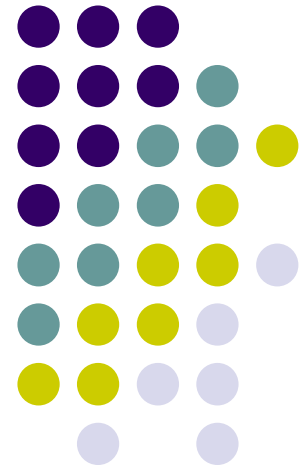


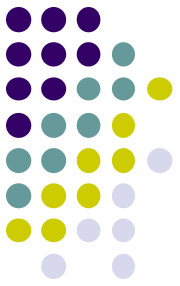
2018
Practical Parallel Computing
(実践的並列コンピューティング)
No. 14

GPU Programming (4)

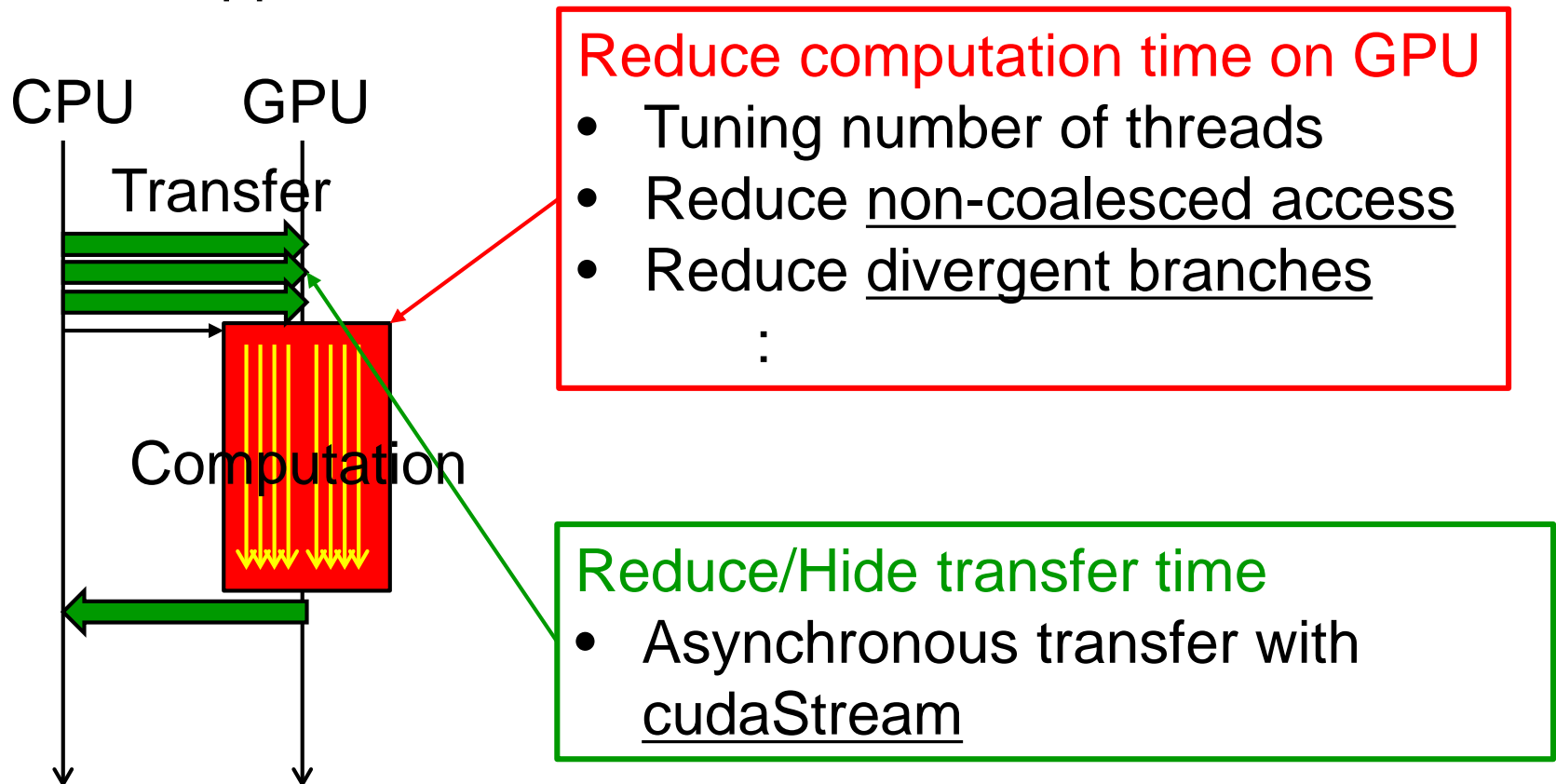
Toshio Endo
School of Computing & GSIC
endo@is.titech.ac.jp

