2019 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

Parallelization of mm Sample (related to [G2])

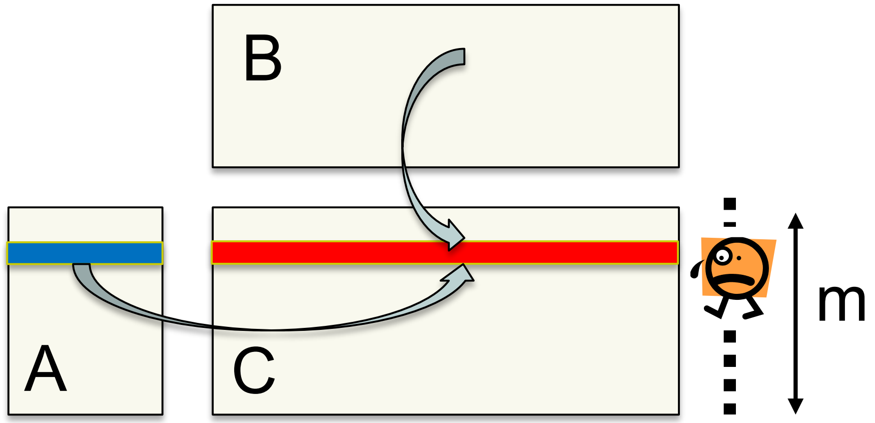
In mm, we can compute different C elements in parallel

CUDA (mm1-cuda)

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

We have seen that this is slower than OpenACC version ☹ -- Why?

- The number of threads (m) is still insufficient on GPUs
- If (1thread = 1element), we can use m*n threads

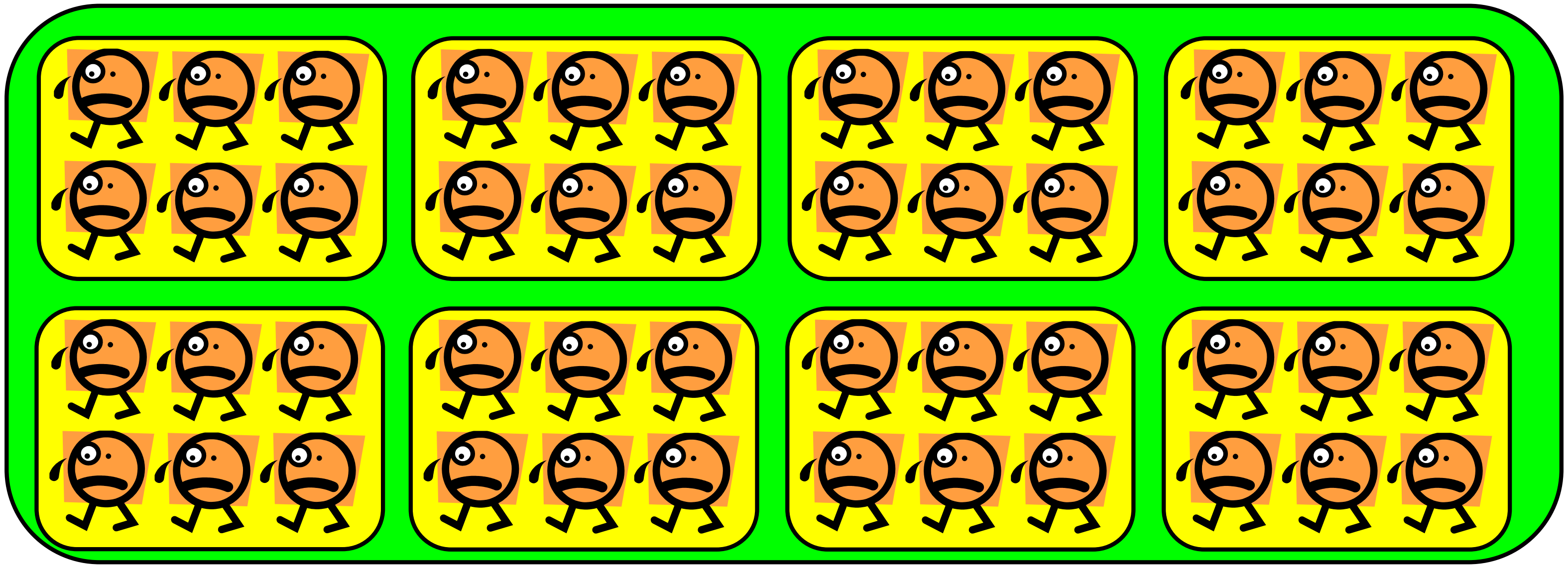


※ This is not the unique way

Creating Threads with 2D/3D IDs

- Now we want to make m*n (may be >1,000,000) threads
  - <<<(m*n)/BS, BS>>> is ok, but coding is bothersome
- On CUDA, gridDim and blockDim may have "dim3" type, 3D vector structure with x, y, z fields

