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

Part 2: GPU

No 4: Effects of GPU Architecture

May 15, 2023

Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp

Overview of This Course

- Part 0: Introduction
  - 2 classes
- Part 1: OpenMP for shared memory programming
  - 4 classes
- Part 2: GPU programming
  - 4 classes ← We are here (4/4)
  - OpenACC (1.5 classes) and CUDA (2.5 classes)
- Part 3: MPI for distributed memory programming
  - 4 classes

Comparing OpenMP/OpenACC/CUDA

<table>
<tr><th></th><th>OpenMP</th><th>OpenACC</th><th>CUDA</th></tr>
<tr><td>Processors</td><td>CPU</td><td colspan="2">CPU+GPU</td></tr>
<tr><td>File extension</td><td colspan="2">.c, .cc</td><td>.cu</td></tr>
<tr><td>To start parallel (GPU) region</td><td>#pragma omp parallel</td><td>#pragma acc kernels</td><td rowspan="2">func<<<…, …>>>()</td></tr>
<tr><td>To specify # of threads</td><td>export OMP_NUM_THREADS=…</td><td>(num_gangs, vector_length etc)</td></tr>
<tr><td>Desirable # of threads</td><td># of CPU cores or less</td><td colspan="2"># of GPU cores or “more”</td></tr>
<tr><td>To get thread ID</td><td>omp_thread_num()</td><td>-</td><td>blockIdx, threadIdx</td></tr>
<tr><td>Parallel for loop</td><td>#pragma omp for</td><td>#pragma acc loop</td><td>-</td></tr>
<tr><td>Task parallel</td><td>#pragma omp task</td><td>-</td><td>-</td></tr>
<tr><td>To allocate device memory</td><td>-</td><td>#pragma acc data</td><td>cudaMalloc()</td></tr>
<tr><td>To copy to/from device memory</td><td>-</td><td>#pragma acc data
#pragma acc update</td><td>cudaMemcpy()</td></tr>
<tr><td>Function on GPU</td><td>-</td><td>#pragma acc routine</td><td>__global__,__device__</td></tr>
</table>

※ “# of XXX” = “The number of XXX”

Speed of GPU Programs and GPU Architecture

Advanced topic

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?

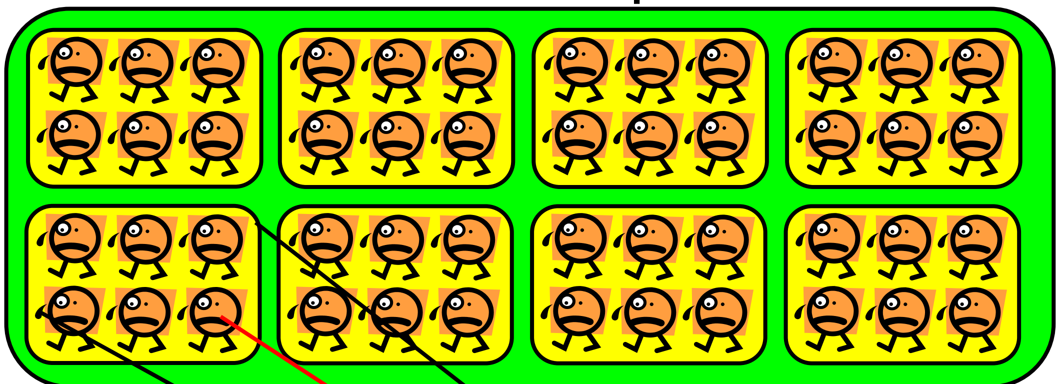
- In mm-cuda, (x = row,y = col) and (x = col, y = row) shows different speed

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 P100 GPU
(16nm, 15Billion transistors)