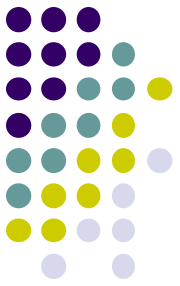


Considering Performance of GPU Programs



- It is best to reduce of amount of computation & transfer!
- Other approaches are ...





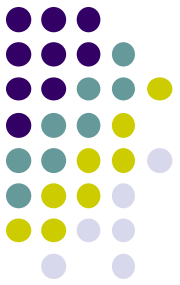
Official Documents

CUDA

- <https://docs.nvidia.com/cuda/>

OpenACC

- <https://www.openacc.org/resources>



Tuning Number of Threads

Specifying number of threads in CUDA

gridDim blockDim

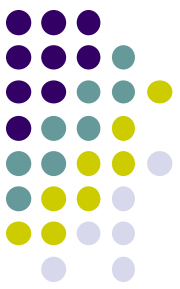
```
func<<<dim3(gx, gy, gz), dim3(bx, by, bz)>>> (...);
```

→ $(gx * gy * gz) * (bx * by * bz)$ threads are created

- When creating 1,000,000 threads,
 - <<<1, 1000000>>> causes an error
 - blockDim must be ≤ 1024
 - <<<1000000, 1>>> can work, but slow → Why?

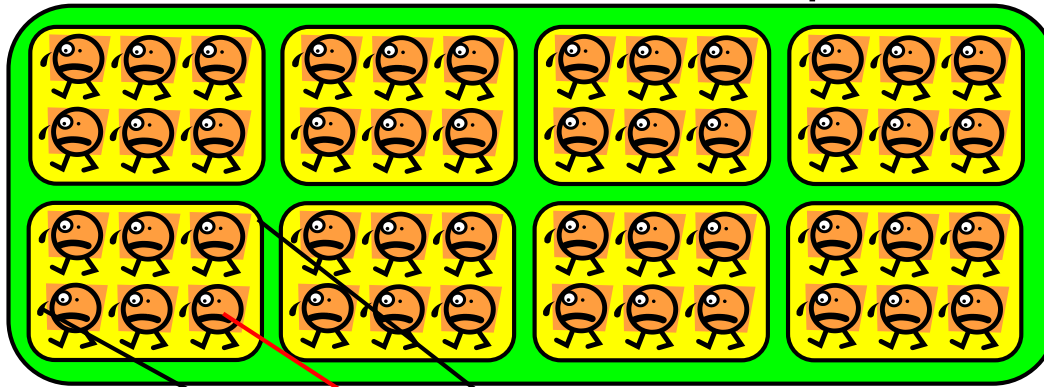
※ For OpenACC, num_gangs \Leftrightarrow gridDim, vector_length \Leftrightarrow blockDim 4

Why Do We Have to Specify both `gridDim` and `blockDim`?



- and why did NVIDIA decide so?

→ Hierarchical structure of GPU processor is considered



Structure of P100 GPU
(16nm, 15Billion transistors)

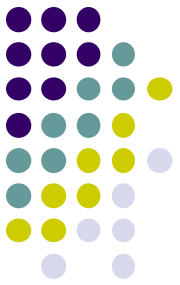
1 GPU = 56 SMX

1 SMX = 64 CUDA core

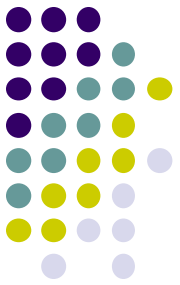


610mm²

Mapping between Threads and Cores

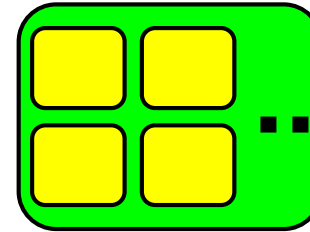
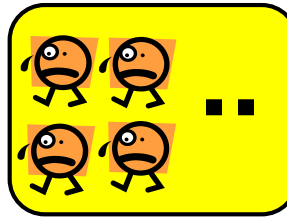


