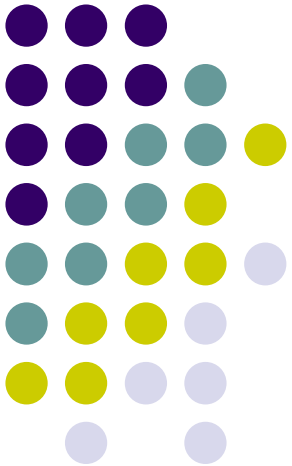
2017 Practical Parallel Computing (実践的並列コンピューティング) No. 14

GPU Programming with CUDA (3)

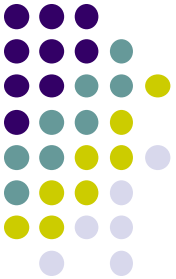


Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp

Specifying Number of Threads in OpenMP and CUDA



- OpenMP:

  cf) export OMP_NUM_THREADS=8

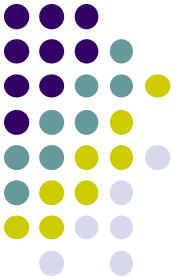
- CUDA:


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

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

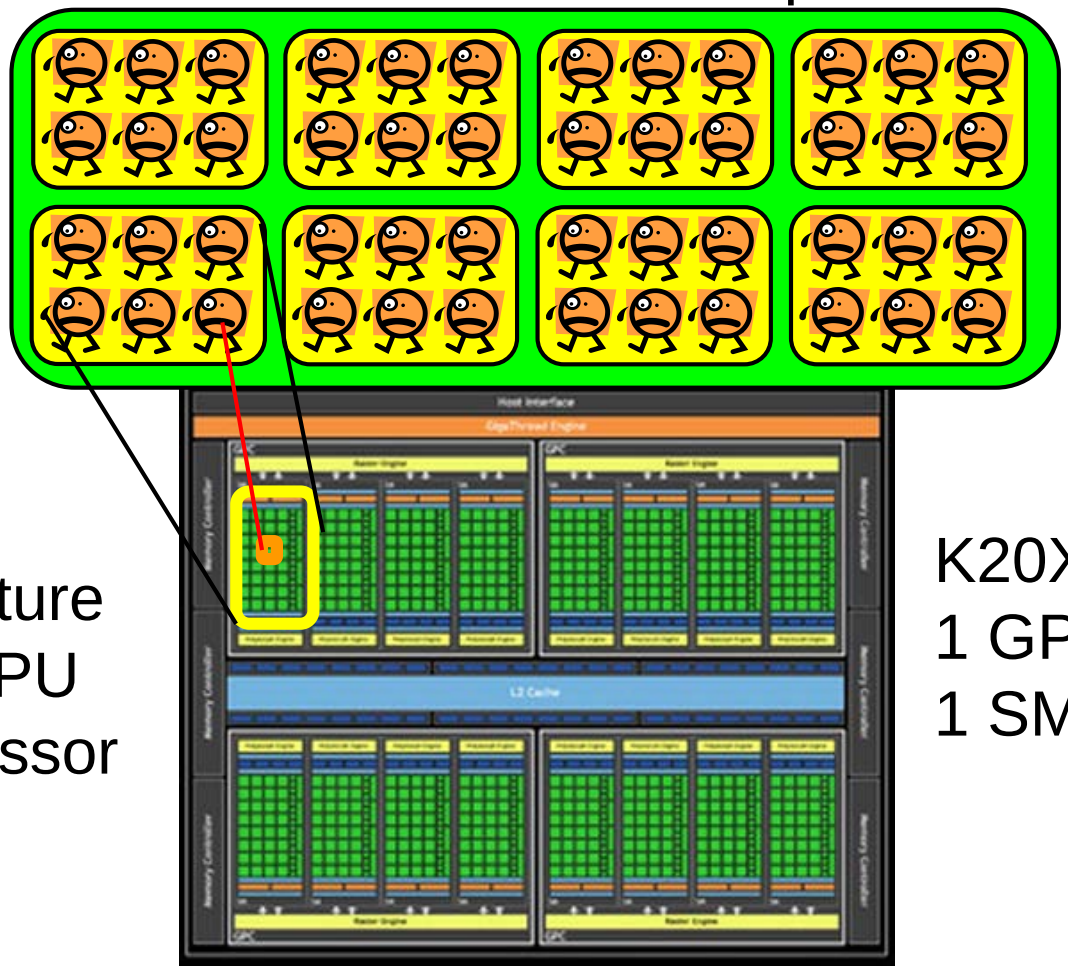
※ func<<<g, b>>>() is equivalent to
func<<<dim3(g,1,1),dim3(b,1,1)>>>()

Why Do We Have to Specify both gridDim and blockDim?