1 GPU = 56 SMXs
1 SMX = 64 CUDA cores
(16 cores x 4 groups)

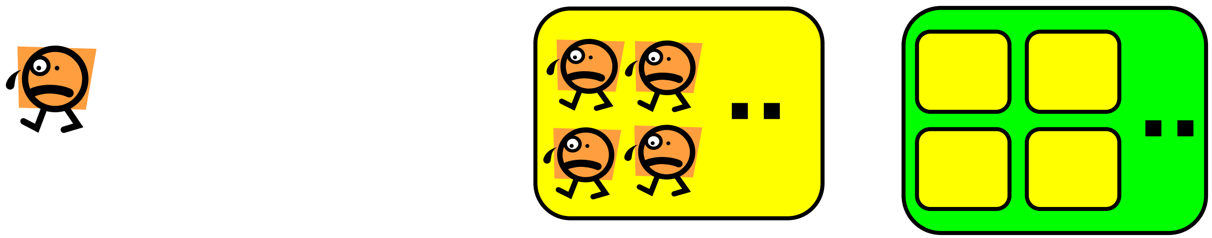
→ 1GPU=3,584 CUDA cores

Mapping between Threads and Cores

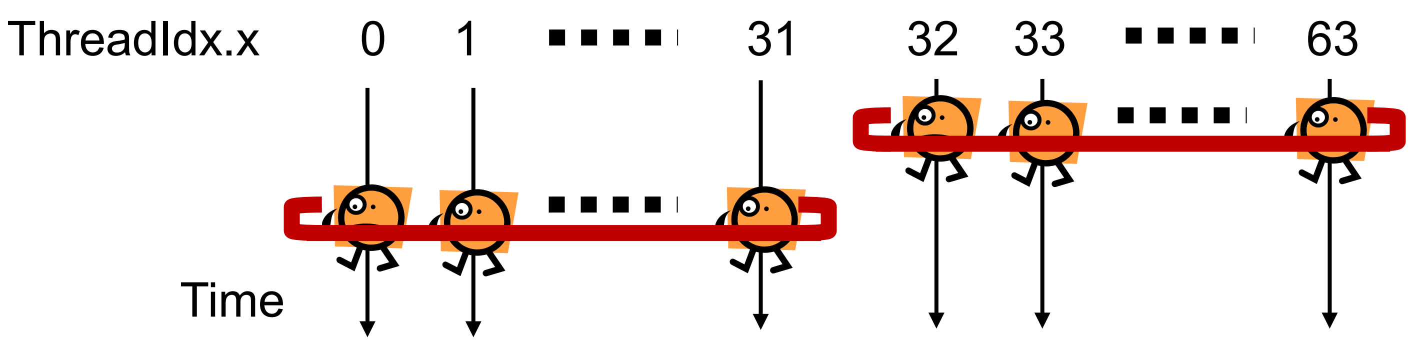
- 1 thread blocks (or more) run on 1 SMX
  - → At least 56 blocks are needed to use all SMXs on P100
  - → gridDim (gx*gy*gz) should be ≧56
- 1 thread (or more) run on a CUDA core
  - → At least 56*64=3584 threads in total are needed to use all CUDA cores on P100
  - → Total threads (gx*gy*gz * bx*by*bz) should be ≧3584
- 32 consective 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

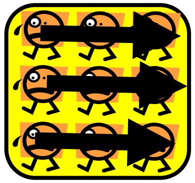


- Threads in a thread block are internally divided into “warp”, a group of contiguous 32 threads
- 32 threads in a warp always are executed synchronously
  - They execute the same instruction simultaneously
  - Only 1 program counter for 32 threads → GPU hardware is simplified
  - Actually 32 threads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

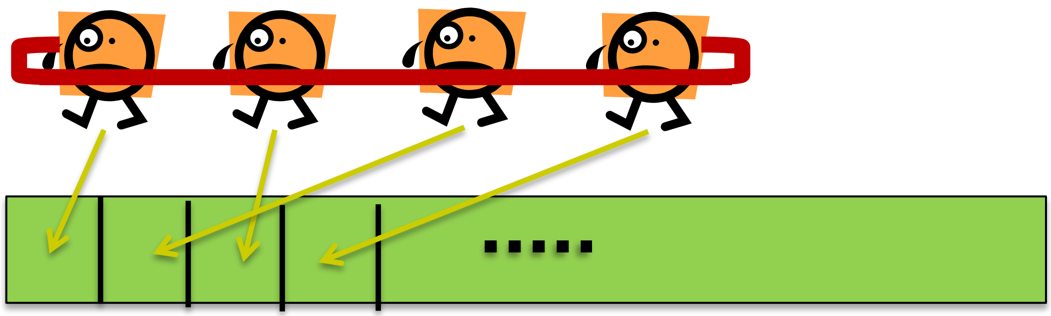


※ In multi-dimensional cases (blockDim.y>1 or blockDim.z>1), "neighborhood" is defined by x-dimension

