units (GPU) have been originally used for computing graphics (including video games)
- A GPU has many (simple) cores
  - CPU: 4 to 32 cores. GPU: >100 cores

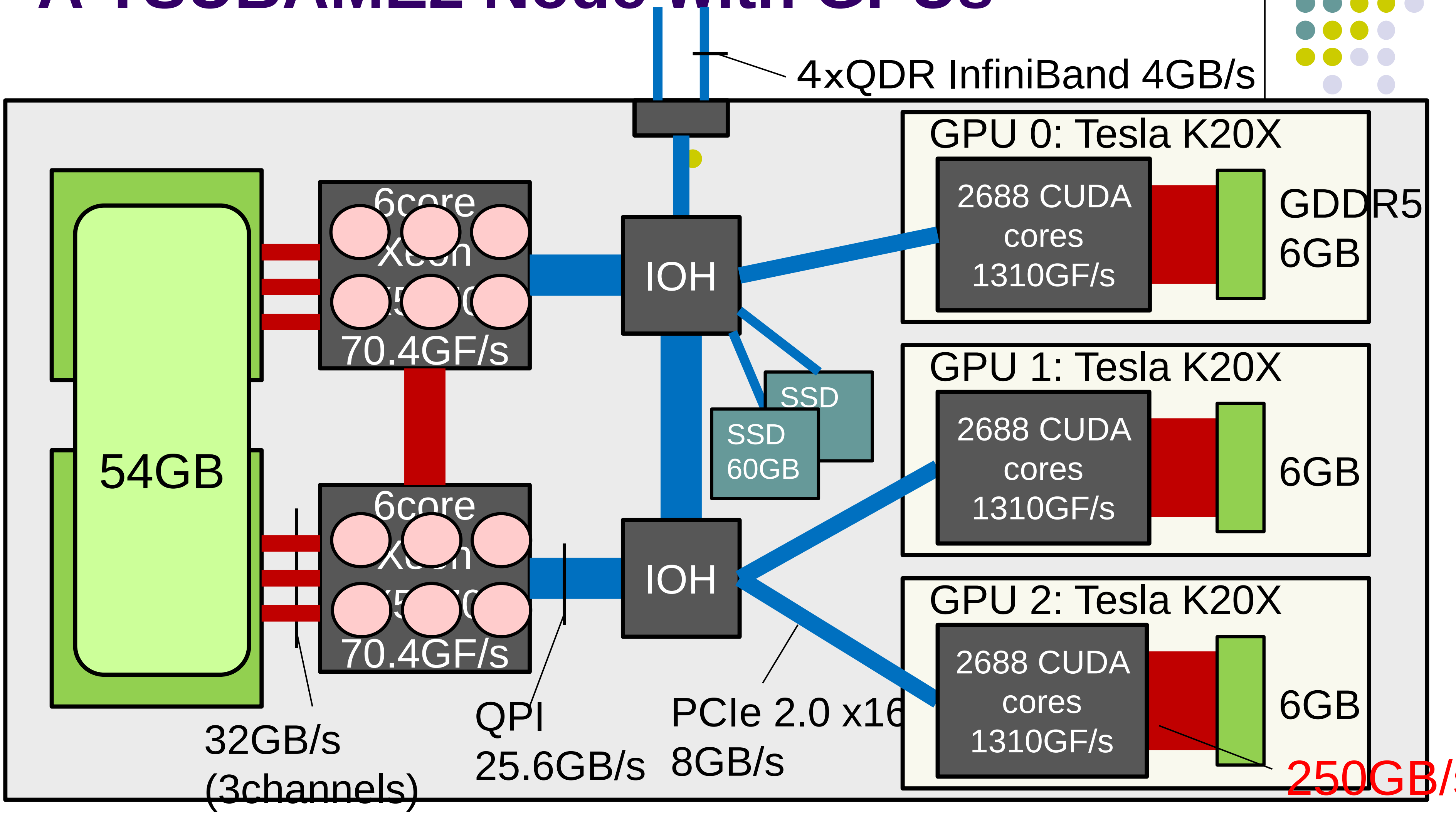
→ Recent GPUs can be used for general applications!