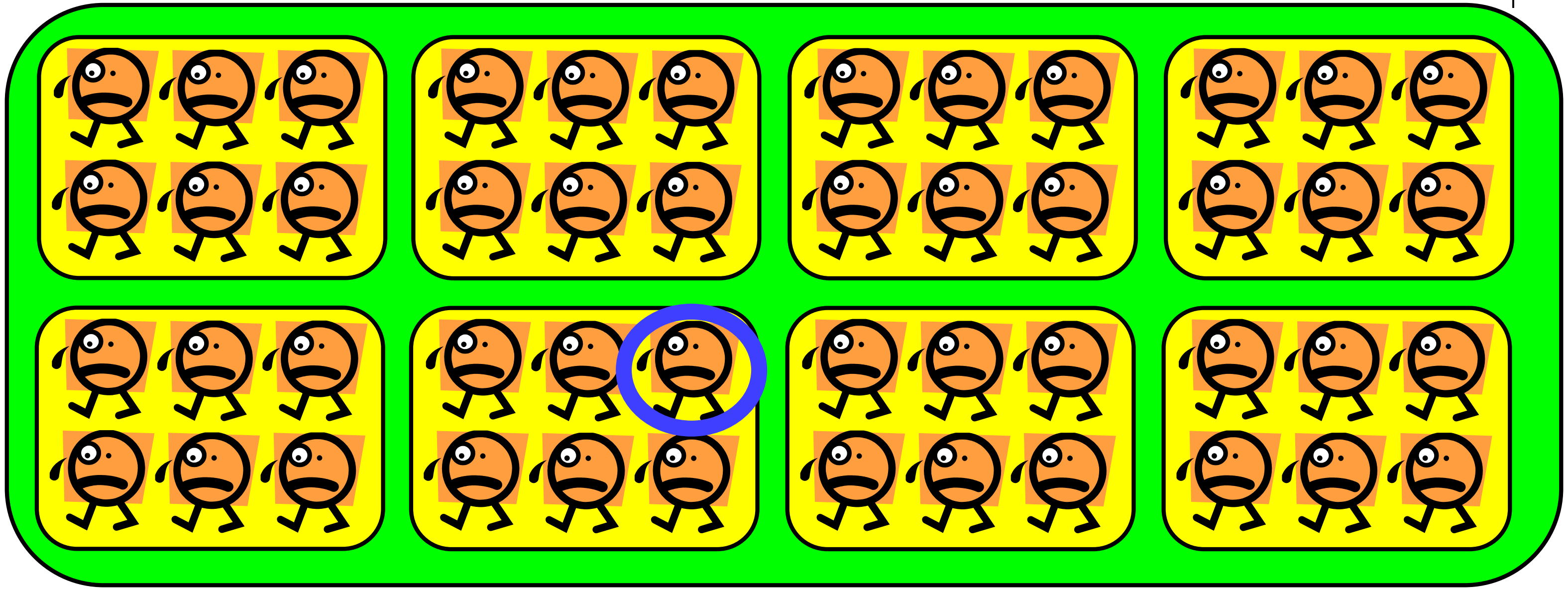
cf) func <<< dim3(4,2,1), dim3(3,2,1) >>> (); → 48 threads



※ This example is the case of 2D (Z dimensions are 1)

