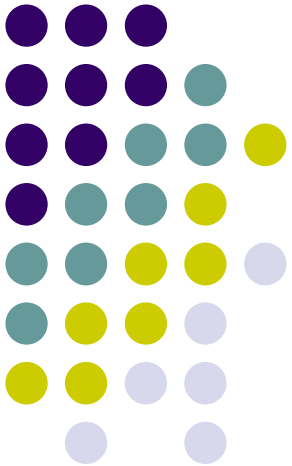
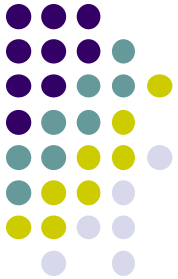
Practical Parallel Computing (実践的並列コンピューティング)

2025 Class No.11
[CUDA Part] (3)
Performance Improvement on GPU



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 ← We are here (3/3)
  - Report (elective)
- MPI Part
  - 3 classes
  - Report (elective)

Note:
Modification to ppcomp-ex github

There were bugs in ppcomp-ex/cuda related to [C1][C2]

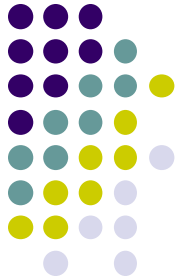
Please update your ppcomp-ex directory

```
cd ppcomp-ex   // your ppcomp-ex directory
git pull
// ➔ cuda/diffusion/Makefile is modified
//  ➔ cuda/bsort/bsort.c is renamed to bsort.cu
```

Please confirm cuda/diffusion and cuda/bsort work

Thanks for cooperation

Speed of GPU Programs and GPU Architecture



Case 1: How should block-size be determined?

When creating 1,000,000 threads,

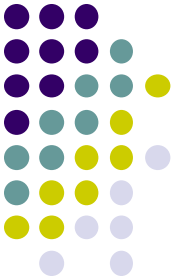
- <<<1, 1000000>>> causes an error
  - blockDim must be <= 1024
- <<<1000000, 1>>> can work, but slow
- <<<1000, 1000>>> is faster → Why?

Case 2: How should each thread access memory?

- In cuda/mm, (x = row,y = col) and (x = col, y = row) shows different speed

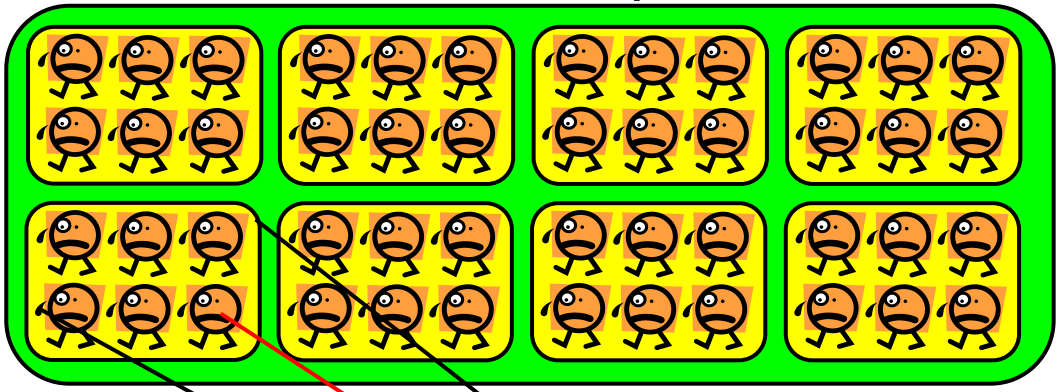
Knowledge of GPU architecture helps understanding of speeds

Why Do We Have to Specify both gridDim and blockDim?



- and why did NVIDIA decide so?

→ Hierarchical structure of GPU processor is considered



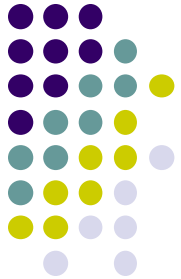
Structure of H100 SXM GPU

1 GPU = 132 SMs
1 SM = 128 CUDA cores

→ 1GPU=16,896 CUDA cores

In TSUBAME4 interactive node (1/2 GPU), ~7680 CUDA cores

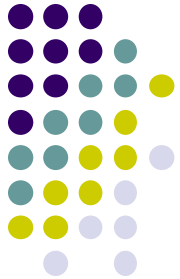
Mapping between Threads and Cores



- 1 thread blocks (or more) run on 1 SM
  - → At least 132 blocks are needed to use all SMs on H100
  - → gridDim (gx*gy*gz) should be ≧132
- 1 thread (or more) run on a CUDA core
  - → At least 132*128=16,896 threads in total are needed to use all CUDA cores on H100
  - → Total threads (gx*gy*gz * bx*by*bz) should be ≧16896

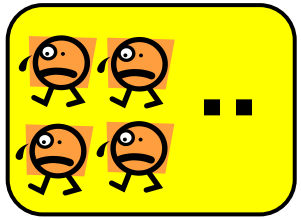
We need more knowledge on GPU architecture to discuss speed