- and why did NVIDIA decide so?

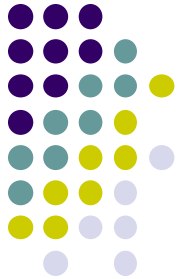
→ Hierarchical structure of GPU processor is considered



Structure of GPU Processor

K20X:
1 GPU = 14 SMX
1 SMX = 192 CUDA core

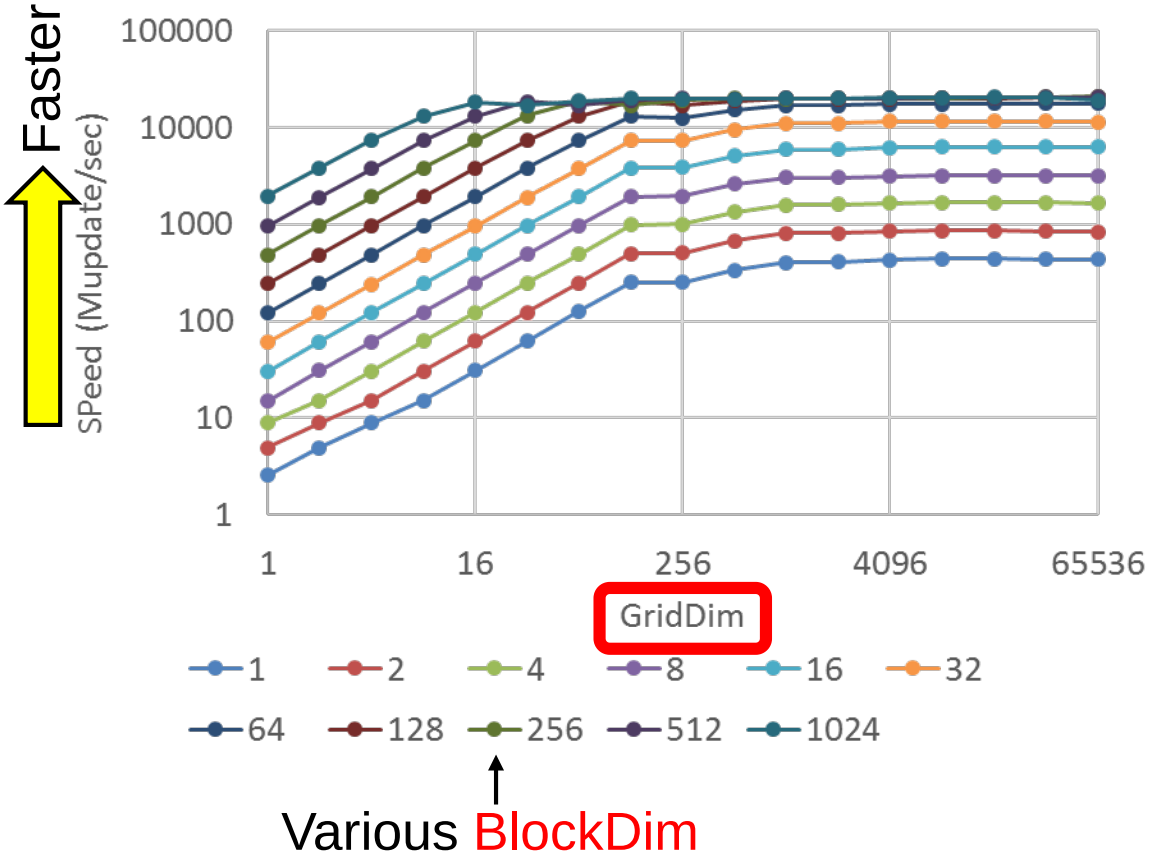
Mapping between Threads and Cores



- M (≧1) thread block run on 1 SM
  - → At least 14 blocks are needed to use all SMXs on K20X
  - → gridDim (gx*gy*gz) should be ≧14
- N (≧1) thread run on a CUDA core
  - → At least 14*192=2688 threads in total are needed to use all CUDA cores on K20X
  - → Total threads (gx*gy*gz * bx*by*bz) should be ≧2688
- 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

Performance with Variable Number of Threads

- inc_rr sample (modified version of inc_par)
  - Available at ~endo-t-ac/ppcomp/17/inc-rr/
  - Number of Threads ≦ array length. Cyclic distribution
  - A K20X on TSUBAME2.5 is used