Thread IDs in multi-dimensional cases

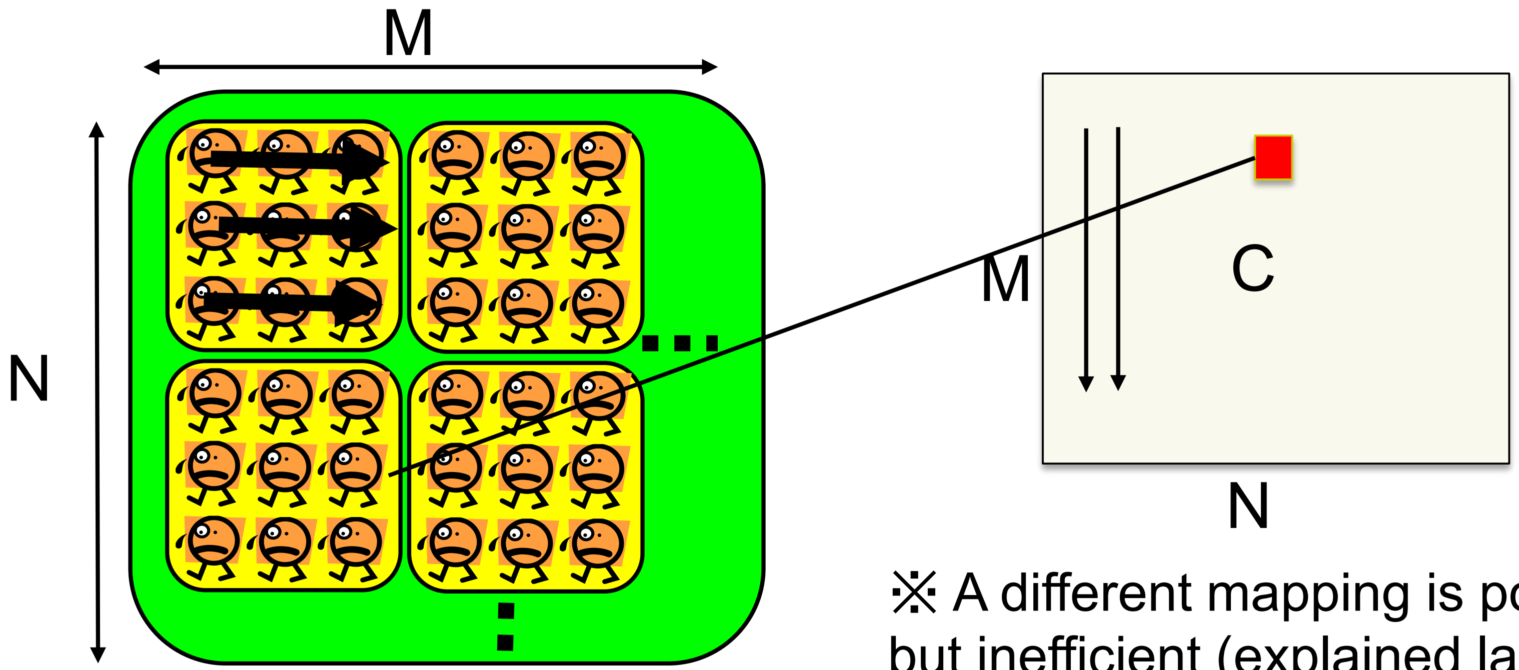
In the case of func <<< dim3(4,2,1), dim3(3,2,1) >>> ();



- For every thread,
  gridDim.x=4, gridDim.y=2, gridDim.z=1
  blockDim.x=3, blockDim.y=2, blockDim.z=1
- For the thread with blue mark,
  blockIdx.x=1, blockIdx.y=1, blockIdx.z=0
  threadIdx.x=2, threadIdx.y=0, threadIdx.z=0

Threads in mm2-cuda Sample

- The total number of threads are m*n
- How do we determine gridDim, blockDim?
  - <<<m, n>>> does not work for constraints explained later
- Here, we use fixed blockDim (x=16, y=16 → 256 threads per block)
  - Then gridDim is computed from M, N
- x is mapped to column index, y is mapped to row index (※)



※ A different mapping is possible, but inefficient (explained later)

Code in mm2-cuda

$$\texttt{matmul\_kernel<<<}\overbrace{\texttt{dim3(m / BS, n / BS, 1)}}^{\text{gridDim}}\texttt{, }\overbrace{\texttt{dim3(BS, BS, 1)}}^{\text{blockDim}}\texttt{>>>}$$

```
matmul_kernel<<<dim3(m / BS, n / BS, 1), dim3(BS, BS, 1)>>>
        (DA, DB, DC, m, n, k);
```

BS=16 in this sample
Actually, we use rounding up

In matmul_kernel function,

```
        :
    j = blockIdx.y * blockDim.y + threadIdx.y;
    i = blockIdx.x * blockDim.x + threadIdx.x;
        :   This thread computes C_ij ← Only 1 for-loop
```

This thread computes C_{ij} ← Only 1 for-loop

Comparing speed of mm-acc, mm1-cuda, mm2-cuda

m=n =k	mm-acc	mm1-cuda (thread =row)	mm1-cuda (thread =col)	mm2-cuda (x = row, y = col)	mm2-cuda (x = col, y = row)
1000	143 (Gflops)	14 (Gflops)	14 (Gflops)	185 (Gflops)	105 (Gflops)
2000	173	27	24	232	?
4000	164	50	29	246	?
6000	138	70	31	240	?
8000	137	85	32	243	?

- Measured with a P100 GPU on TSUBAME3
- CUDA version is compiled with –arch=sm_60 option
- Data transfer costs are included

Please make this version and compare (optional in [G2])

CUDA Rules on Number of Threads

func<<<gs, bs>>> (...); is interpreted as
func<<<dim3(gs,1,1), dim3(bs,1,1)>>> (...);

$$\text{func<<<}\overbrace{\text{dim3(gx, gy, gz)}}^{\text{gridDim}}\text{, }\overbrace{\text{dim3(bx, by, bz)}}^{\text{blockDim}}\text{>>> (...);}$$