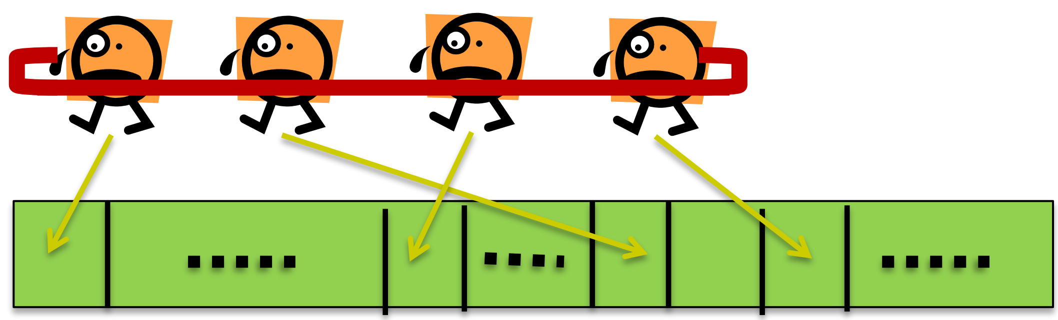
Coalesced Memory Access

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

Coalesced access
→ **Faster**



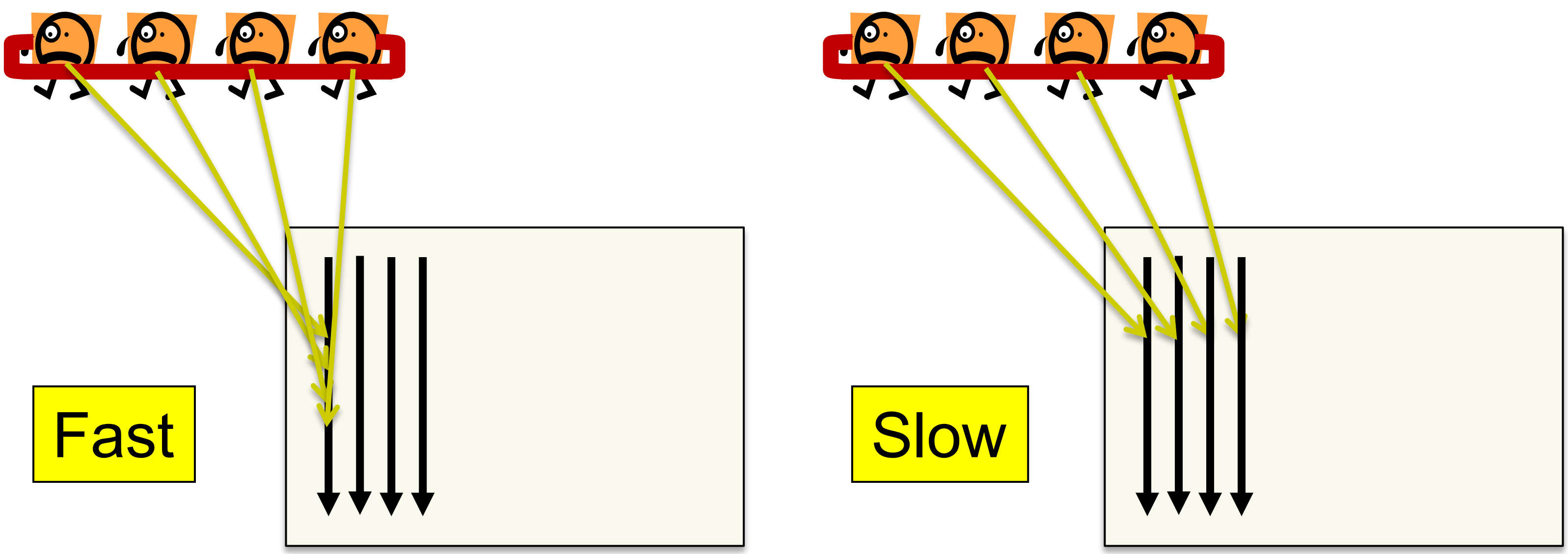
Non-coalesced access
→ **Slower**



Accesses in mm-cuda Sample

- mm-cuda: (x = row,y = col) → coalesced and fast
- mm-nc-cuda: (x = col, y = row) → non-coalesced and slow
  - /gs/hs1/tga-ppcomp/23/mm-nc-cuda