  - The concept is called GPGPU (General-Purpose computing on GPU)

- Became popular since NVIDIA corp. invented CUDA language in 2007



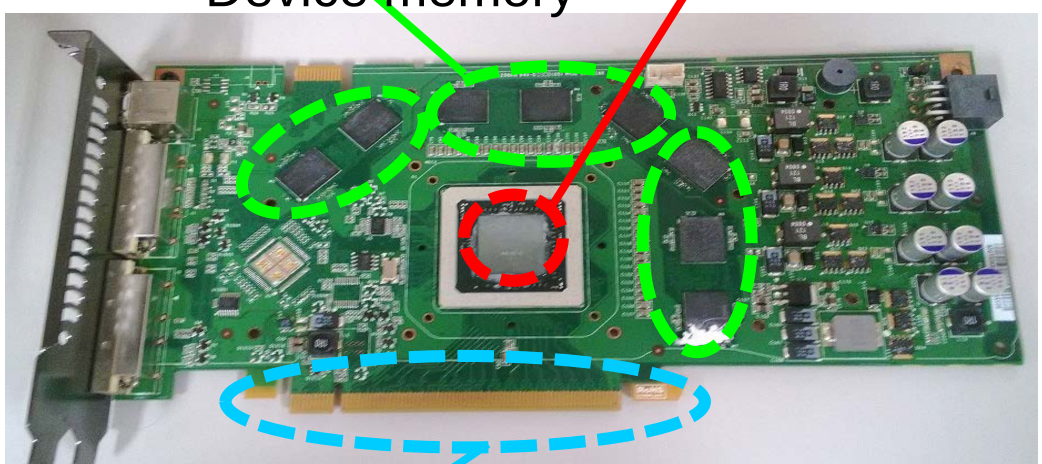
A TSUBAME2 Node with GPUs



Characteristics of GPUs

- A GPU is a board or a card attached to computers
  - ⇒It cannot work alone. Driven by CPUs
- A GPU has many cores (called <u>CUDA cores</u>)
  - ⇒K20X (TSUBAME's GPU) has 2688 CUDA cores (=192 x 14SMs)
- A GPU has dedicated memory (called <u>device memory</u>), which is different from CPU's memory
  - ⇒K20X has 6GB memory

Device memory GPU processor



PCI-Express connector (connected with motherboard)

CUDA Programming Language

- A programming language for NVIDIA GPUs
- Extensions to C/C++/Fortran
- Compile with nvcc command
  - File extension is .cu

Official documents: http://docs.nvidia.com/cuda/
“Programming Guide” is important

※ CUDA does not work computers without NVIDIA GPUs
  - AMD GPUs, Intel GPUs, no GPU machine…
- OpenCL can work on such machines, but harder to program
- Recently OpenACC is becoming popular
  - Directive-based (#pragma) GPU programming

inc-seq: First Sample of CUDA

- Available at ~endo-t-ac/ppcomp/17/inc-seq/
- It creates an integer array. The array elements are incremented on GPU
- Compile and execute

```
$ nvcc inc-seq.cu –o inc-seq
$ ./inc-seq
```

※ nvcc also takes optimization flags such as “-O”

Submission of GPU Jobs

(1) Make a script file (For example, the name is job.sh) :

```
#!/bin/sh
cd $PBS_O_WORKDIR
./inc-seq
```