≦ 2^{31}-1

≦ 65535

≦ 1024

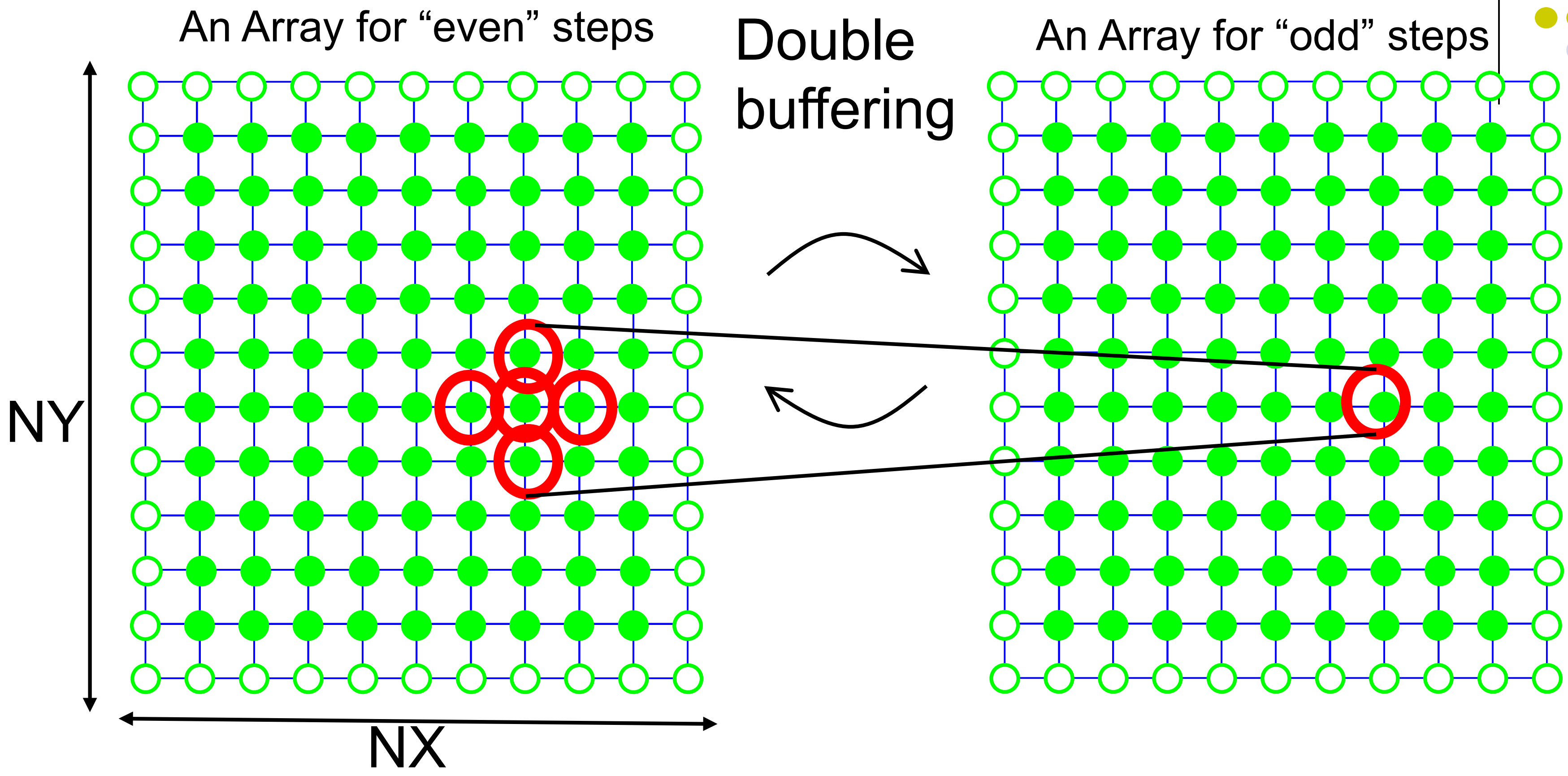
≦64

Also, bx*by*bz must be ≦1024

BlockDim has severe limitation ☹

Cf) <<<m, n>>> causes an error if n>1024 ☹

Discussion on parallel “diffusion” (related to [G1])



- Speed is improved by assignment: 1 Thread = 1 Point (optional)

Considering gridDim/blockDim

- Points [1, NX-1) × [1, NY-1), excluded boundary, should be computed.

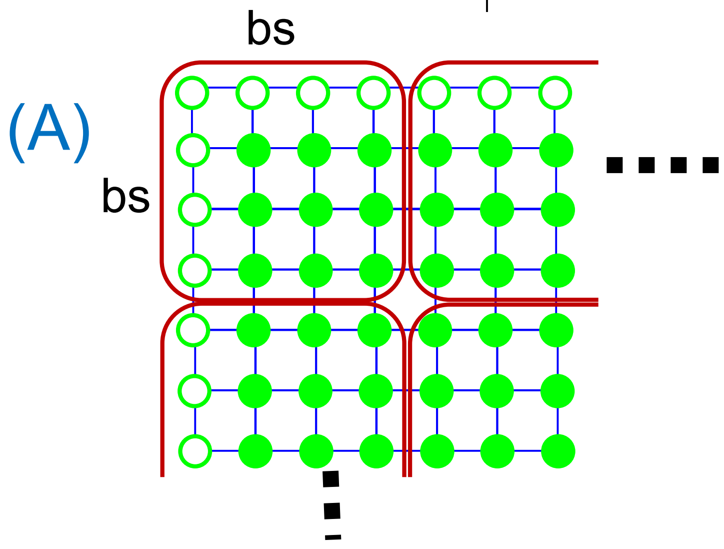
  There are choices:

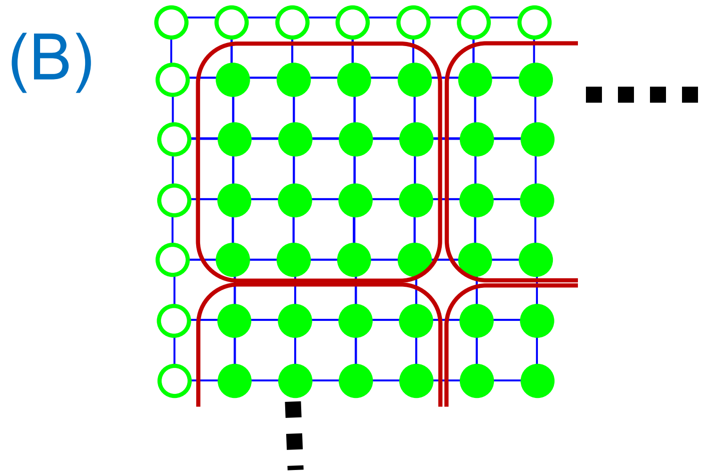
  (A) Create NX x NY threads

  (B) Create (NX-2) x (NY-2) threads
- For gridDim/blockDim, using "dim3" type would be a good idea

```
int BS =16
…<<< dim3(NX/BS, NY/BS, 1),
dim3(BS,BS,1)>>>…
```

  - Actually, we need rounding up and excluding extra threads
  - "mm2-cuda" sample is a hint
- Again, <<<NX, NY>>> causes error
  - BS must be 1024 or less

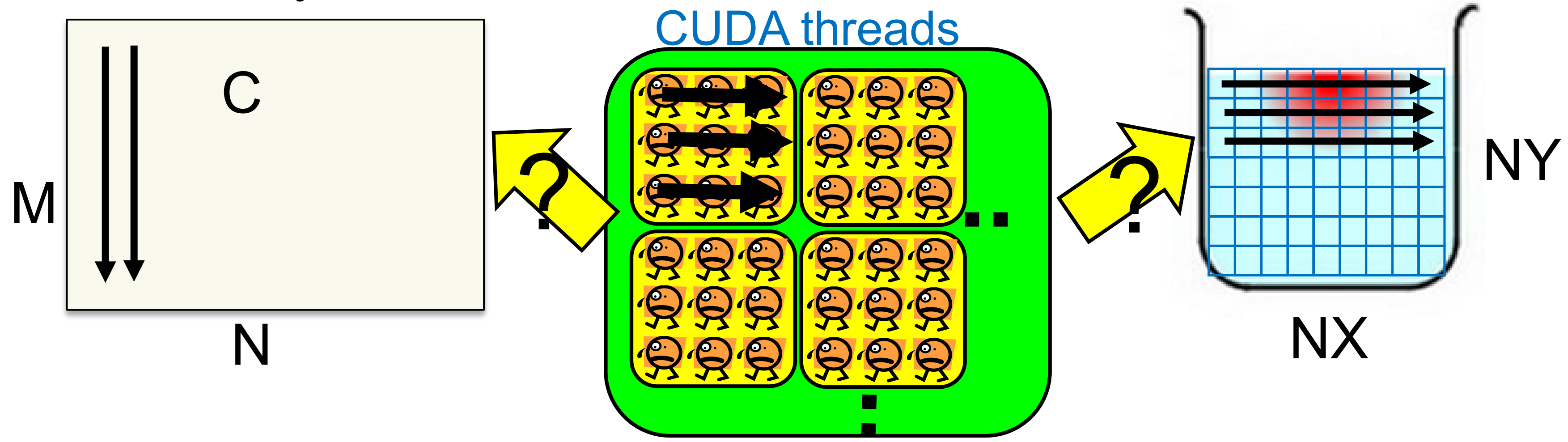




Mapping between Threads and Data

mm
Matrices has
column-major format

diffusion:
2D array has
row-major format



```
j = blockIdx.y * blockDim.y +
threadIdx.y;
i = blockIdx.x * blockDim.x +
threadIdx.x;
 : This thread computes Cij
```

```
y = blockIdx.y * blockDim.y +
threadIdx.y;
x = blockIdx.x * blockDim.x +
threadIdx.x;
 : This thread computes[y][x]
```

[Q] What if the dimensions are exchanged?