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

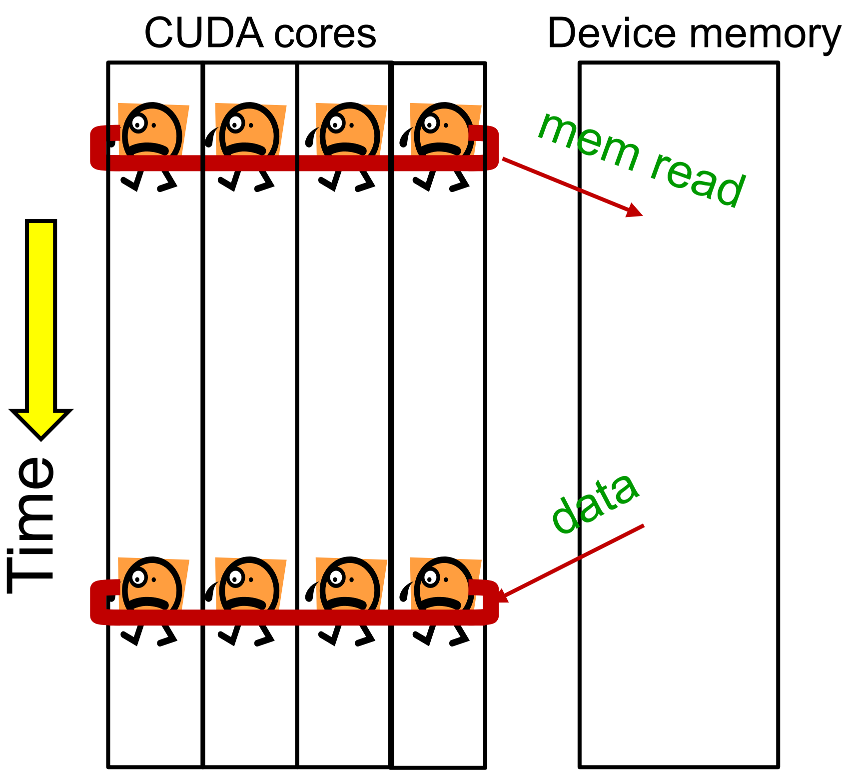


matrices in column-major format

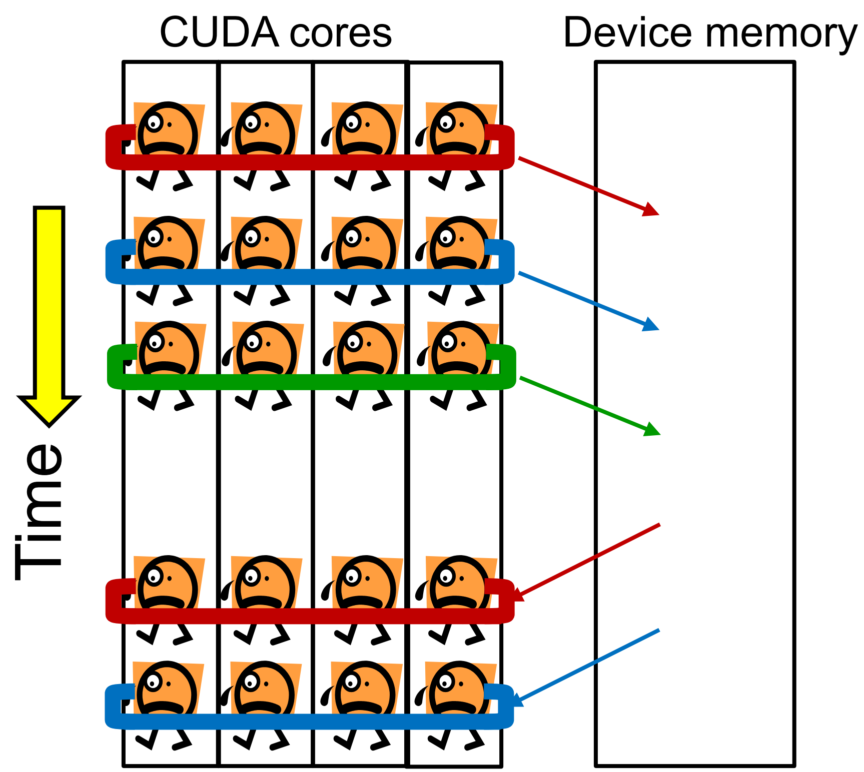
Why #threads >> #cores Works Well on GPUs?