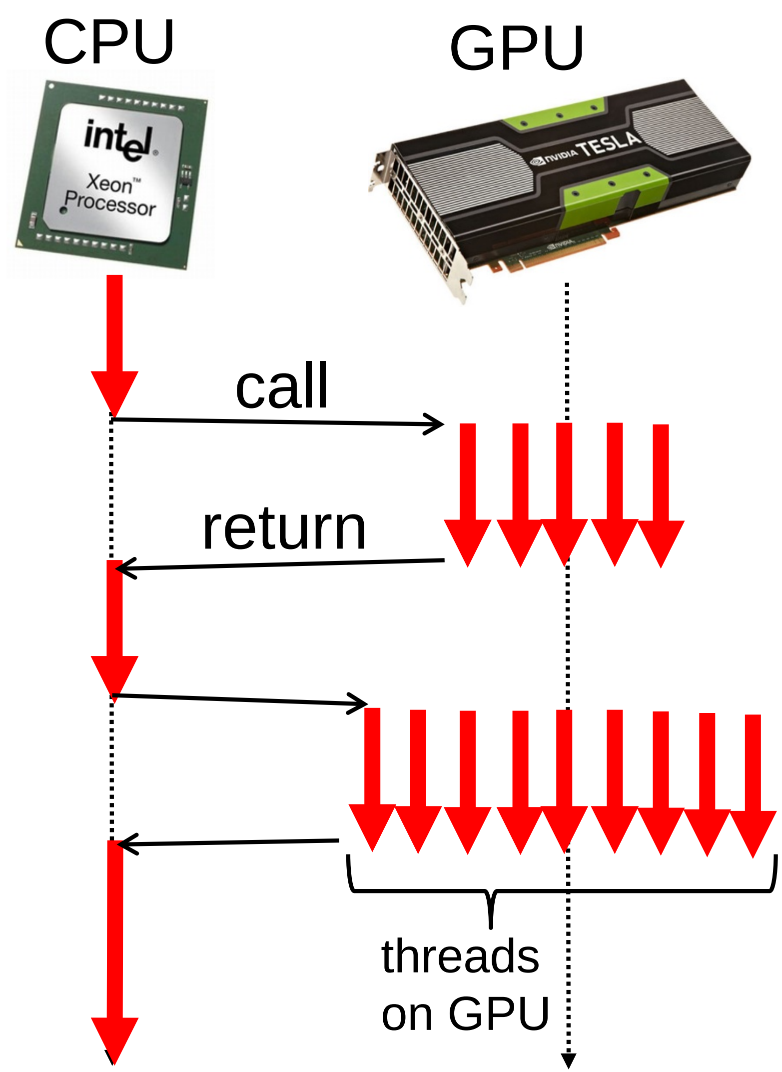
(2) Throw the job with “t2sub”

Group name is this course

```
t2sub –q S –W group_list=t2g-ppcomp
      –l ncpus=1 –l gpus=1 ./job.sh
```

Number of used GPUs per node

Programming Model of CUDA



- The program starts from main() function on CPU
  - GPU only works when some tasks are asked by CPU
  - Functions running on GPUs = GPU kernel functions
- CPU and GPU has distinct memory space
  - Host memory on CPU
  - Device memory on GPU
  - "Distributed memory model" here
- Many threads run on a GPU
  - Threads can share data on device memory