Discussions on CUDA Speed

- How should block-size determined?

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

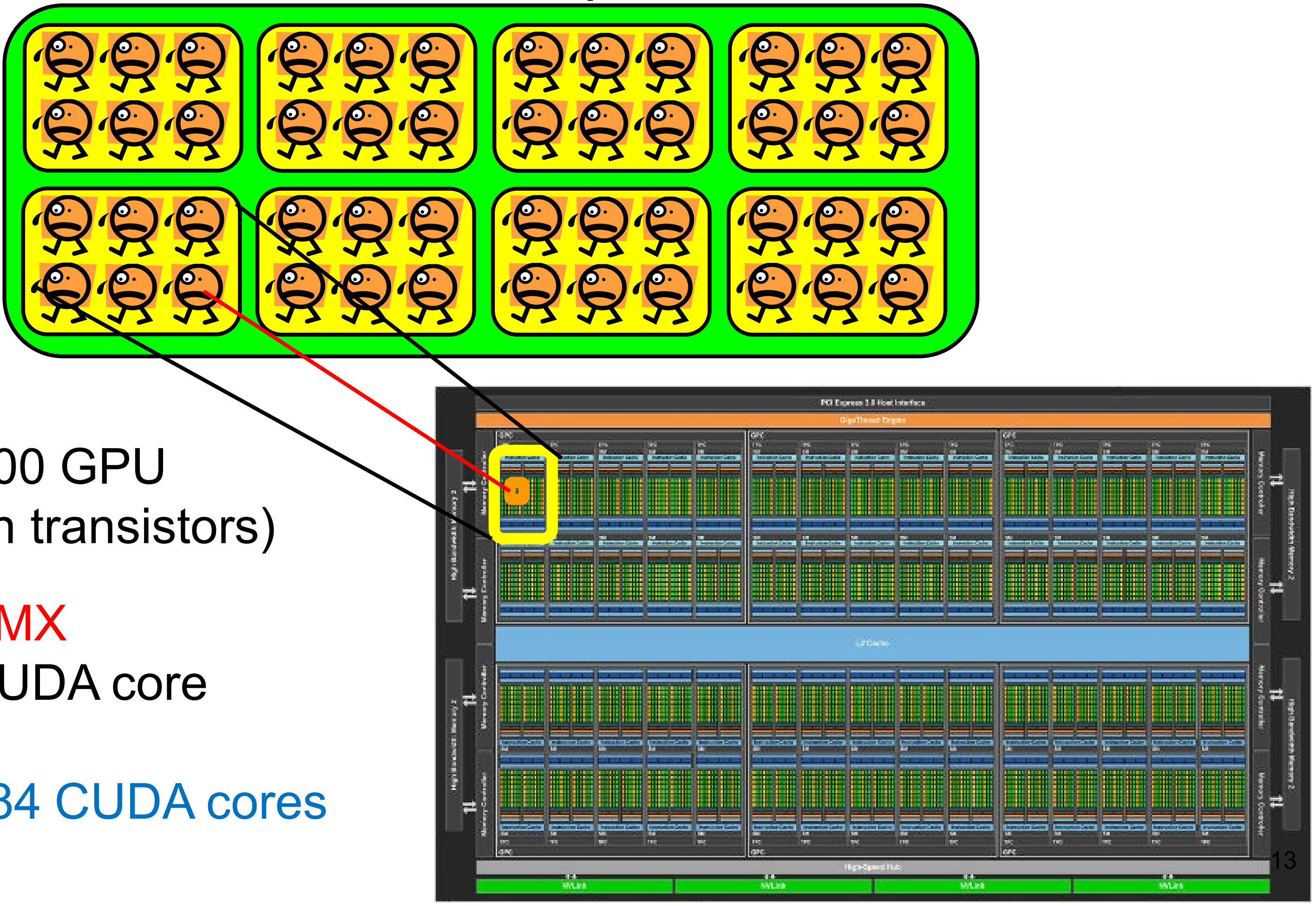
- How should each thread access memory?
  - In mm2-cuda, (x = row,y = col) and (x = col, y = row) shows different speed