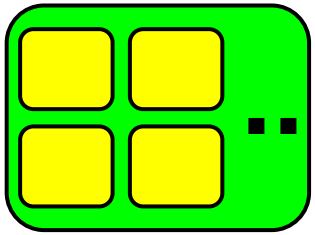
Warp: Internal Execution Unit (1)



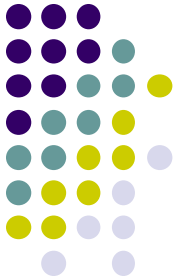
- 32 consecutive threads (in a block) are batched (called a *warp*)
  - → At least 32 threads per block are needed for performance
  - → blockDim (bx*by*bz) should be ≧32

thread < warp < thread block < grid

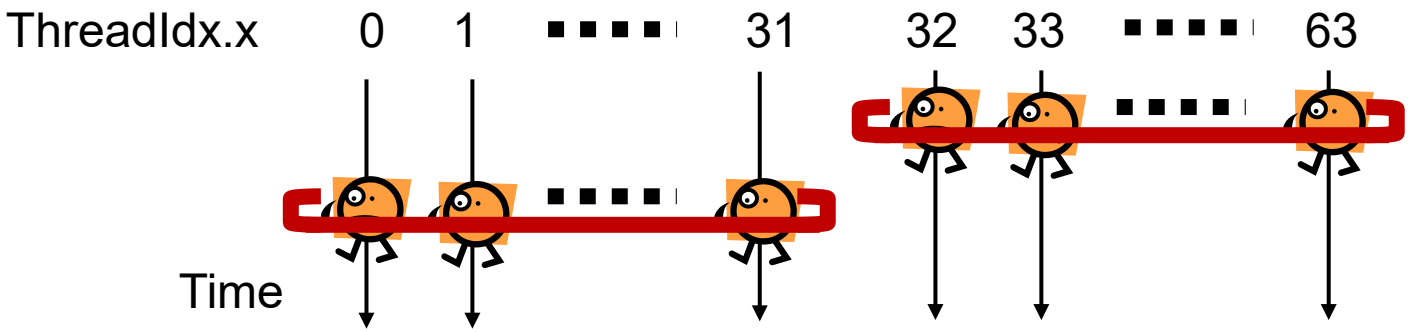




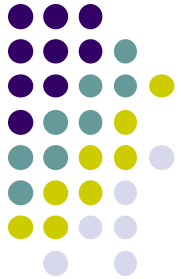
Warp: Internal Execution Unit (2)



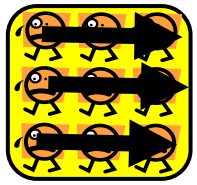
- Threads in a thread block are internally divided into "warp", a group of contiguous 32 threads
- 32 threads in a warp always are executed synchronously
  - They execute the same instruction simultaneously
  - Only 1 program counter for 32 threads → GPU hardware is simplified
  - Actually 32 threads are executed on 16 CUDA cores



Observations due to Warps



- If number of threads per block (blockDim) is not *32 x 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

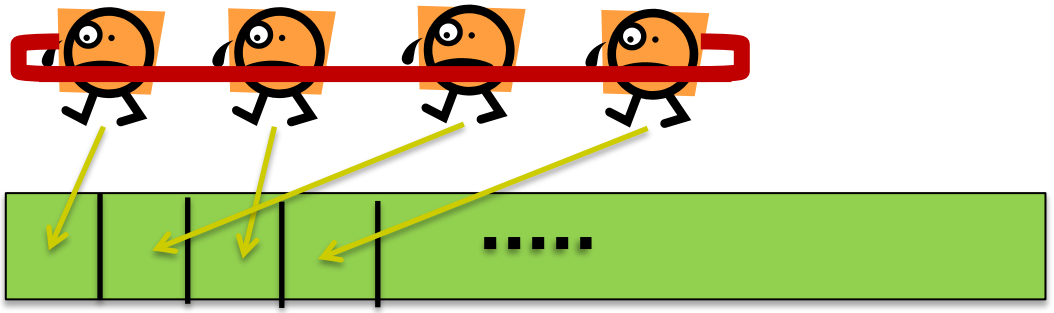


※ In multi-dimensional cases (blockDim.y>1 or blockDim.z>1), “neighborhood” is defined by x-dimension