- GPU supports very fast (~1 clock) context switches
  ➔ With many threads, memory access latency can be hidden

#threads == #cores



#threads > #cores

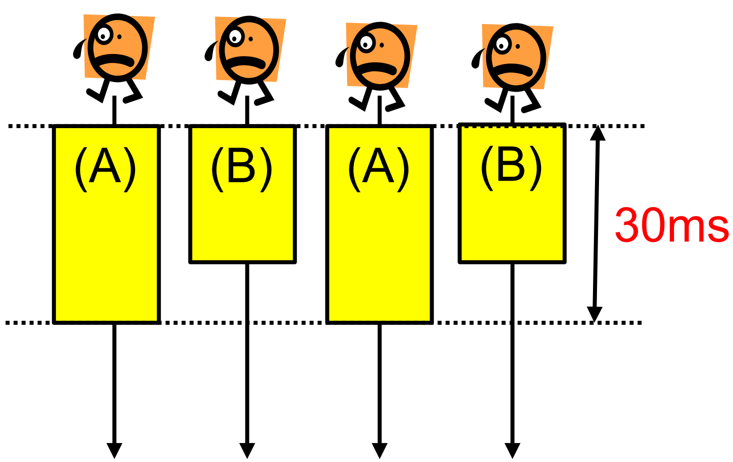


Considering Branches in Parallel Programs

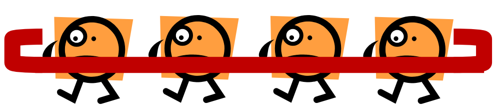
Consider this code. How long is execution time?

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

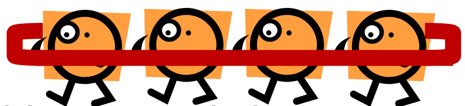
On CPU (OpenMP)

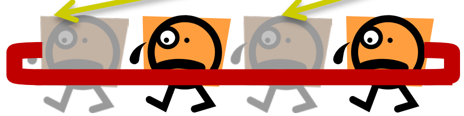
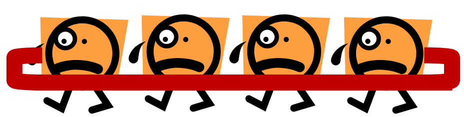


On GPU, threads in a warp must execute the same instruction. What happens?



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

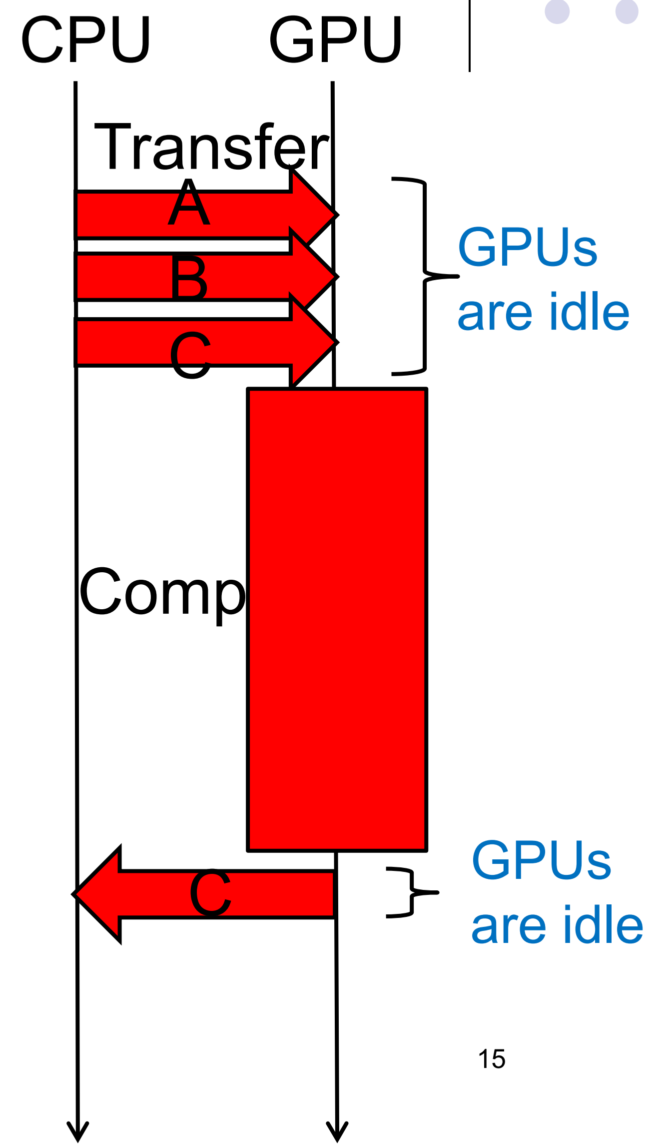
Considering Data Transfer Costs of mm Sample