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 SMX
1 SMX = 64 CUDA core

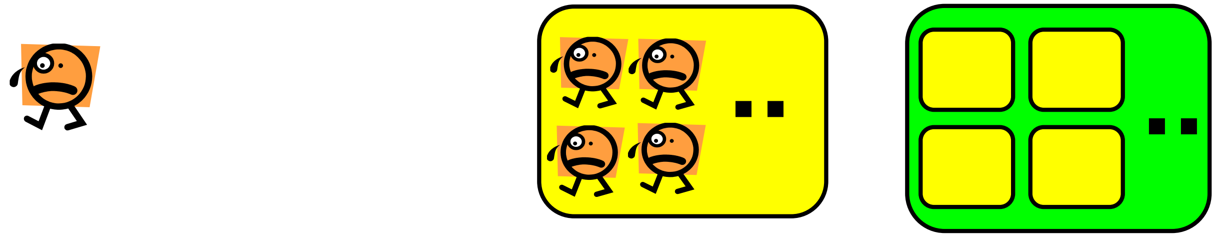
→ 1GPU=3,584 CUDA cores

Mapping between Threads and Cores

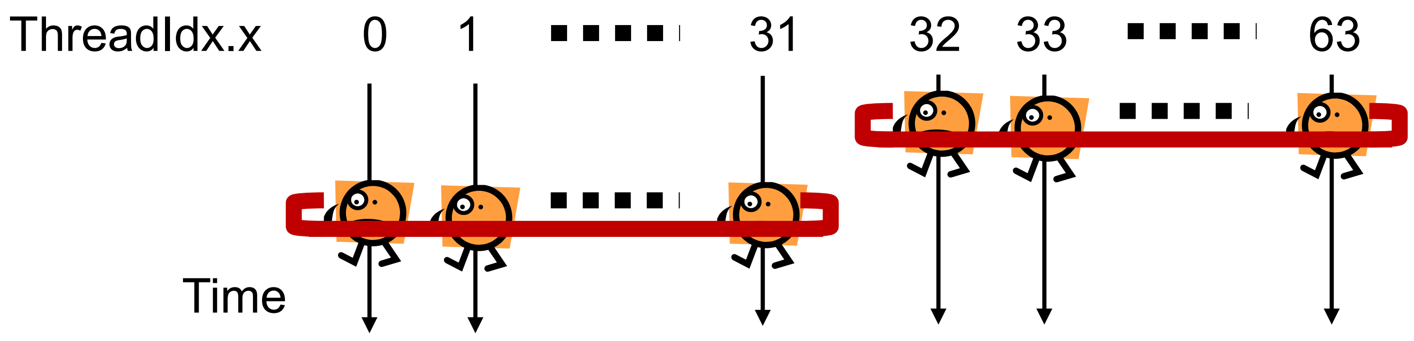
- 1 thread blocks (or more) run on 1 SMX
  - At least 56 blocks are needed to use all SMXs on P100
  - gridDim (gx*gy*gz) should be ≧56
- 1 thread (or more) run on a CUDA core
  - At least 56*64=3584 threads in total are needed to use all CUDA cores on P100
  - Total threads (gx*gy*gz * bx*by*bz) should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

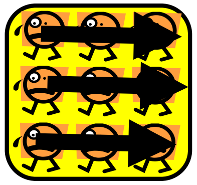


- Threads in a thread block are internally divided into "warp", a group of contiguous 32 threads
- 32 threads in a warp always are executed synchronously
  - They execute the same instruction simultaneously
  - There is only one program counter for 32 threads! → Structure of a GPU core is simplified



Observations due to Warps

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

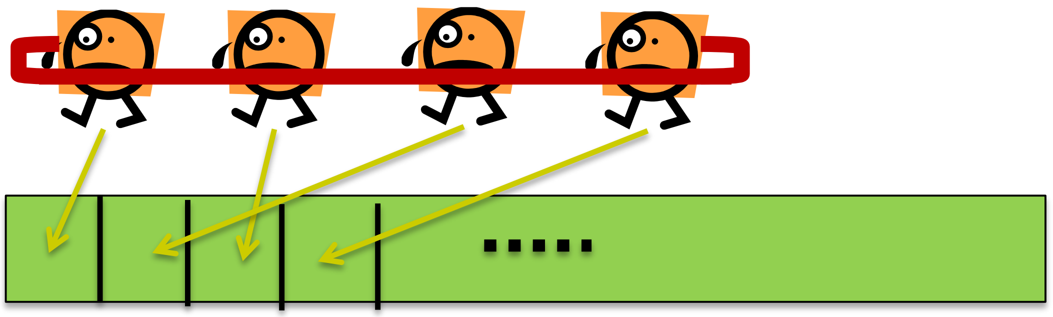


※ In multi-dimensional cases (blockDim.y>1 or blockDim.z>1), “neighborhood” is defined by x-dimension