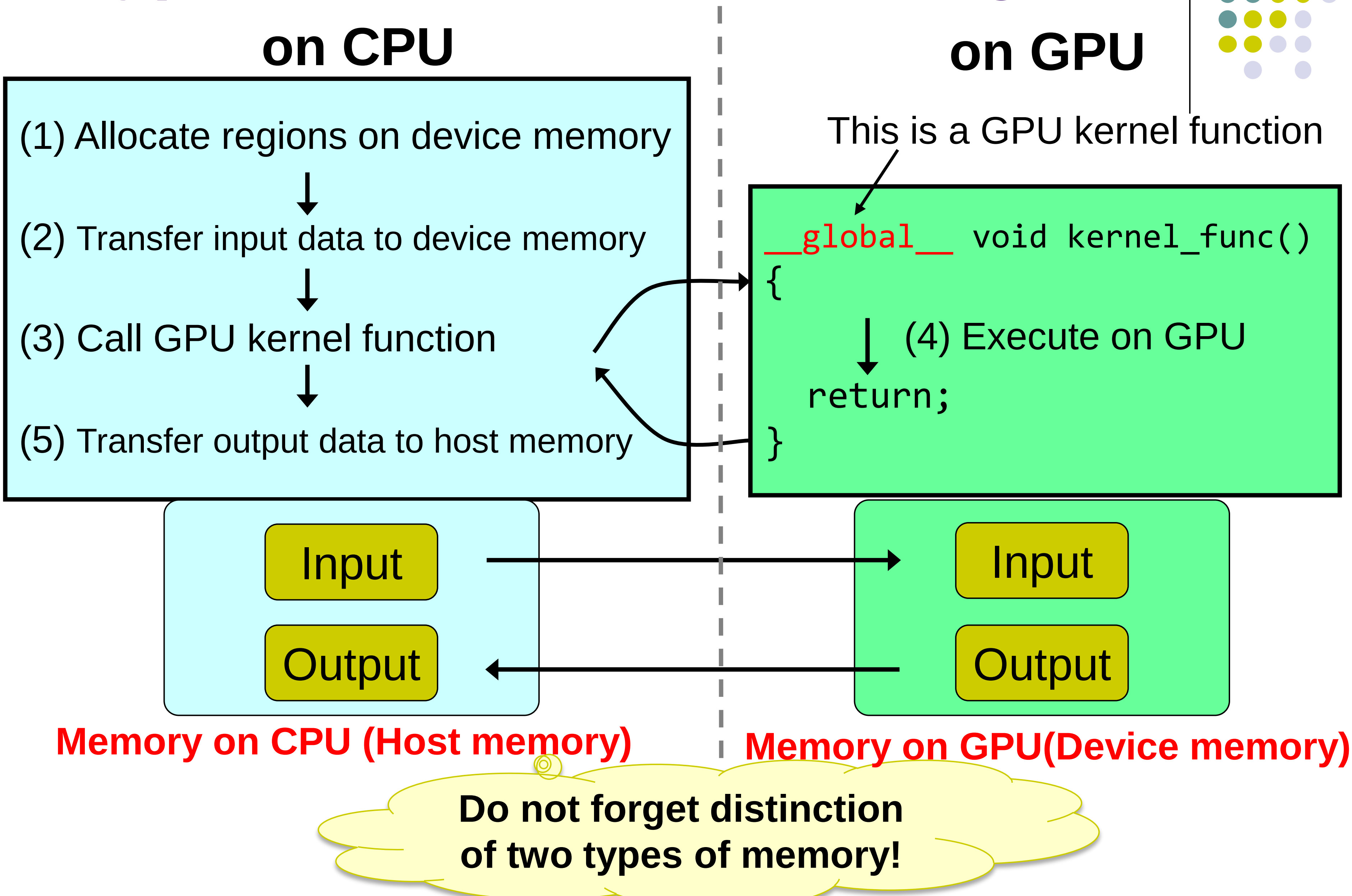
→ "Shared memory model" here

Structure of a CUDA Program

Host Functions + GPU Kernel Functions

- Two types of functions are in “.cu” files
- Host functions
  - Functions run on CPU, including main()
  - It can
    - transfer data to/from GPU
    - call GPU kernel function
- GPU kernel functions
  - Functions run on GPU
  - Have keywords “__global__” or “__device__” (later)

Typical Flow of a CUDA Program



Step (1) in inc-seq
Allocate a Region on GPU

- `cudaMalloc(void **devpp, size_t count)`
  → allocate a memory region on device memory
  - devpp: result pointer is put into *devpp
  - count: region size in bytes

cf) Allocate an int array of length 32

```
int *arrayD;
cudaMalloc((void **)&arrayD, sizeof(int) * 32);
// arrayD has the address of allocated device memory
```

Note: cudaMalloc() must be called on CPU, not on GPU

Step (2) in inc-seq Transfer Data to GPU

- `cudaMemcpy(void *dst, const void *src, size_t count, cudaMemcpyKind kind)`

  → Transfer data between host memory and device memory

  - dst: Destination address
  - src: Source address
  - count: Transfer size in bytes
  - kind: Transfer type. When transferring from CPU to GPU, this is `cudaMemcpyHostToDevice`

cf) Transfer contents of arrayH on CPU to arrayD on GPU

```
int arrayH[32];
     :
cudaMemcpy(arrayD, arrayH, sizeof(int)*32,
           cudaMemcpyHostToDevice);
```

Note: cudaMemcpu() must be called on CPU, not on GPU

Step (3) in inc-seq
Call a GPU Kernel Function from CPU

- ```
  kernel_func<<<grid_dim, block_dim>>>(kernel_param1, …);
  ```
  - kernel_func: Function name
  - kernel_param: Parameters to the function

cf) Call a GPU kernel function “inc”

Parameter 2: Array length

```
inc<<<1, 1>>>(arrayD, 32);
```

Function name.
“inc” must be declared with __global__ keyword

New syntax in CUDA!!
Here number of threads are described
(in next class)

Parameter 1:
A pointer of input array.
This must be arrayD on device memory

What if arrayH on host is specified?
```

Step (4) in inc-seq
Execute a GPU Kernel Function on GPU

- Function must be have a keyword "__global__"
  note： 2 underbars before global, 2 underbars after global
- Return type must be "void" (cannot return a value)
- In the function, GPU can access to device memory. cannot access to host memory

cf)： Increment elements of int array (by 1 thread)