- $M (\geq 1)$ thread blocks run on 1 SMX
 - At least 56 blocks are needed to use all SMXs on P100
 - `gridDim (gx*gy*gz)` should be ≥ 56
- $N (\geq 1)$ thread run on a CUDA core
 - At least $56*64=3584$ threads in total are needed to use all CUDA cores on K20X
 - `Total threads (gx*gy*gz * bx*by*bz)` should be ≥ 3584
- 32 consecutive threads (in a block) are batched (called a warp) and scheduled
 - At least 32 threads per block are needed for performance
 - `blockDim (bx*by*bz)` should be ≥ 32

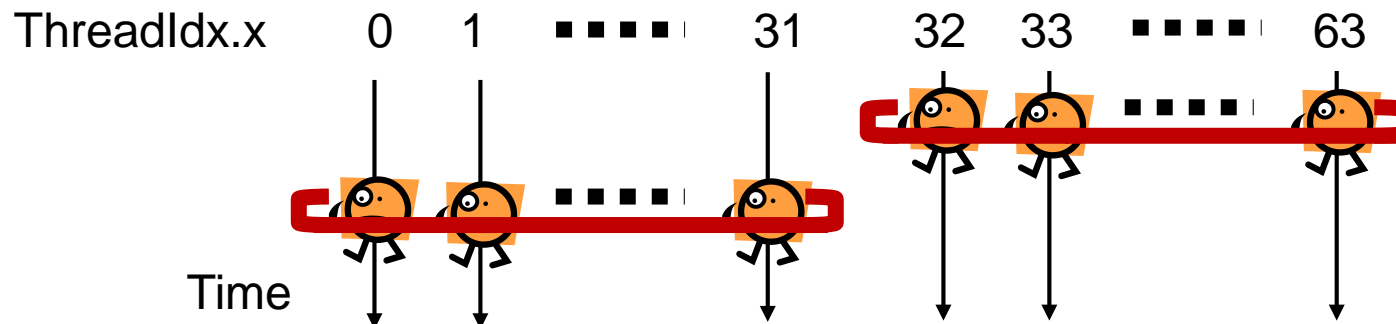


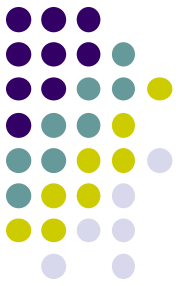
Warp: Internal Execution Unit

thread < **warp** < thread block < grid



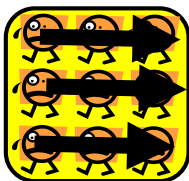
- Threads in a thread block are internally divided into “**warp**”, a group of contiguous 32 threads
- 32 threads in a warp always are executed synchronously
 - They execute the same instruction simultaneously
 - There is only one program counter for 32 threads! → Structure of a GPU core is simplified



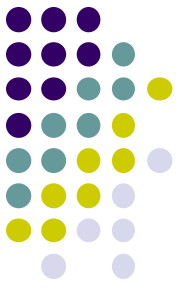


Observations due to Warps

- If number of threads per block (blockDim) is not $32 \times n$, it is inefficient
 - Even if blockDim=1, the system creates a warp for it
- Characteristics in memory addresses accessed by threads in a warp affect the performance
 - Coalesced accesses are fast
- Characteristics in branch (such as “if”) affect the performance
 - Divergent branches are slow



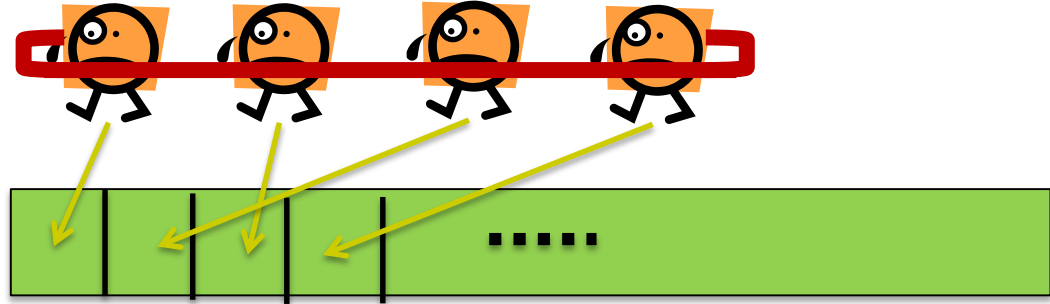
⌘ In multi-dimensional cases (blockDim.y>1 or blockDim.z>1), “neighborhood” is defined by x-dimension



