CUDA GPU 프로그래밍
무조건 따라하기

노재동 (2021년 7월 21일)

https://github.com/jdnoh

강의 시작 전에 확인합시다

- Download updated presentation file and source files at https://githut.com/jdnoh

- CUDA C on Linux (ubuntu 20.04) https://docs.nvidia.com/cuda
  `$ sudo apt install nvidia-driver-460` ⇐ GPU driver
  `$ sudo apt install nvidia-cuda-toolkit` ⇐ CUDA
  `$ sudo apt install nsight-systems` ⇐ profiling
  installation check: `$ nvcc -version; $ nvidia-smi`

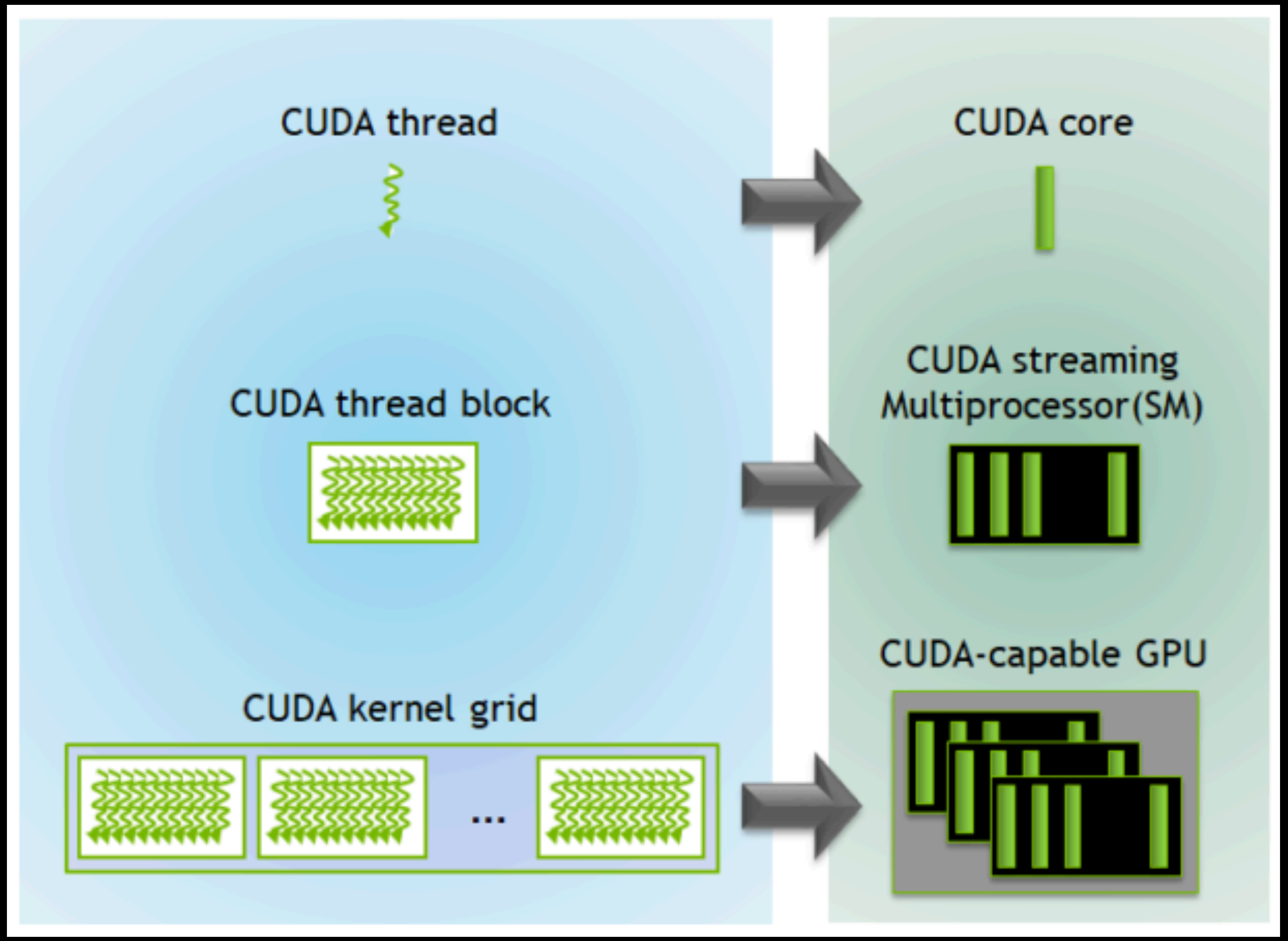
- Python (anaconda) https://numba.readthedocs.io/en/stable/cuda
  `$ conda install numba cudatoolkit`
  `$ conda install cupy` https://cupy.dev (pyculib is obsolete)
  installation check: “`from numba import cuda`”