```
__global__ void inc(int *array, int len)
{
    int i;
    for (i = 0; i < len; i++) array[i]++;
    return;
}
```

Step (5) in inc-seq Transfer Data from GPU

- `cudaMemcpy` is used
- Transfer type should be `cudaMemcpyDeviceToHost`

cf) Transfer contents of arrayD on GPU to arrayH on CPU

destination source

```
cudaMemcpy(arrayH, arrayD, sizeof(int)*32,
           cudaMemcpyDeviceToHost);
```

To discard a region on device memory,use `cudaFree(arrayD);`

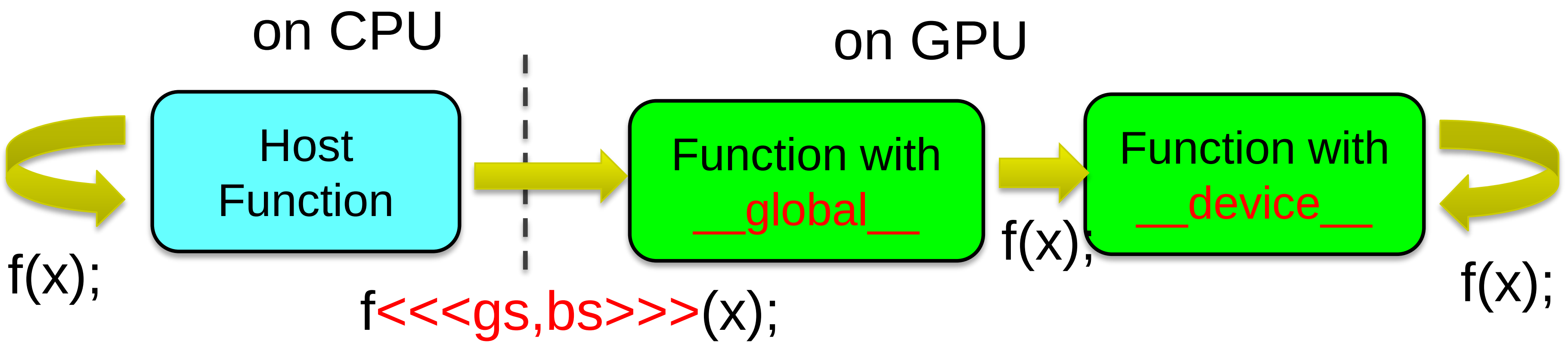
Two Types of GPU Kernel Functions

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



What Can be Done on GPUs?

- 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
  - Exceptionally, printf() is OK
    - Appendix B.17 in “Programming Guide”
  - Several mathematical functions, sin(), sqrt()… are OK
    - Appendix B.7 in “Programming Guide”
  - Calling malloc()/free() on GPU is OK, if the size is small
    - Appendix B.18 in “Programming Guide”
    - If we need large regions on device memory, call cudaMalloc() from <u>CPU</u>

Assignments in this Course

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

Due Date May 8	Part 1 OpenMP	[O1] diffusion [O2] sort [O3] free	*Select 1 problem*
Due Date May 29	Part 2 MPI	[M1] [M2] [M3]	*Select 1 problem*
Due Date June 12	Part 3 GPU	[G1] [G2] [G3?]	*Select 1 problem*

Select 2 parts

Assignments in GPU Part (1)

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

Due date: June 12 (Monday)

[G1] Parallelize "diffusion" sample program by CUDA (explained later).

Optional:

- Make array sizes variable parameters
- Improve performance further
  - Different assignment of threads and elements
  - Using shared memory
  - etc.

Assignments in GPU Part(2)

[G2] Evaluate speed of “mm-cuda” in detail (explained later).

- Use various matrices sizes
- Evaluate effects of data transfer (cudaMemcpy) cost
- Compare with CPU (OpenMP) version

Optional :

- You may change the program
  - Different data format
  - Different assignment of threads and elements
  - Using shared memory
  - etc

Assignments in GPU Part (3)

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

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

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

- 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 TSUBAME2
    - With varying number of processor cores
    - With varying problem sizes
    - Discussion with your findings
    - Other machines than TSUBAME2 are ok, if available

Next Class:

- GPU Programming (2)
  - Parallelization with massive number of threads