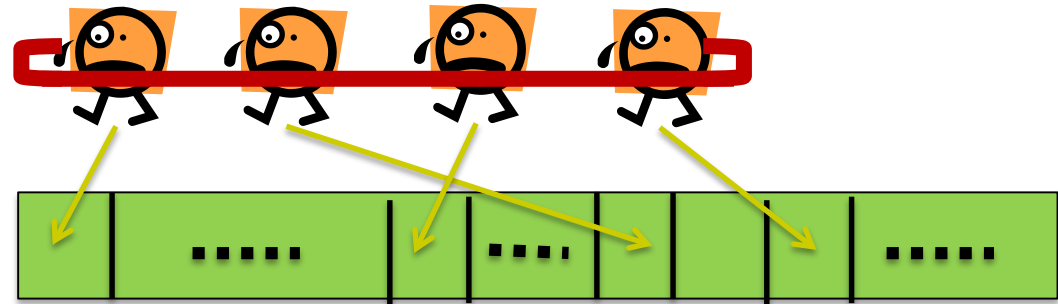
Coalesced Access

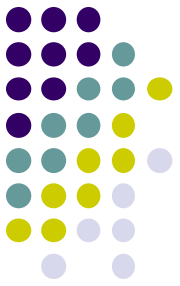
- When threads in a warp access “neighbor” address on memory (**coalesced access**), it is more efficient

Coalesced access
→ **Faster**



Non-coalesced access
→ **Slower**

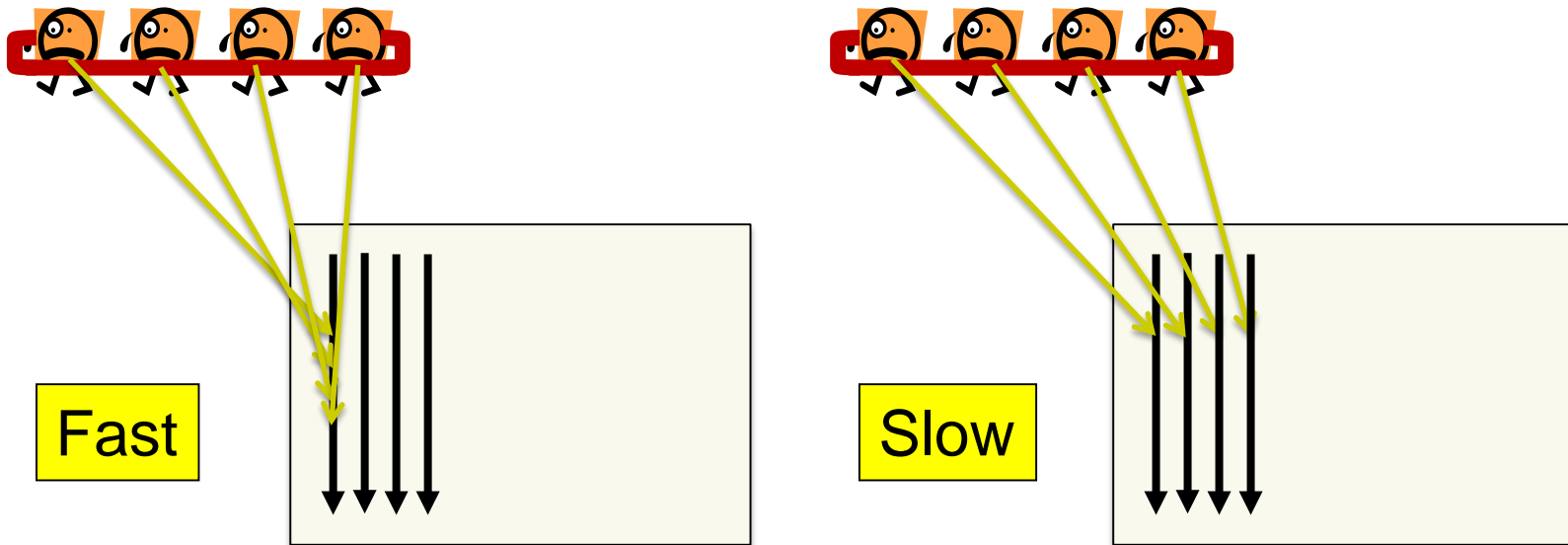




Accesses in mm Samples

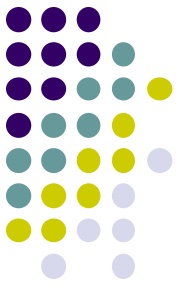
- Accesses in `mm-cuda`, `mm-acc` are coalesced
- Accesses in `mm-slow-cuda`, `mm-slow-acc` are coalesced