Computations

- CPU: serial
- many CPUs: MPI
- CPUs with many cores: openMP, shared memory
- CPUs (host) + GPUs (device)
  : heterogeneous and multithreads (control, data transfer)
- computation capacity = volume (high throughput) × speed (low latency)

GPU looks like

- GPU ≃ dual LED-monitors display
- Streaming Multiprocessor ≃ monitor
- CUDA core ≃ pixel
- Structural and logical hierarchy:
  cuda core => SM => GPU
  threads => blocks => grid



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

- ex) Geforce RTX 3080 has 8704 CUDA cores and 68 SMs

Key Concepts for GPU Computing

- Control flow
- CPU-GPU interaction via data transfer
- Hierarchical organization of threads: threads => blocks => grid(s)

HelloWorld.cu & HelloWorld.py

```
#include “iostream”
```

GPU kernel

```
__global__ void hello_fromGPU(int n)
{
  int tid = blockIdx.x*blockDim.x + threadIdx.x;
  printf(“Hello World from thread %d-%d\n”, n, tid);
}
```

CPU subroutine

```
void hello_fromCPU()
{
  printf(“Hello World from CPU\n”);
}
```

```
int main()
{
  hello_fromGPU<<<2,3>>>(0);
  // hello_fromGPU<<<2,3>>>(1);
  // cudaDeviceSynchronize();
  hello_fromCPU();
  return 0;
}
```

```
$ nvcc HelloWorld.cu; ./a.out
```

```
from numba import cuda

@cuda.jit
def hello_fromGPU(n):
  tid = cuda.blockIdx.x*cuda.blockDim.x +
        cuda.threadIdx.x
  print(“Hello World from thread “, n, tid)

def hello_fromCPU():
  print(“Hello World from CPU\n”)

hello_fromGPU[2,3](0)
#hello_fromGPU[2,3](1)
#cuda.synchronize()
hello_fromCPU()
```

```
$ python HelloWorld.py
```

HelloWorld.cu

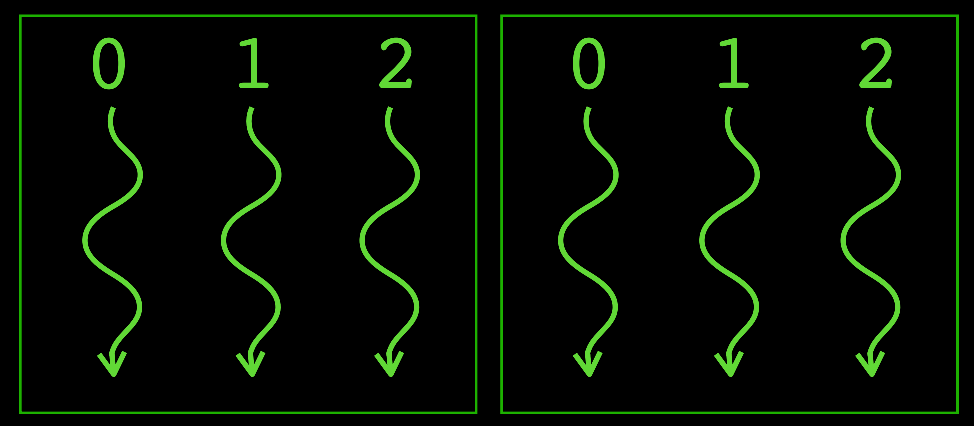
```
#include “iostream”

__global__ void hello_fromGPU(int n)
{
  int tid = blockIdx.x*blockDim.x + threadIdx.x;
  printf(“Hello World from thread %d-%d\n”, n, tid);
}

void hello_fromCPU()
{
  printf(“Hello World from CPU\n”);
}

int main()
{
  hello_fromGPU<<<2,3>>>(0);
  // hello_fromGPU<<<2,3>>>(1);
  // cudaDeviceSynchronize();
  hello_fromCPU();
  return 0;
}
```

GPU kernel

CPU subroutine

- **kernel with the prefix** `__global__`
  launched by host, executed in device

`threadIdx.x =`



built-in variables

`blockIdx.x = 0  1`

`gridDim` `blockDim`

- ***`<<<nBlocks,nThreads>>>`*?**
  creating (nBlocks * nThreads) threads

Control flow in “Hello World”