- 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

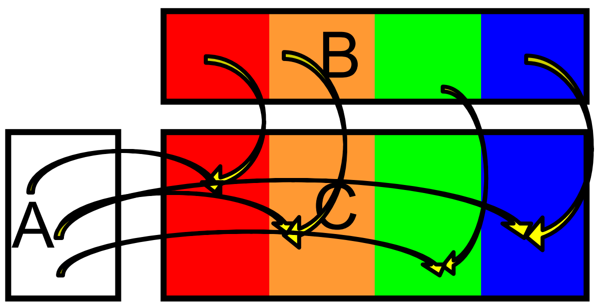
→ The faster sample is at /gs/hs1/tga-ppcomp/23/mm-str-cuda/



Split mm Computation (1)

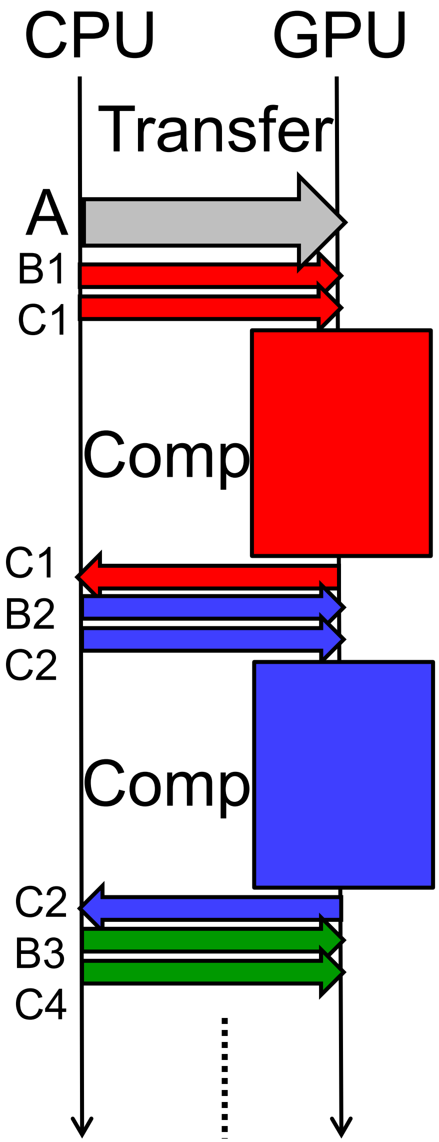
- Computation of "C ← A×B" is split by splitting B and C vertically
  - $C_1 \leftarrow A \times B_1, C_2 \leftarrow A \times B_2, \ldots, C_n \leftarrow A \times B_n$

  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

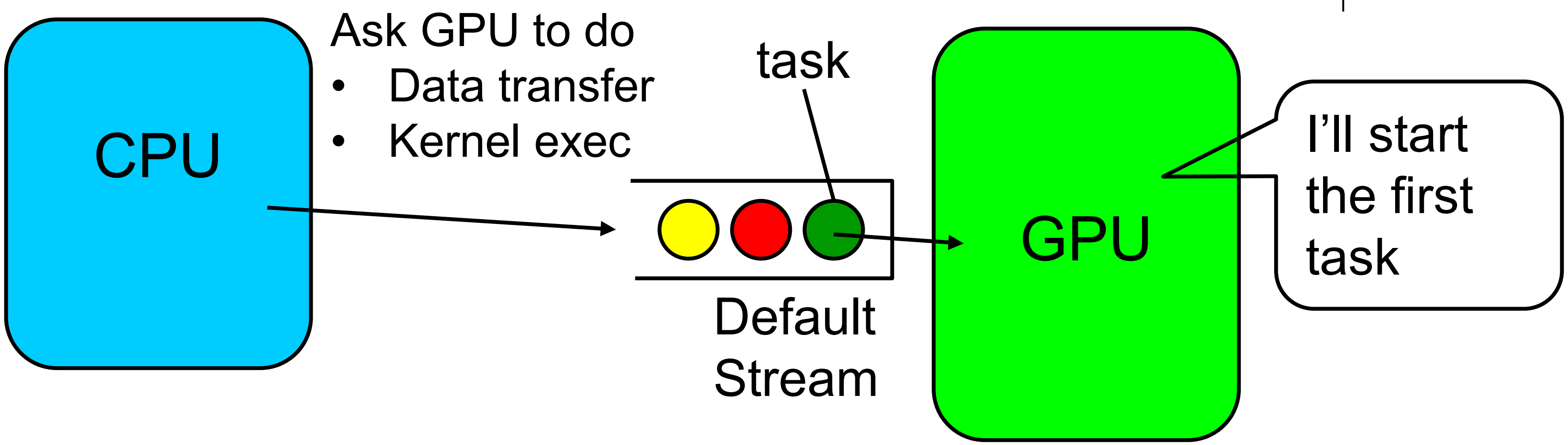
  (2) Compute $C_i \leftarrow A \times B_i$