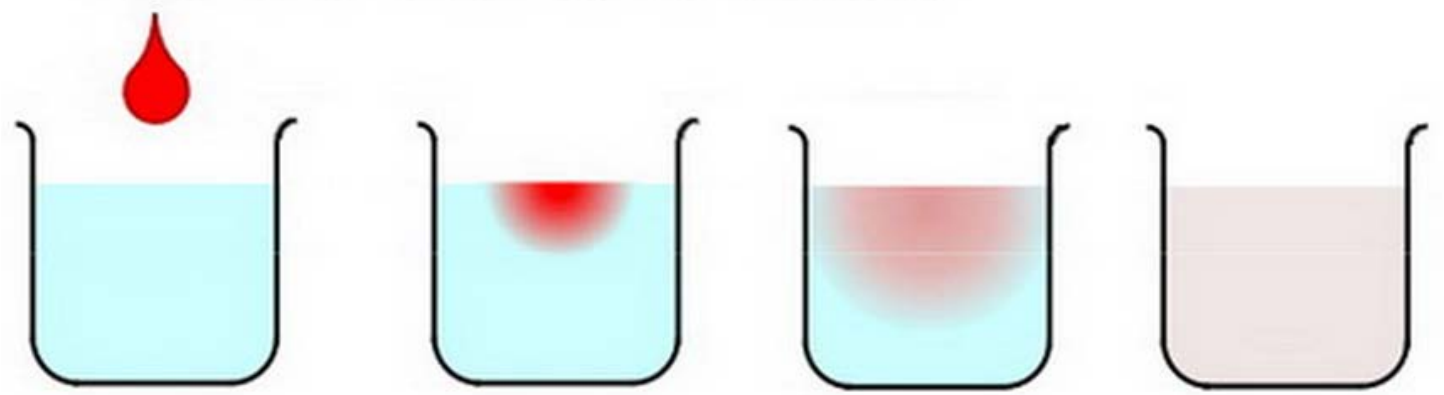
- Faster with larger GridDim, BlockDim
- In this program, speed reaches peak with GridDim × BlockDim ≧ 16384 and BlockDim ≧ 64

These numbers depend on both hardware and software

“diffusion” Sample Program (1) (Revisited, related to [G1])

An example of diffusion phenomena:

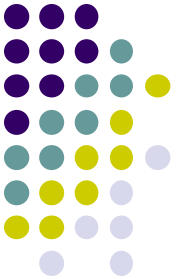
•Pour a drop of ink into a water glass



The ink spreads gradually, and finally the density becomes uniform (Figure by Prof. T. Aoki)

- Density of ink in each point vary according to time → Simulated by computers
- Stencil computation

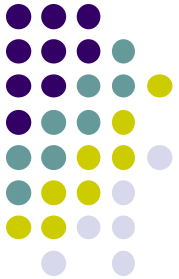
“diffusion” Sample Program (2) (Revisited)



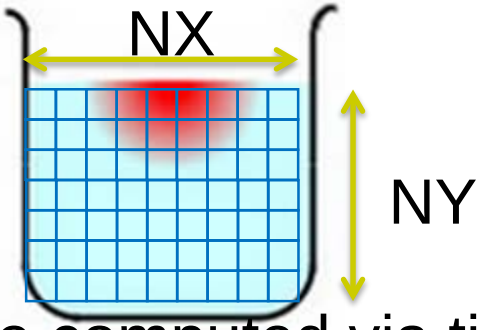
1 CPU version
Available at ~endo-t-ac/ppcomp/17/diffusion/

- Execution：./diffusion [nt]
- nt: Number of time steps
- nx, ny: Space grid size
  - nx=8192, ny=8192 (Fixed. See the code)
  - How can we make them variables? (See mm sample)
- Compute Complexity：O(nx × ny × nt)

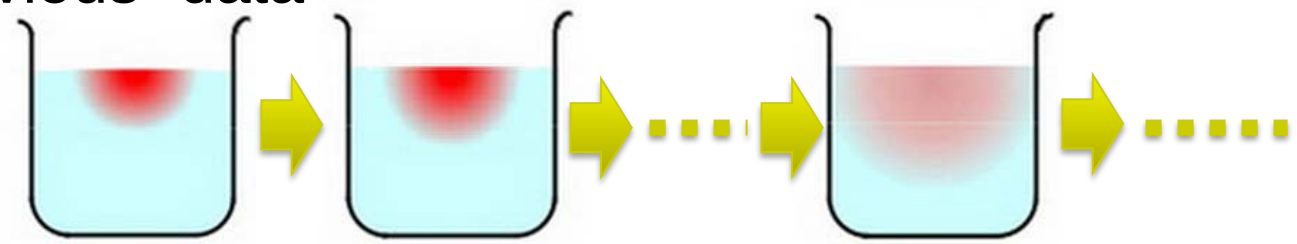
Data Structures in diffusion (Revisited)