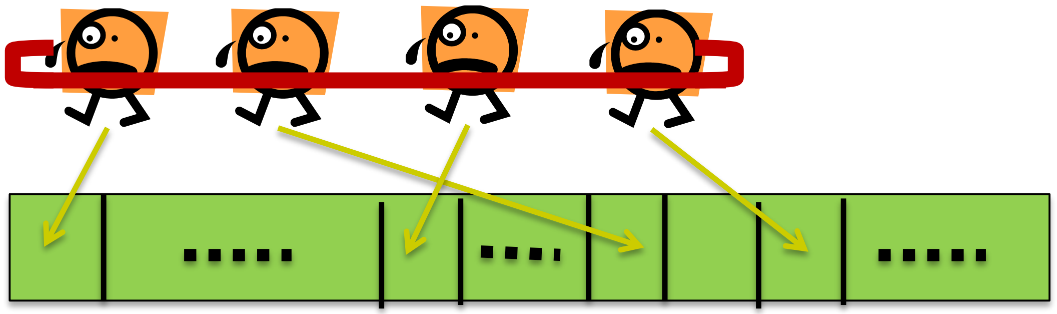
Coalesced Access

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

Coalesced access
→ **Faster**



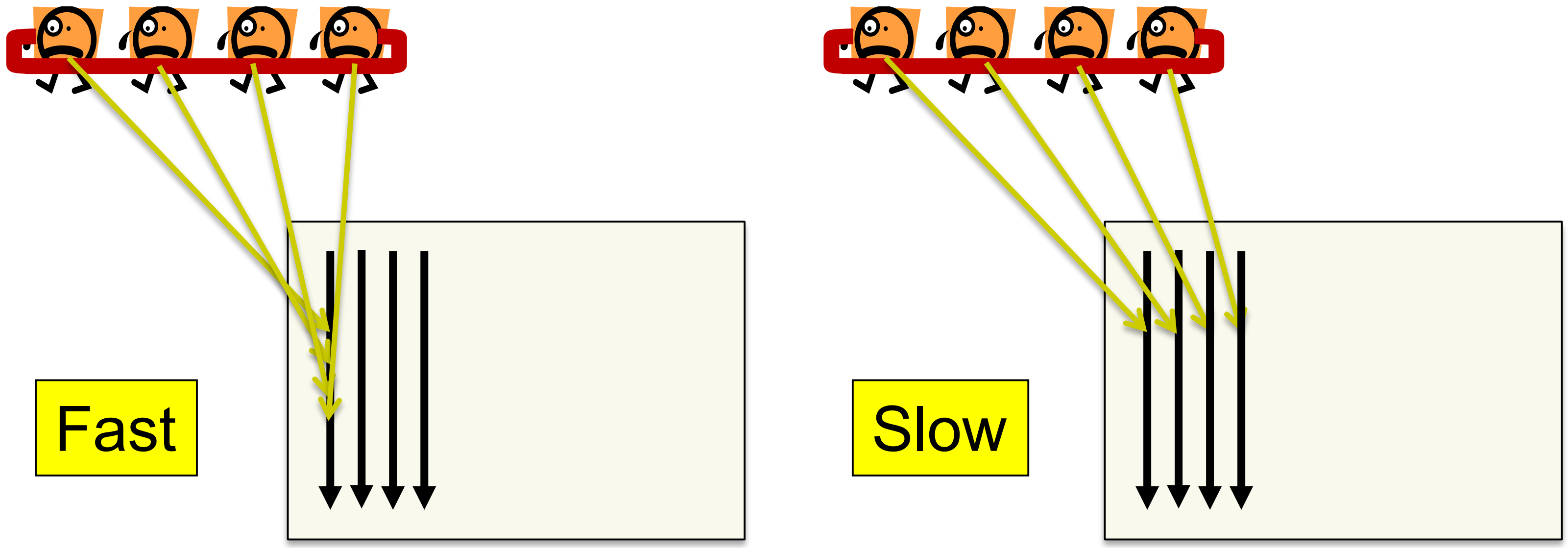
Non-coalesced access
→ **Slower**



Accesses in mm2-cuda Sample

- In mm2-cuda,
  - (x = row,y = col) → coalesced and fast
  - (x = col, y = row) → non-coalesced and slow

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



matrices in column-major format

More Things to Study

- Overlapping data transfer and computation by using cudaMemcpyAsync()
- Performance impact by divergent branch
- 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…

Official Documents

CUDA

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

OpenACC

- https://www.openacc.org
  - → Resources
  - → Spec

We Have Learned

- Part 1: Shared memory parallel programming with OpenMP
- Part 2: Distributed memory parallel programming with MPI
- Part 3: GPU programming with OpenACC and CUDA

Many common strategies for speed-up

- To understand source of bottleneck
- Reducing computation and communication
- Overlapping computation and communication
- To understand property of architecture

Assignments in GPU Part (Abstract)

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

Due date: June 17 (Monday)

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

[G2] Evaluate speed of “mm-acc” or “mm-cuda” (mm1-cuda and mm2-cuda) in detail

[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

Information

Lecture

- Slides are uploaded in OCW
  - www.ocw.titech.ac.jp → search “2019 practical parallel computing”
- Assignments information/submission site are in OCW-i
  - Login portal.titech.ac.jp → OCW/OCW-i
- Inquiry
  - ppcomp@el.gsic.titech.ac.jp
- Sample programs
  - Login TSUBAME, and see ~endo-t-ac/ppcomp/19/ directory

TSUBAME

- Official web including Users guide
  - www.t3.gsic.titech.ac.jp
- Your account information
  - Login portal.titech.ac.jp → TSUBAME portal