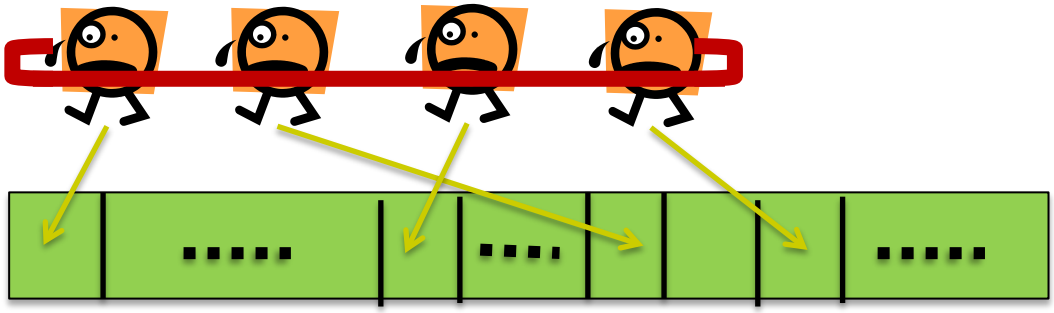
Coalesced Memory Access

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

Coalesced access
→ **Faster**



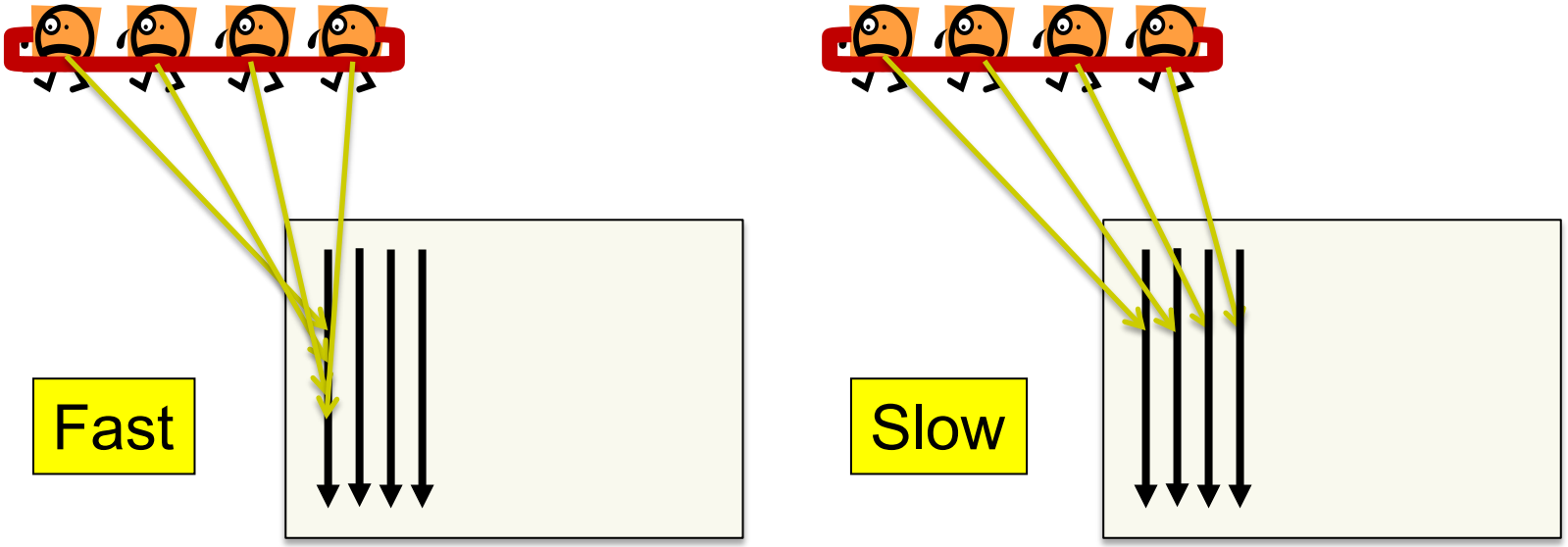
Non-coalesced access
→ **Slower**



Accesses in cuda/mm Sample (Related to [C3], optional)

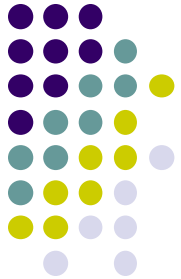
- cuda/mm: (x = row,y = col) → coalesced and fast
- cuda/mm-nc: (x = col, y = row) → non-coalesced and slow
  - ppcomp-ex/cuda/mm-nc

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



※In these samples, matrices are in column-major format

Why #threads >> #cores is Good on GPUs? (1)