- Space to be simulated are divided into grids, and expressed by arrays (2D in this sample)



- Array elements are computed via timestep, by using “previous” data

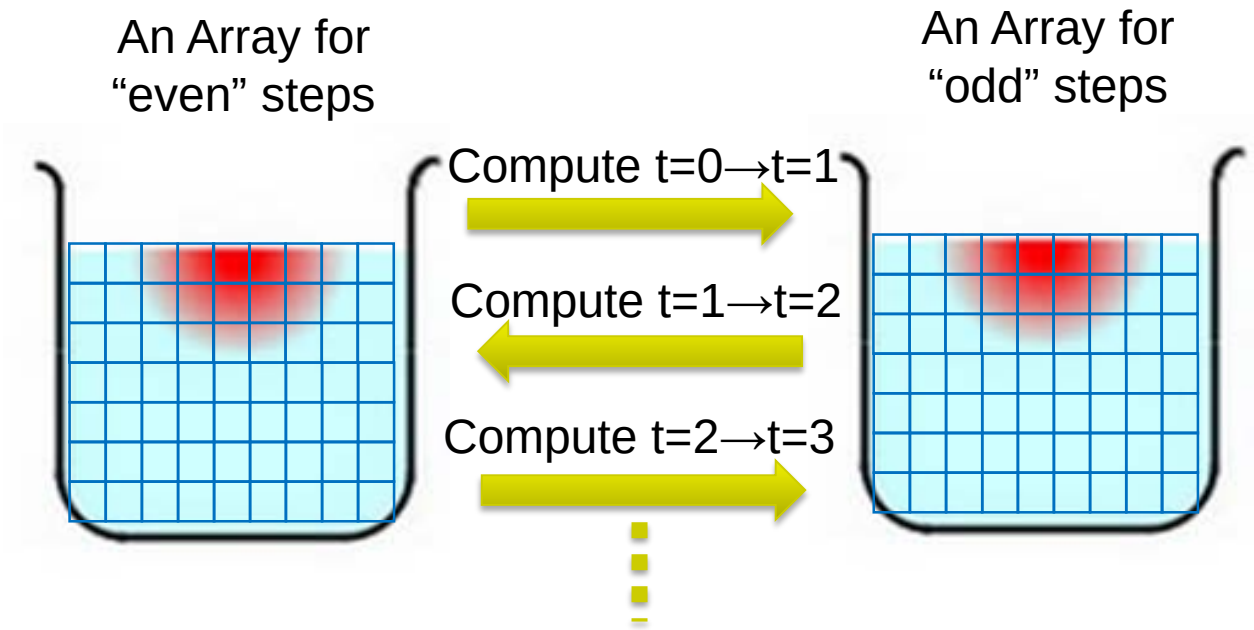


Time step t=0 t=1 t=20

Double Buffering Technique (Revisited)

- A simple way is to make arrays for all time steps, but it consumes too much memory!

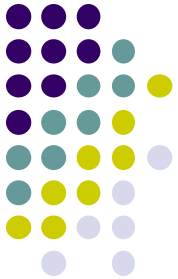
→ It is sufficient to have “current” array and “previous” array. “Double buffers” are used for many times



※ Sample program uses a global variables

float data[2][NY][NX];

How Do We Parallelize "diffusion" Sample?



Parallelization method with OpenMP:

[Algorithm] Parallelize spatial (Y or X) for-loop

- "1 parallel region = 1 time step" is easier
- Each thread computes its part in the space
- Time (T) for-loop cannot be parallelized, due to dependency

[Data] Data structure is same as sequential version

With CUDA:

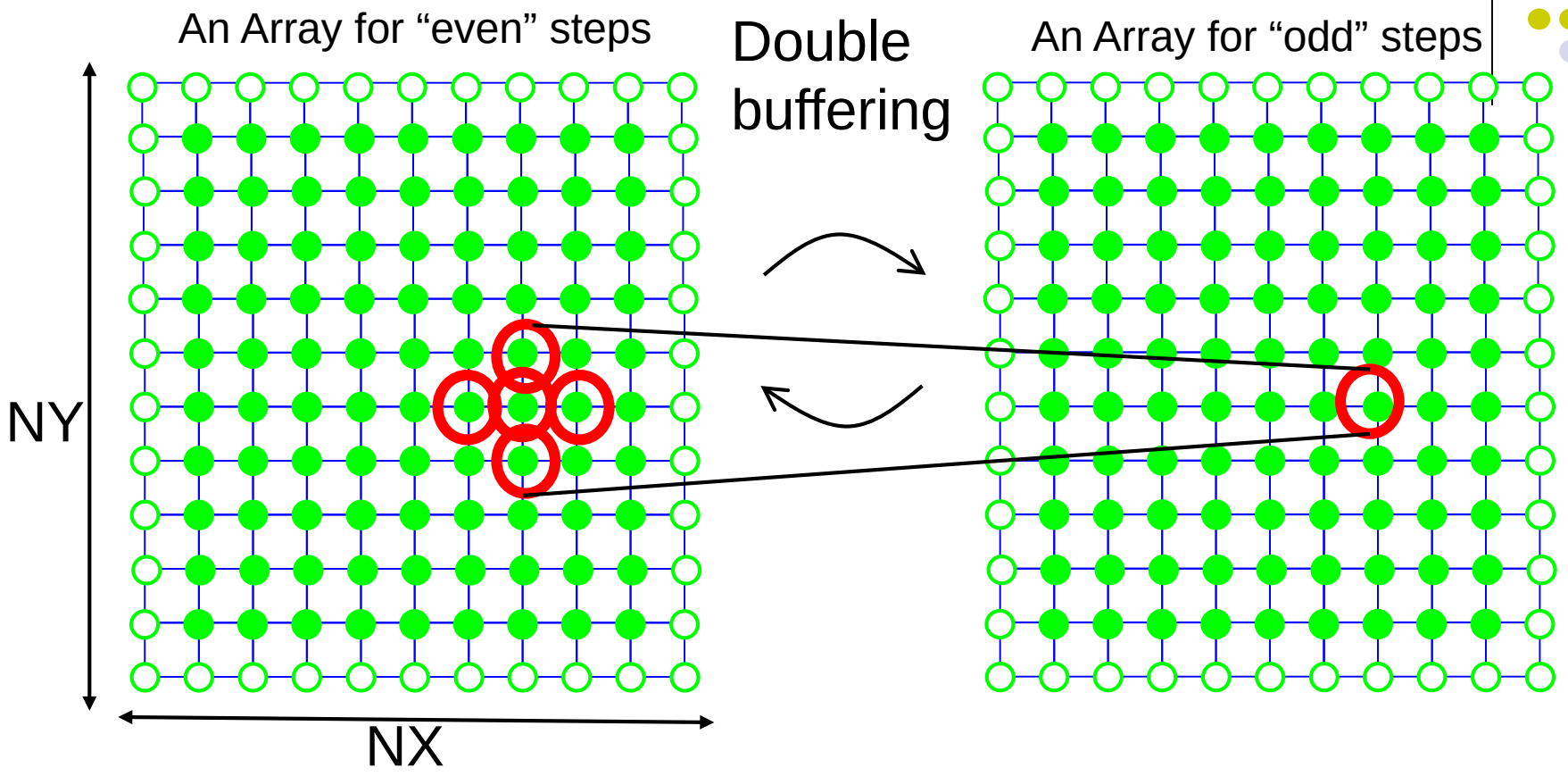
[Algorithm] Similar policy as OpenMP version

- "1 GPU kernel function call = 1 time step" is easier
- Unlike OpenMP, "1 thread = 1 point" policy is ok

[Data] Data structure is same as sequential version, but…

- When should we do cudaMemcpy?

Parallelize “diffusion” Sample



- In diffusion, computation of a new point requires 5 old points (5-point stencil)
- Points on boundary are exceptional. In this sample, no computation is done

Considering gridDim/blockDim

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

  There are choices:

  (A) Create NX x NY threads

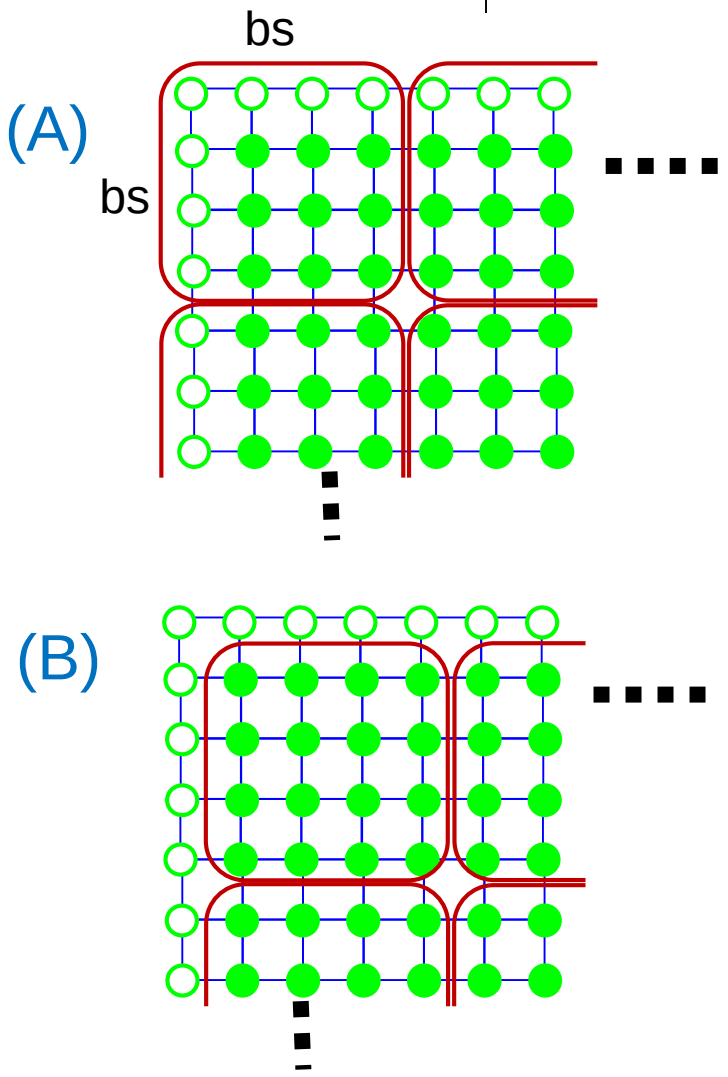
  (B) Create (NX-2) x (NY-2) threads

- 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
  - “mm-cuda” sample is a hint

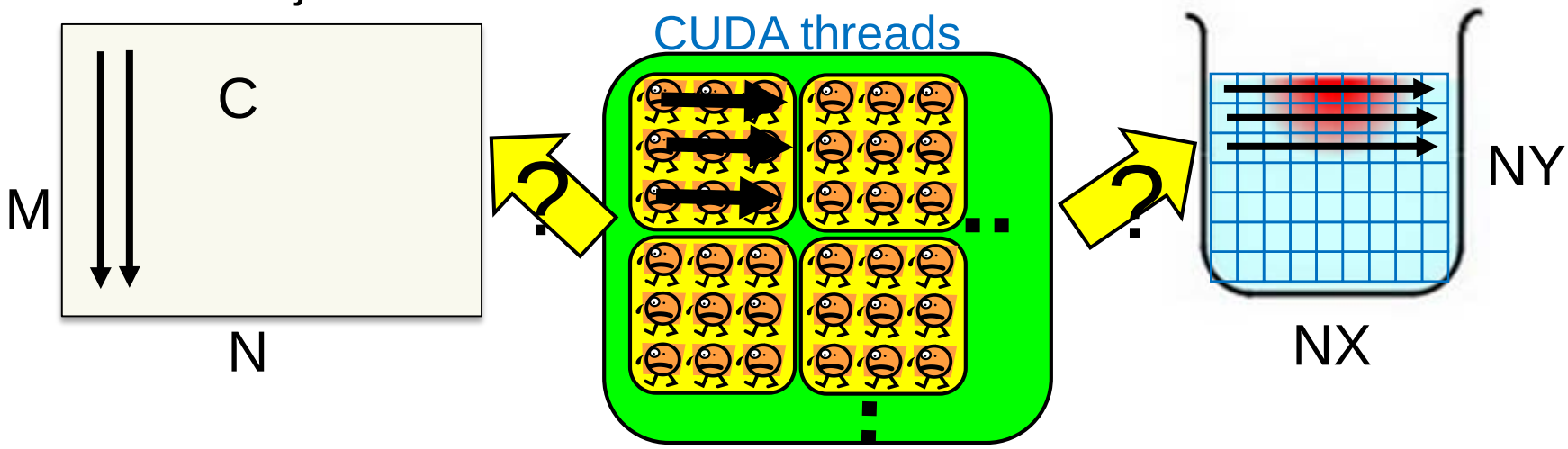
- On the other hand, <<<NX, NY>>> is not good ☹
  - See slides in previous class