We should see “what data are accessed by threads in a warp simultaneously



matrices in column-major format

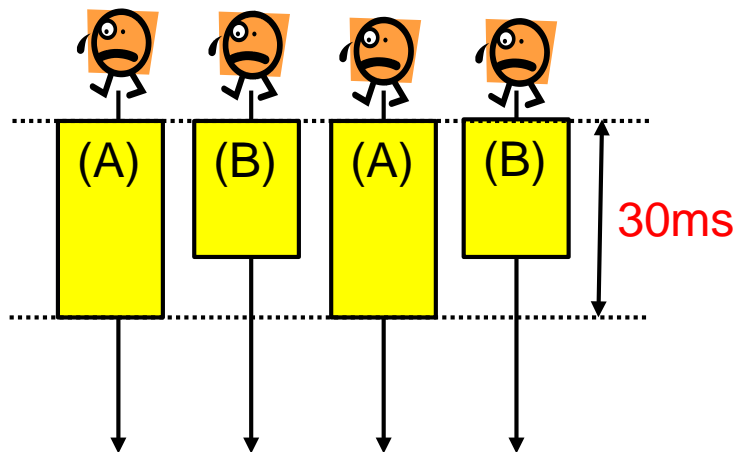
Considering Branches in Parallel Programs



Consider this code. How long is execution time?

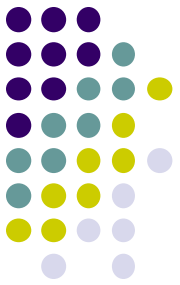
```
if (thread-id % 2 == 0) {  
    : // (A) 30msec  
} else {  
    : // (B) 20msec  
}
```

On CPU (OpenMP)

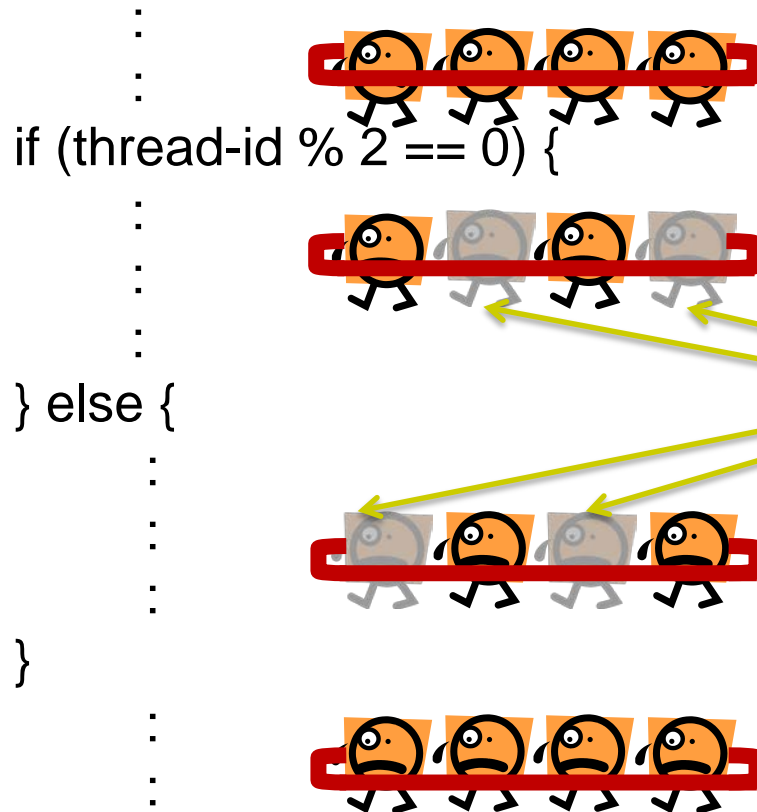


On GPU, threads in a warp must execute the same instruction. What happens?





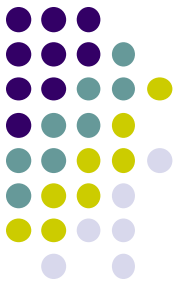
Branches on GPU (1)



Some threads are made sleep
Both “then” and “else” are executed!

→ Answer to previous question is **50ms** !