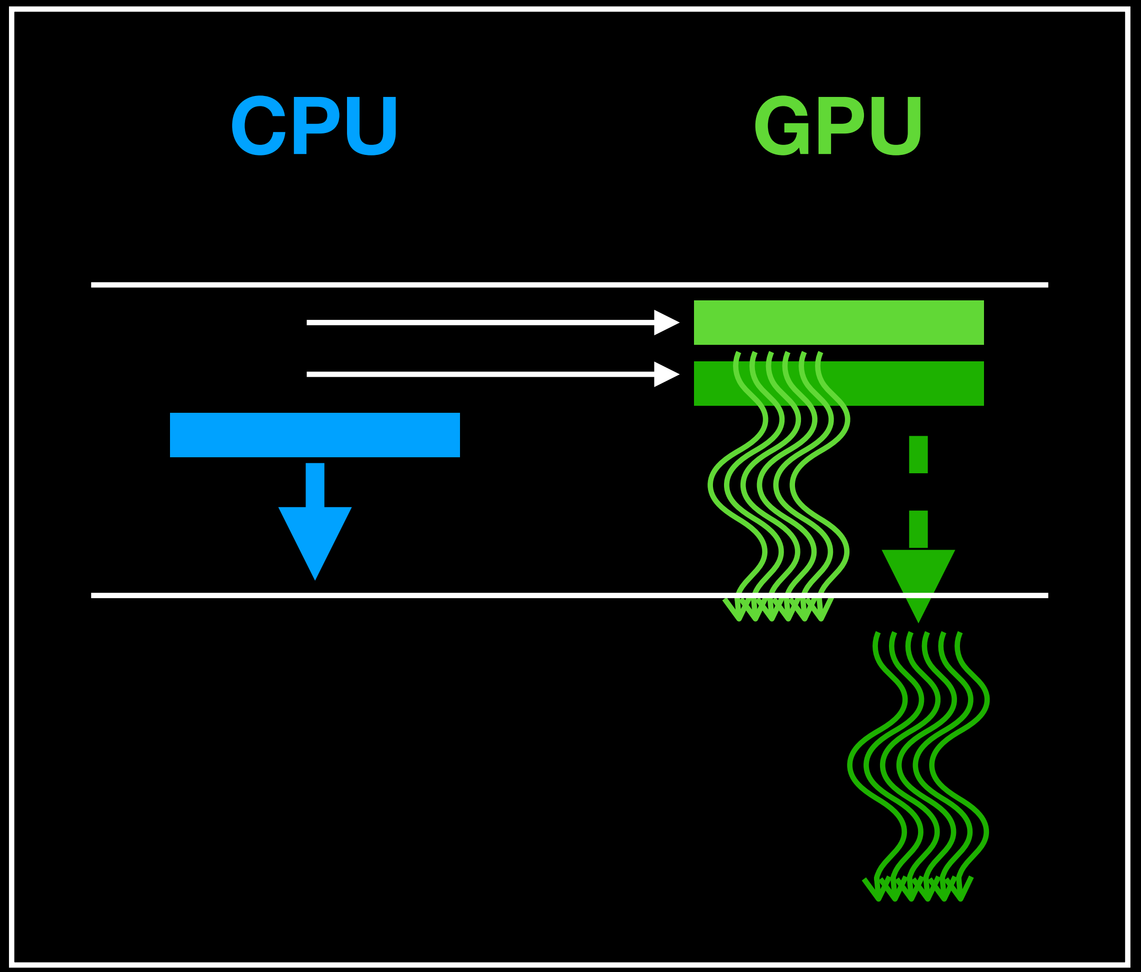
- Synchronous (serial) vs asynchronous (parallel)