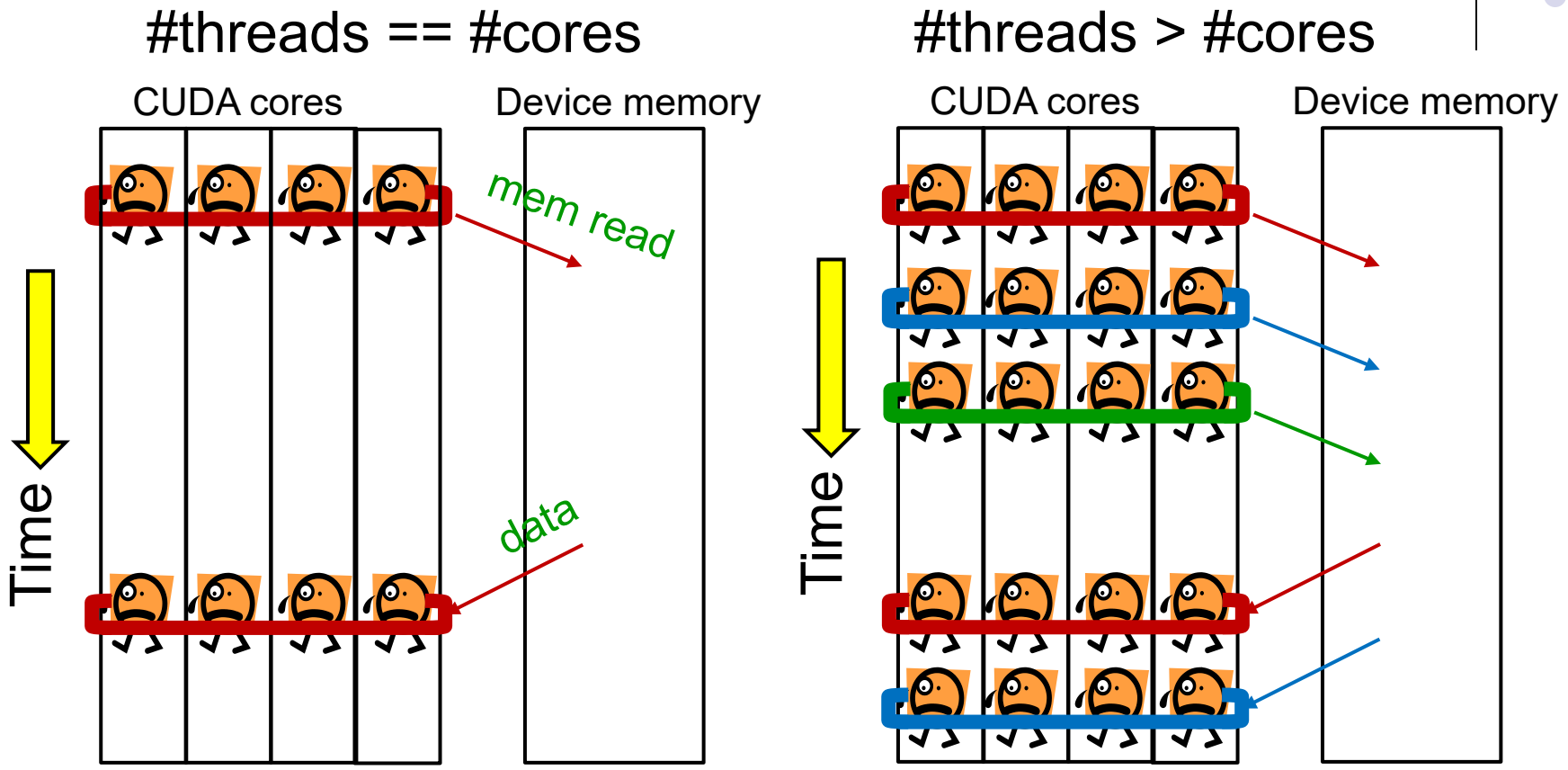
GPU architecture matters:

- Computation is fast
  - cf: cij += ail*blj;
  - Computation on registers takes a few clocks
- Memory access is slow
  - cf: ail = A[i+l*m];
  - If cache miss happens, memory access takes >100 clocks!
  - If non-coalesced access happens, it is even worse!

- GPU supports very fast (~1 clock) context switches among warps

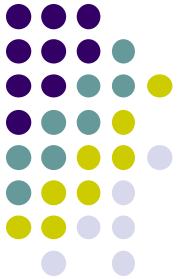
  ➔ With many threads, memory access latency can be hidden

Why #threads >> #cores is Good on GPUs? (2)



- If #threads > #cores, we can reduce “idle” times! ➔ Faster

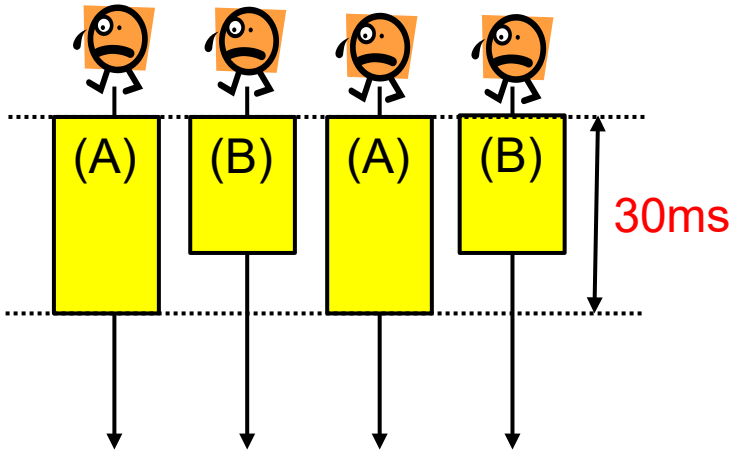
Considering Branches in Parallel Programs



Consider this code. How long is execution time?