  (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)

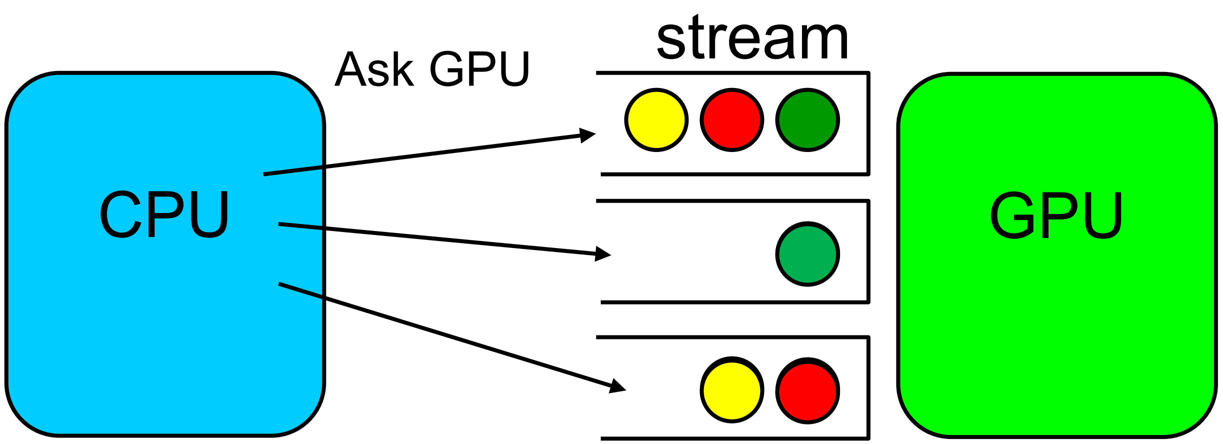


- A GPU is idle until asked to do something by CPU
- CPU asks the GPU to do one of followings (called tasks here)
  - Data transfer (Host → Device) or
  - GPU Kernel function execution or
  - Data transfer (Device → Host)
- Then the task is put on a FIFO queue, called *default stream*
- GPU takes a task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

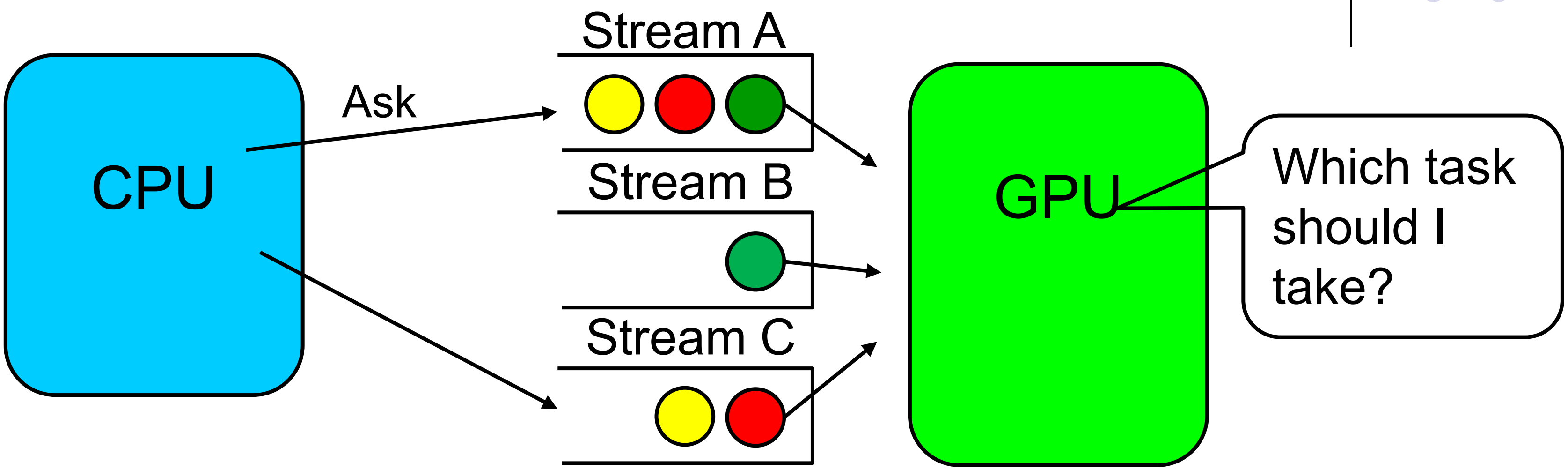
```
func<<<gs, bs, 0, str>>>( … );
// 3rd parameter is related to for "shared memory"
```