Mapping between Threads and Data

mm-cuda:
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
```

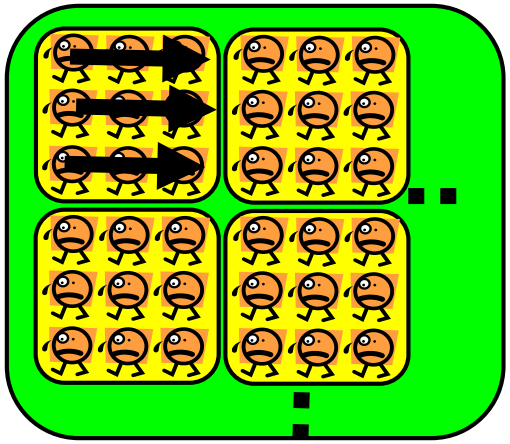
```
y = blockIdx.y * blockDim.y +
threadIdx.y;
x = blockIdx.x * blockDim.x +
threadIdx.x;
 : This thread computes[y][x]
```

[Q] What if the dimensions are exchanged?

Coalesced Access in CUDA

- Exchanging of dimensions does not affect computation complexity, but actual performance changes
- This is due to characteristics of CUDA:

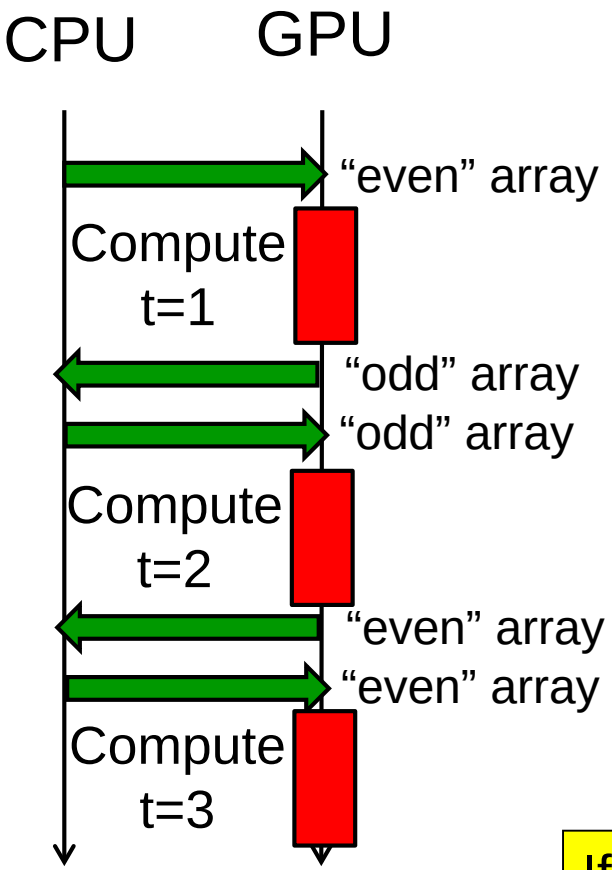
When “neighbor” threads in a thread block access “neighbor” address on memory (coalesced access), it is more efficient than non-coalesced access



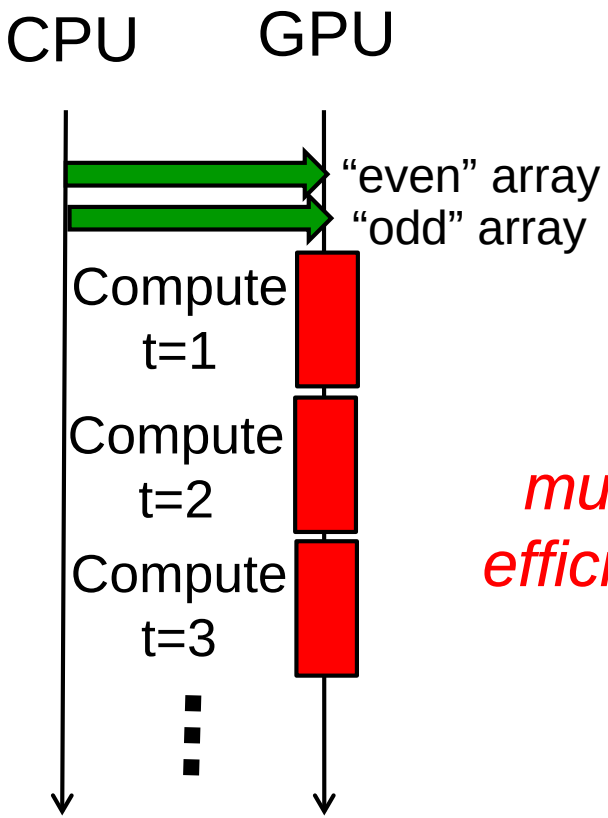
If threads are created with 2D/3D blockDim, “neighborhood” is defined by x-dimension

When Should We Do cudaMemcpy?