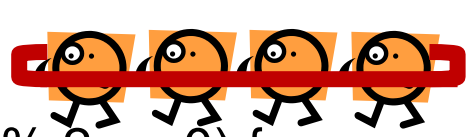
```
if (thread-id % 2 == 0) {
        :  // (A) 30msec
} else {
        :  // (B) 20msec
}
```

On CPU (OpenMP)



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

Branches on GPU (1)



```
if (thread-id % 2 == 0) {
```

```
} else {
```

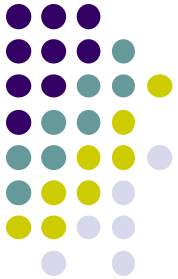
```
}
```

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

→ Answer to previous question is 50ms !

※ Similar cases happen in for, while…

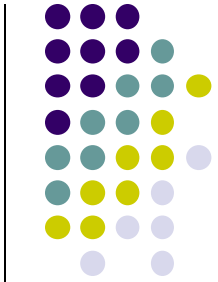
Branches on GPU (2)



- 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 (threads in a warp) is important for speed

Considering Data Transfer Costs of mm Sample

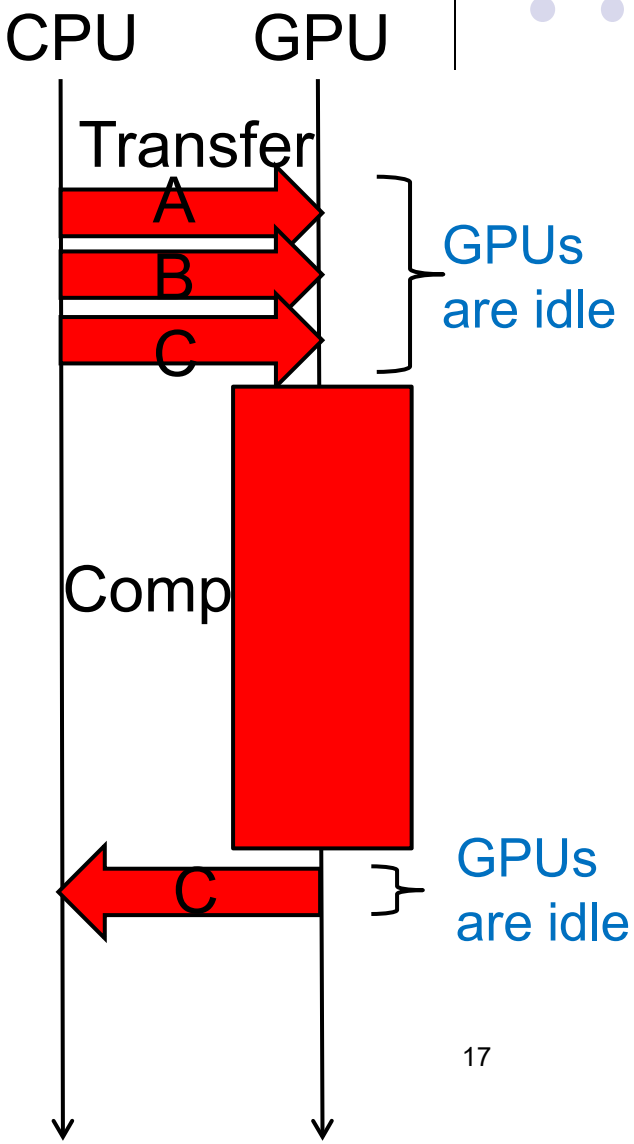


- In mm sample, the speed is degraded by data transfer costs ☹

- This can be improved by combination of:

1. Split computation
2. Using CUDA streams

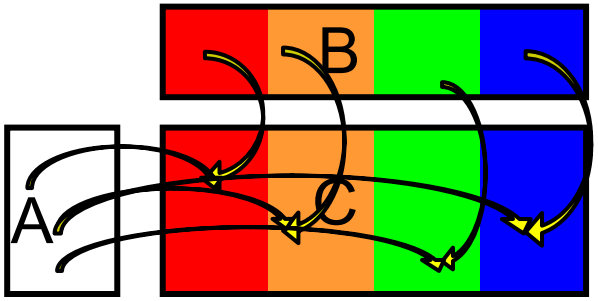
→ The faster sample is at ppcomp-ex/cuda/mm-str/



Split mm Computation (1)