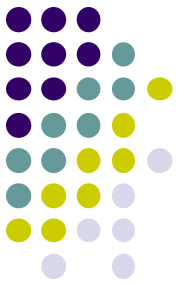
⌘ Similar cases happen in for, while...



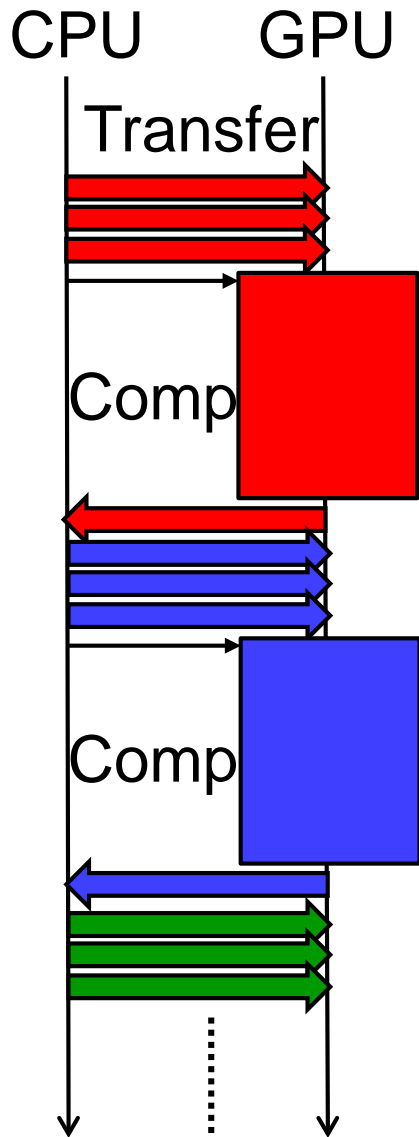
Branches on GPU (2)

- As exceptional cases, if threads in a warp “agree” in branch condition, either “then” part or “else” part is executed → **Efficient!**
- If there is difference of opinion (previous page), it is called a **divergent branch**

→ Agreement among buddies is important



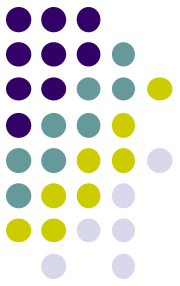
Considering Data Transfer Costs



Example case: We are going to multiple matrix multiplications.

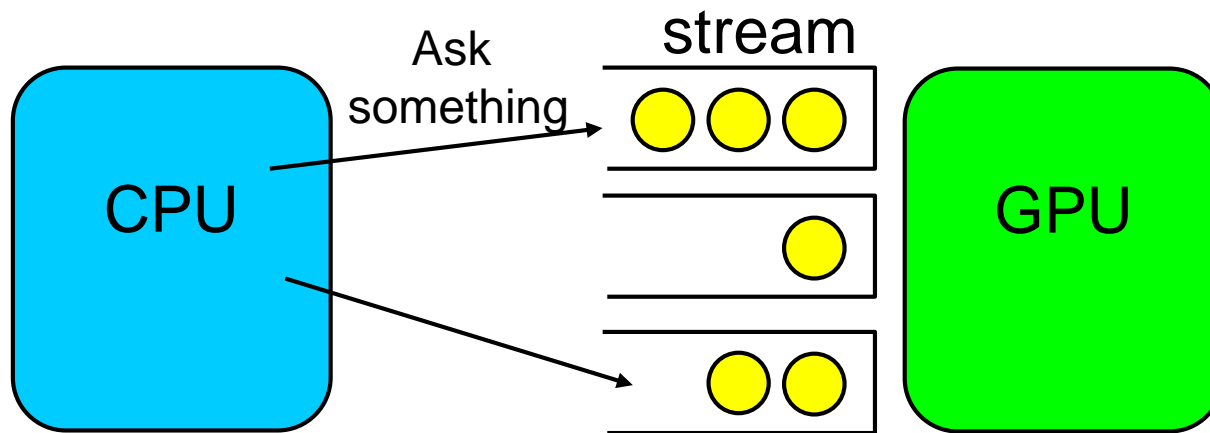
- Input data are on host memory
 - $C1 = A1 \times B1$
 - $C2 = A2 \times B2$
 -
 - $Cn = An \times Bn$
- In default, GPU cannot compute during transfer
- Hiding transfer costs is a good idea
 - `cudaStream` in CUDA
 - `async` in OpenACC

Asynchronous Executions with `cudaStream` (1)



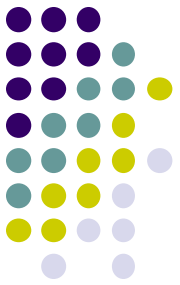
What are `streams`?