(1) before/after every GPU kernel function



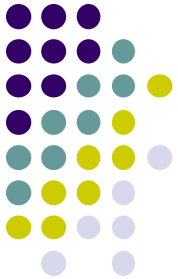
(2) before/after entire computations



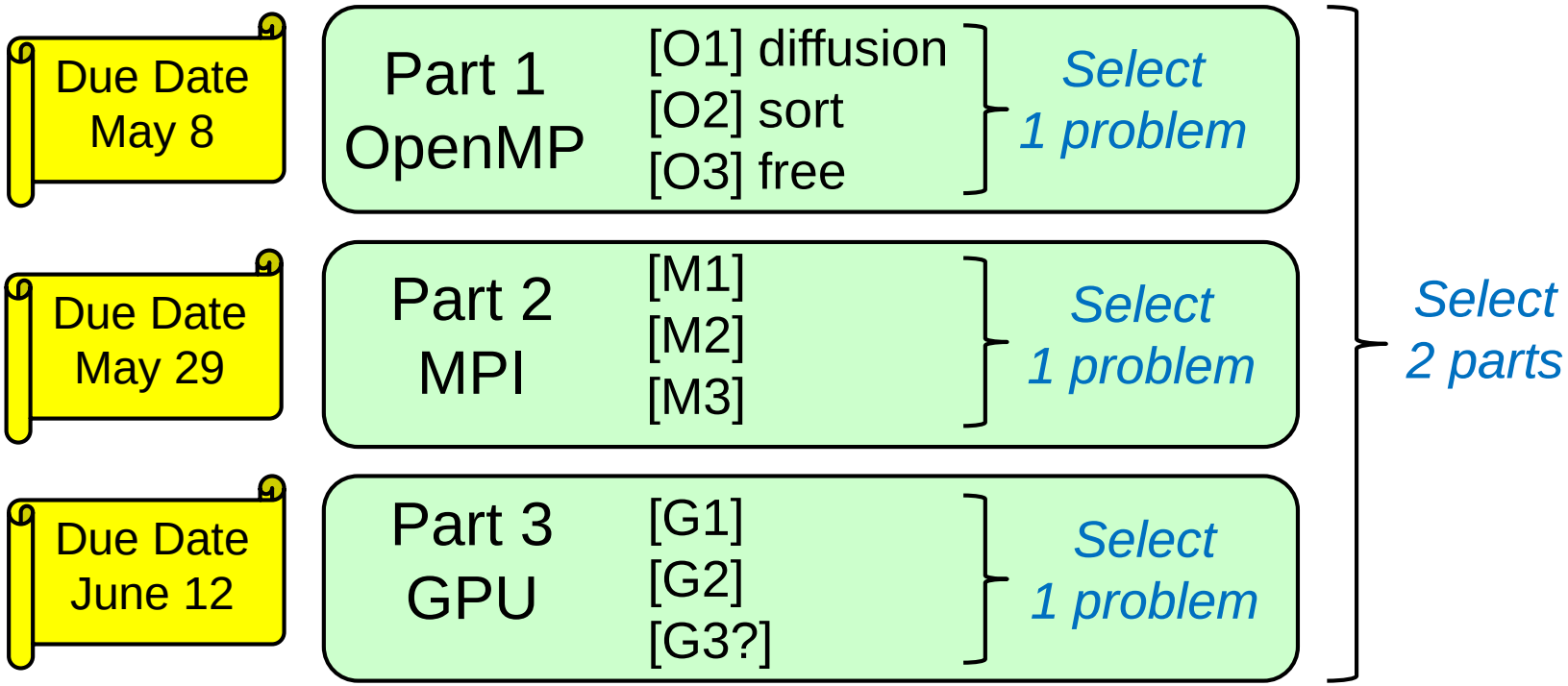
much efficient!

If you want to use intermediate state on CPU (such as file I/O), you need cudaMemcpy

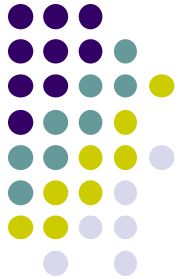
Assignments in this Course



- 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

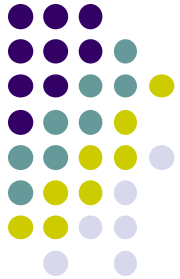
Due date: June 12 (Monday)

[G1] Parallelize “diffusion” sample program by CUDA.

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

[G3] (Freestyle) Parallelize *any* program by 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
- 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 (Final):

- GPU Programming (4)
  - Optimization techniques in GPU programming (cont'd)
  - TSUBAME tour