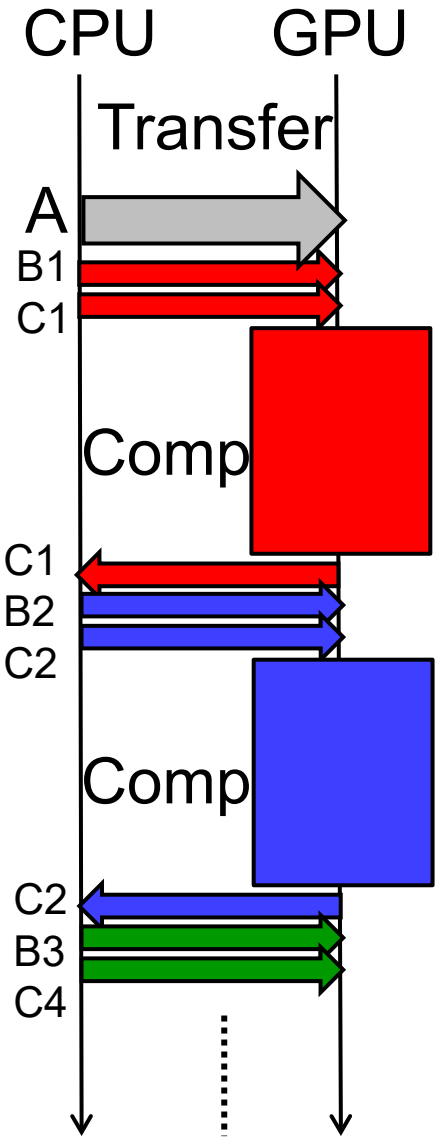
- Computation of "C ← A×B" is split by splitting B and C vertically
  - $C_1 \leftarrow A \times B_1$, $C_2 \leftarrow A \times B_2$, … , $C_n \leftarrow A \times B_n$

  The n computations are independent each other



A is reused for all computations

Split mm Computation (2)



Algorithm:

(1) Copy A from CPU to GPU

(2) For each partition i (sequentially)

  (1) Copy Bi and Ci to GPU

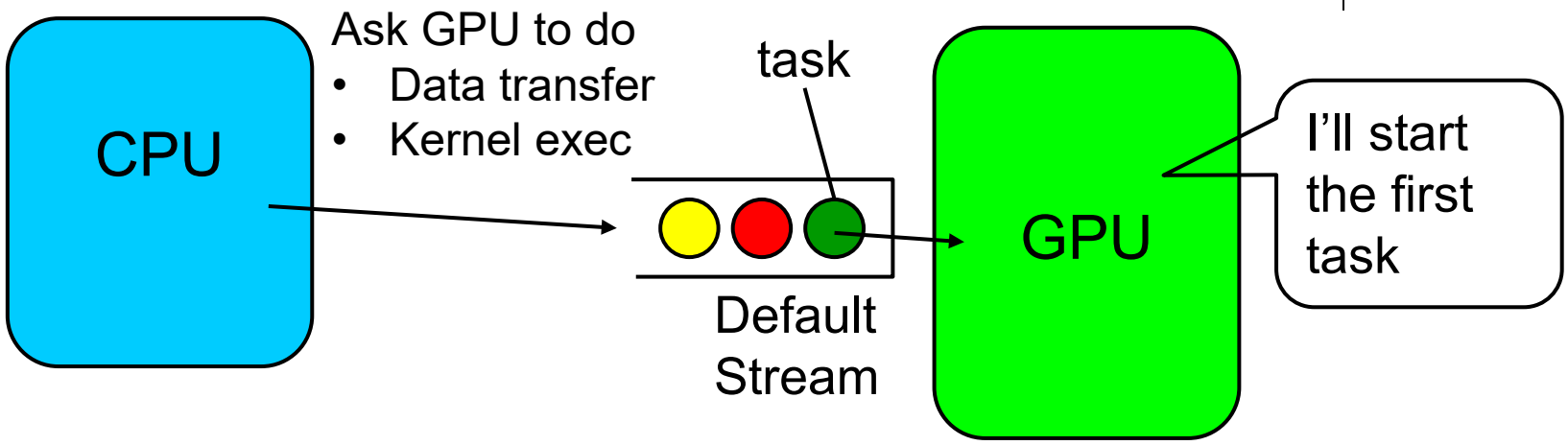
  (2) Compute $C_i \leftarrow A \times B_i$

  (3) Copy back Ci to GPU

This does NOT improve speed yet, since neither total computation costs nor total transfer costs change

→ cudaStream is useful for hiding transfer costs

How GPU Executes Tasks
(Without multiple streams)