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

mm-str-cuda 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

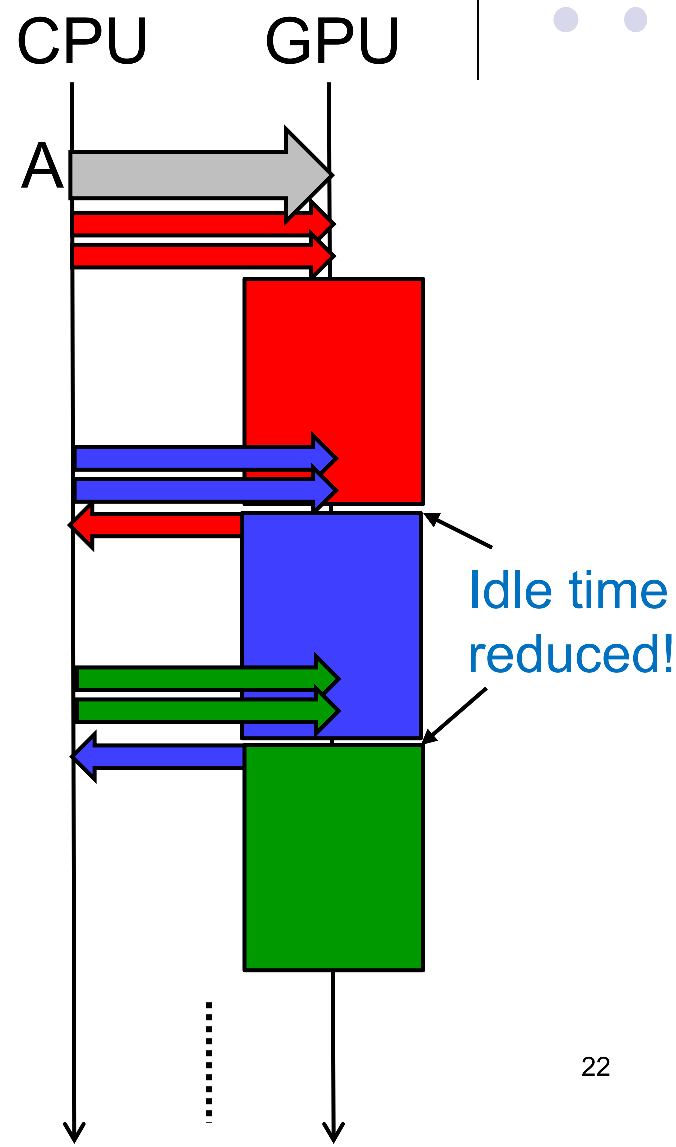
  ….
- Cn 🡐 A × Bn

→ We will see speed up since

(Total comp time + Total trans time)

is improved to roughly

max(Total comp time, Total trans time)



This is not a unique solution;
It is ok to use 2 or 3 streams repeatedly → we can save GPU memory and stream resources

cudaMallocHost() is used instead of malloc()
This speeds up cudaMemcpyAsync()

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
  - Google with “cudaMallocManaged”
- Using Tensor-core to accelerate deep learning
  - Only on V100 GPUs or later
  - Unfortunately, TSUBAME3 has older P100 ☹
- Using multiple GPUs towards petascale computation
  - MPI+CUDA, MPI+OpenACC
- More and more…

Assignments in GPU Part (Abstract)

Choose <u>one of </u>[G1]—[G3], and submit a report

Due date: May 25 (Thursday)

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

[G2] Evaluate speed of “mm-acc” or “mm-cuda” in detail

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

Next Class:

- Part 3: MPI Programming (1)
  - Introduction to distributed memory parallel programming

- Planned schedule
  - May 18: Part 3 (1)
  - May 22: Part 3 (2)
  - May 25: cancelled (休講) & Due for Part2 assignment
  - May 29: Part 3 (3)
  - June 1: Part 3 (4)