- GPU's “service counters” that accept tasks from CPU
 - Each stream looks like a queue
- “Tasks” from CPU to GPU include
 - Data transfer (Host → Device)
 - GPU kernel function call
 - Data transfer (Device → Host)



Sample Program: [~endo-t-ac/ppcomp/18/array-async-cuda/](https://github.com/endo-t-ac/ppcomp/18/array-async-cuda/)

Asynchronous Executions with `cudaStream` (2)



Create a stream

```
cudaStream_t str;  
cudaStreamCreate(&str); // Create a stream
```

Data transfer using a specific stream

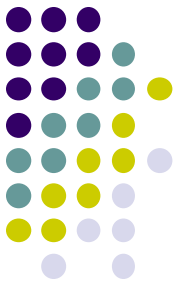
```
cudaMemcpyAsync(dst, src, size, type, str);
```

Call GPU kernel function using a stream

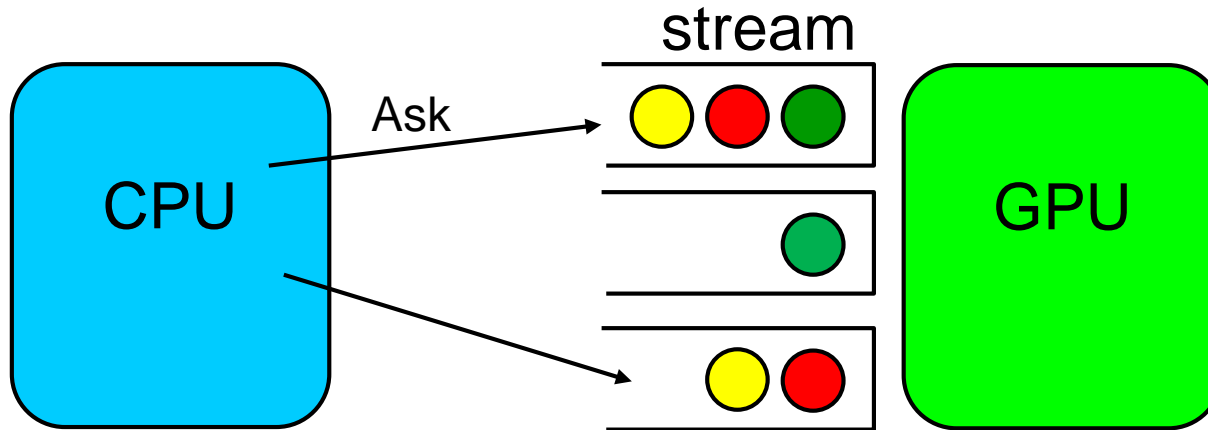
```
func<<<gs, bs, 0, str>>>( ... );  
// 3rd parameter is related to for “shared memory”
```

Wait until all tasks on a stream are finished

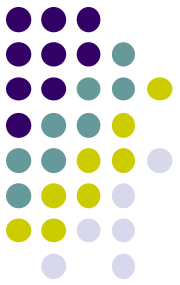
```
cudaStreamSynchronize(str);
```



How GPU does Tasks



- Tasks on the same stream is done in FIFO
- If tasks are in different streams, and have different kinds, they may be done simultaneously
 - Kinds: $H \rightarrow D$, kernel, $D \rightarrow H$
 - Note: If tasks are in the same kind, no speed up



“Async” Option in OpenACC

- `kernels`, `data`, `enter data...` directives can have **async** option

```
#pragma acc data copy ... async(1)  
#pragma acc kernels async(2)
```

Integer: streamID

→ Execution (of copy or kernel) is non-blocking

- Waiting the end of non-blocking operations

```
#pragma acc wait(2)  
#pragma acc wait
```

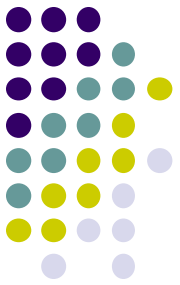
Integer: streamID

Sample Program: [~endo-t-ac/ppcomp/18/array-async-cuda/](https://github.com/endo-t-ac/ppcomp/18/array-async-cuda/)

⌘ The program is more complex than expected ☹

This is not a unique solution;
Use 2 or 3 streams repeatedly → we can save
memory and stream resources