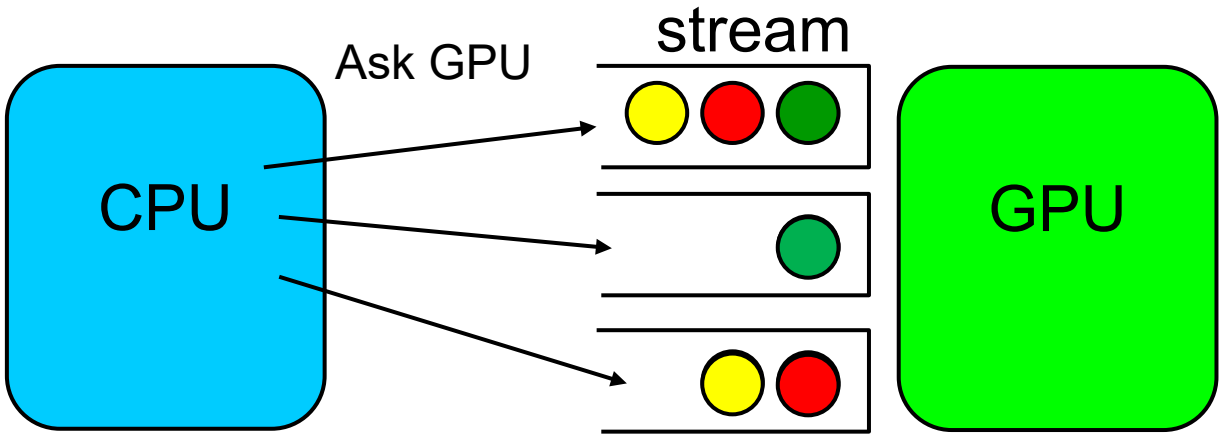


- A GPU is idle until asked to do something by CPU
- CPU asks the GPU to do one of followings (called tasks here)
  - Data transfer (Host → Device) or
  - GPU Kernel function execution or
  - Data transfer (Device → Host)
- Then the task is put on a FIFO queue, called *default stream*
- GPU takes a task from the stream and executes it in FIFO

Asynchronous Executions with cudaStream (1)

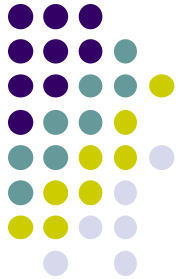
What are streams?

- GPU's "service counters" that accept tasks from CPU
  - In addition to default stream user program can create streams,
  - Each stream looks like a FIFO queue



All of CUDA sample programs, except cuda/mm-str, are using the single "default stream"

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

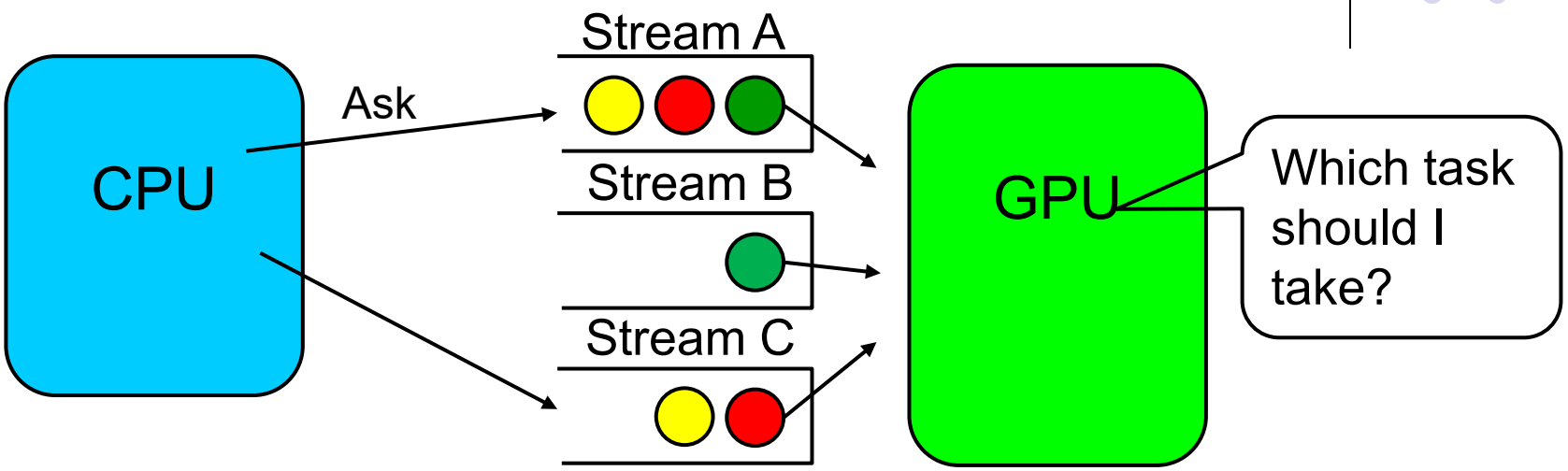
```
func<<<gs, bs, 0, str>>>( … );
// 3rd parameter is related to for “shared memory”
```

Wait until all tasks on a stream are finished

```
cudaStreamSynchronize(str);
```

※ *The default stream is expressed as* (cudaStream_t)0

How GPU Executes Tasks with Multiple Streams



- Rule: Tasks on the same stream are done in FIFO
  - The GPU considers that “tasks on one stream have dependencies, so I’ll do them in the order”
- If tasks are in different streams, and have different kinds, they may be done simultaneously
  - Kinds: Host→Device, kernel, Device→Host
  - Note: If tasks are in the same kind, no speed up

cuda/mm-str sample: Overlapping Computation and Transfer

n streams can be used for n independent "task sets"

- C1 ← A × B1 (includes H->D, Calc, D->H)
- C2 ← A × B2

….