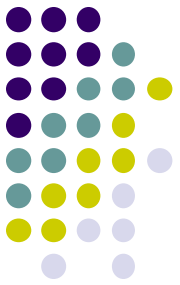




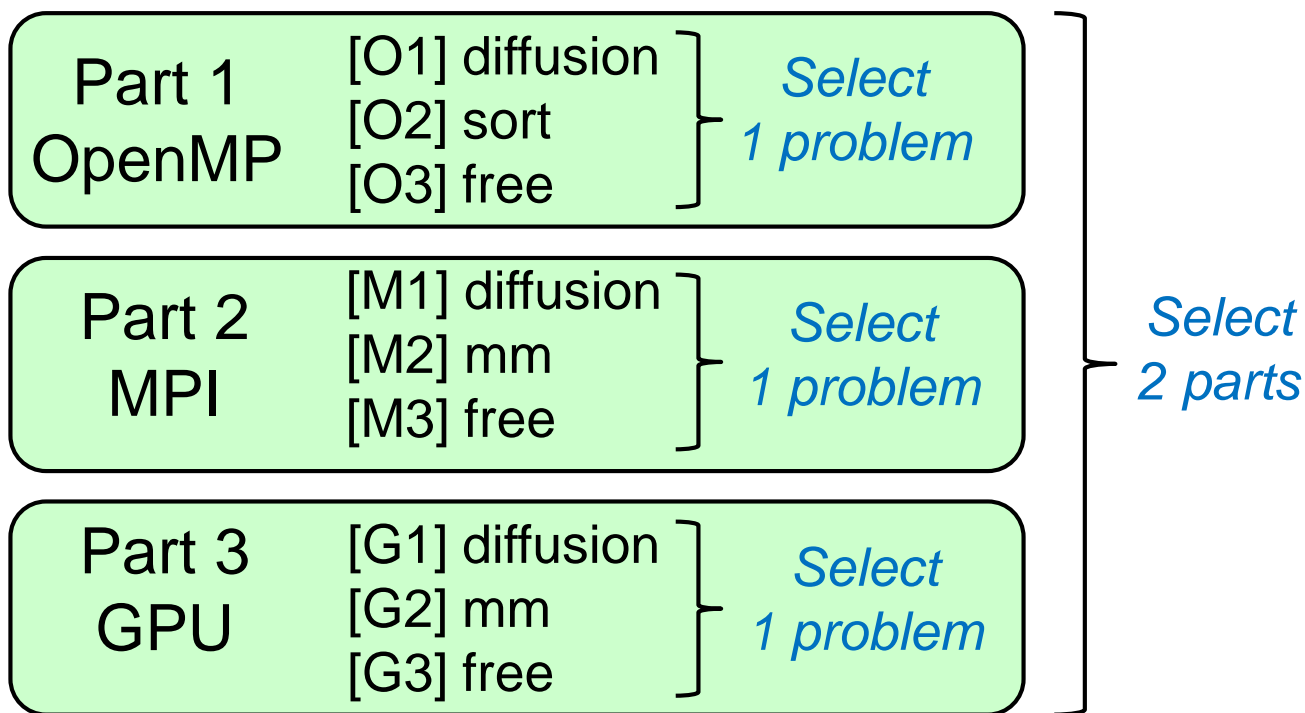
More Things to Study

- Using CUDA shared memory
 - fast and small memory than device memory
- Unified memory in recent CUDA
 - `cudaMemcpy` can be omitted for automatic data transfer
- Using multiple GPUs towards petascale computation
 - MPI+CUDA!
- More and more...

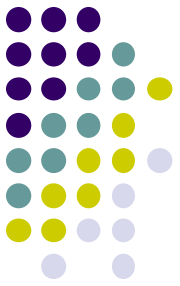
Assignments in this Course



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



Assignments in GPU Part (Abstract)



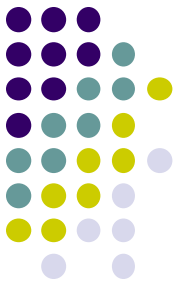
Choose one of [G1]—[G3], and submit a report

Due date: June 14 (Thursday)

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

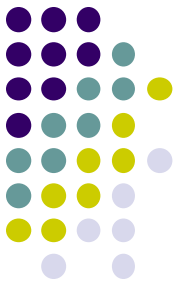
[G2] Improve “mm-acc” or “mm-cuda” to support larger matrices

[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 (Final):

- Invited talk for distributed framework with GPUs