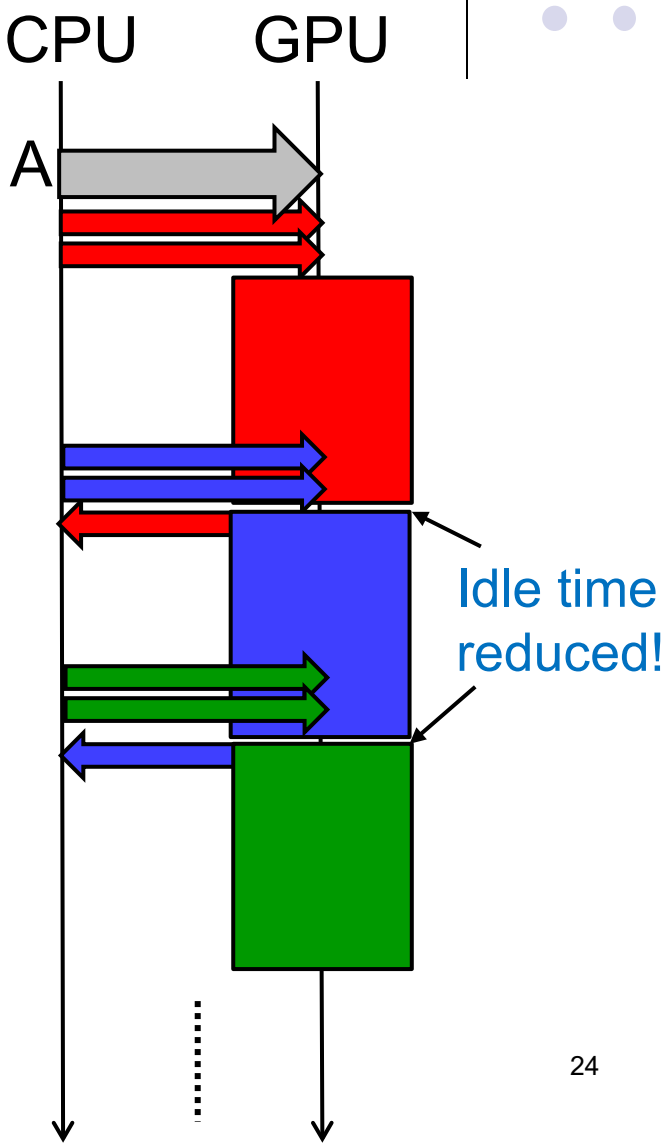
- Cn ← A × Bn

→ We will see speed up since

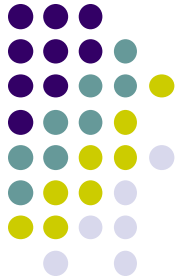
(Total comp time + Total trans time)

is improved to roughly

max(Total comp time, Total trans time)



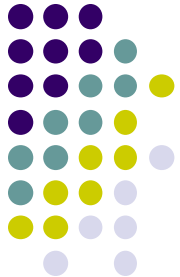
Notes on cuda/mm-str



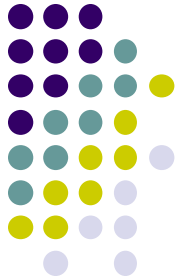
- In allocation of A, B, C on host, cudaHostAlloc() is used instead of malloc()
  - cudaHostAlloc(&A, sizeof(double)*m*k, cudaHostAllocMapped);
  - This allocates “pinned” memory ➔ cudaMemcpyAsync() gets faster
- Currently, cuda/mm-str uses NDIV streams, but there are other ways
  - Using 2 (or 3) streams and 2 (or 3) smaller buffers repeatedly

  ➔ We can save GPU memory and exceed GPU memory capacity!

More Things to Study



- Using CUDA shared memory
  - fast and small memory than device memory
  - memcpy_async for efficient copy between shared memory and device memory
- Using cache memory efficiently
- Using specialized hardware to accelerate deep learning
  - Tensor core for fast matmul (wmma::mma_sync())
- Unified memory in recent CUDA
  - cudaMemcpy can be omitted for automatic data transfer
  - Google with “cudaMallocManaged”
- Using multiple GPUs towards petascale computation
  - MPI+CUDA, MPI+OpenACC
- More and more…



Assignments in CUDA Part

Choose one of [C1]—[C4], and submit a report

Due date: May 26 (Monday)

[C1] Parallelize “diffusion” sample program by CUDA

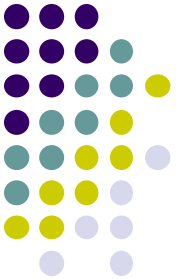
[C2] Parallelize “bsort” sample program by CUDA

[C3] Evaluate speed of “cuda/mm” sample in detail

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

For more details, please see ppcomp25-9 slides

Next Part: MPI Part



- Class #12
  - Introduction to MPI, message passing
- Class #13
  - Non-blocking communication, group communication
- Class #14
  - Performance improvement
- Class #15 (May 29, Optional)
  - TSUBAME supercomputer tour

NOTICE:

- Due date for OpenACC report is May 15 (today)
  - Changed from May 12, for machine trouble around May 10