- CPU
  - subroutine: SYNC
  - kernel: ASYNC†

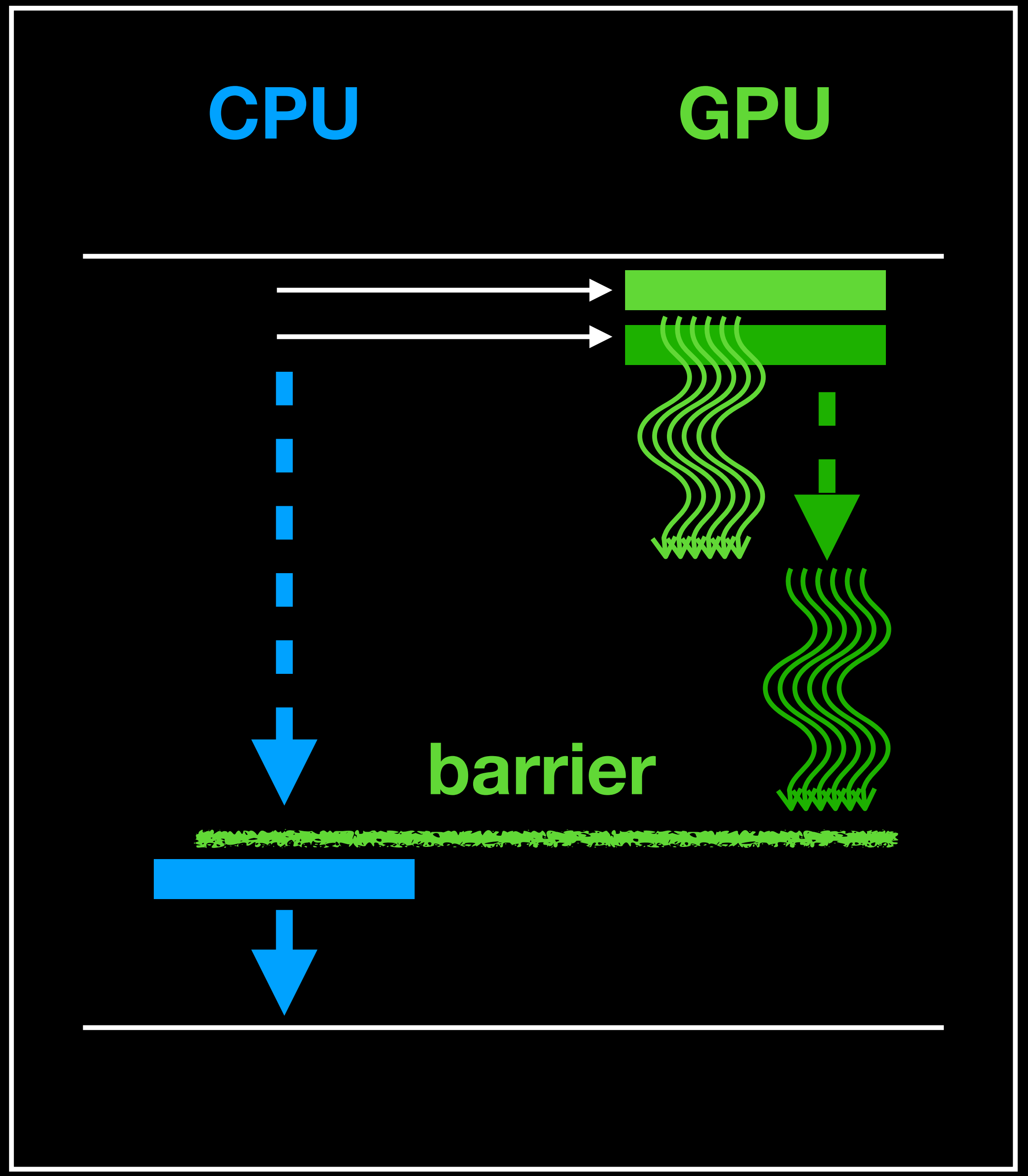
- GPU
  - threads in a kernel: ASYNC†
  - kernel -> kernel: SYNC†

- Racing condition for asynchronous update

†There are exceptions.



w/o `cudaDeviceSynchronize()`



`cudaDeviceSynchronize()`

Data Transfer

`cudaMemcpy(…, …, …, cudaMemcpyHostToDevice)`

host memory
(`malloc`)

device memory
(`cudaMalloc`)

`cudaMemcpy(…, …, …, cudaMemcpyDeviceToHost)`

Data Transfer (mem_cpy.cu or .py)

```
#include <iostream>
__global__ void add_constant(int n, int *x)
{
  int tid = threadIdx.x + blockIdx.x*blockDim.x;
  x[tid] += n;
}
int main(void)
{
  int dNum = 1<<24;
  int *x, *d_x, nT=32;
  size_t memSize = sizeof(int)*dNum;

  x = (int *)malloc(memSize);
  for(int i=0; i<dNum; i++) x[i] = i;

  cudaMalloc(&d_x, memSize);
  cudaMemcpy(d_x, x, memSize, cudaMemcpyHostToDevice);
  add_constant<<<dNum/nT, nT>>>(1, d_x);
  printf("%d\n", x[0]);
  cudaMemcpy(x, d_x, memSize, cudaMemcpyDeviceToHost);
  printf("%d\n", x[0]);
  return 0;
}
```

```
import numpy as np
from numba import cuda

@cuda.jit
def add_constant(n, arr):
  pos = cuda.grid(1)
  if n < arr.size:
    arr[pos] += n

dNum = 1<<24
nT = 32

x = np.arange(dNum, dtype=int)


d_x = cuda.device_array_like(x)
d_x = cuda.to_device(x)
add_constant[dNum//nT, nT](1,d_x)
print(x[0])
d_x.copy_to_host(x)
print(x[0])
```

Data Transfer

`cudaMemcpy(…, …, …, cudaMemcpyHostToDevice)`

host memory
(`malloc`)

device memory
(`cudaMalloc`)

`cudaMemcpy(…, …, …, cudaMemcpyDeviceToHost)`

* `cudaMemcpy` is very slow
  => minimize the number of data transfer (how?)

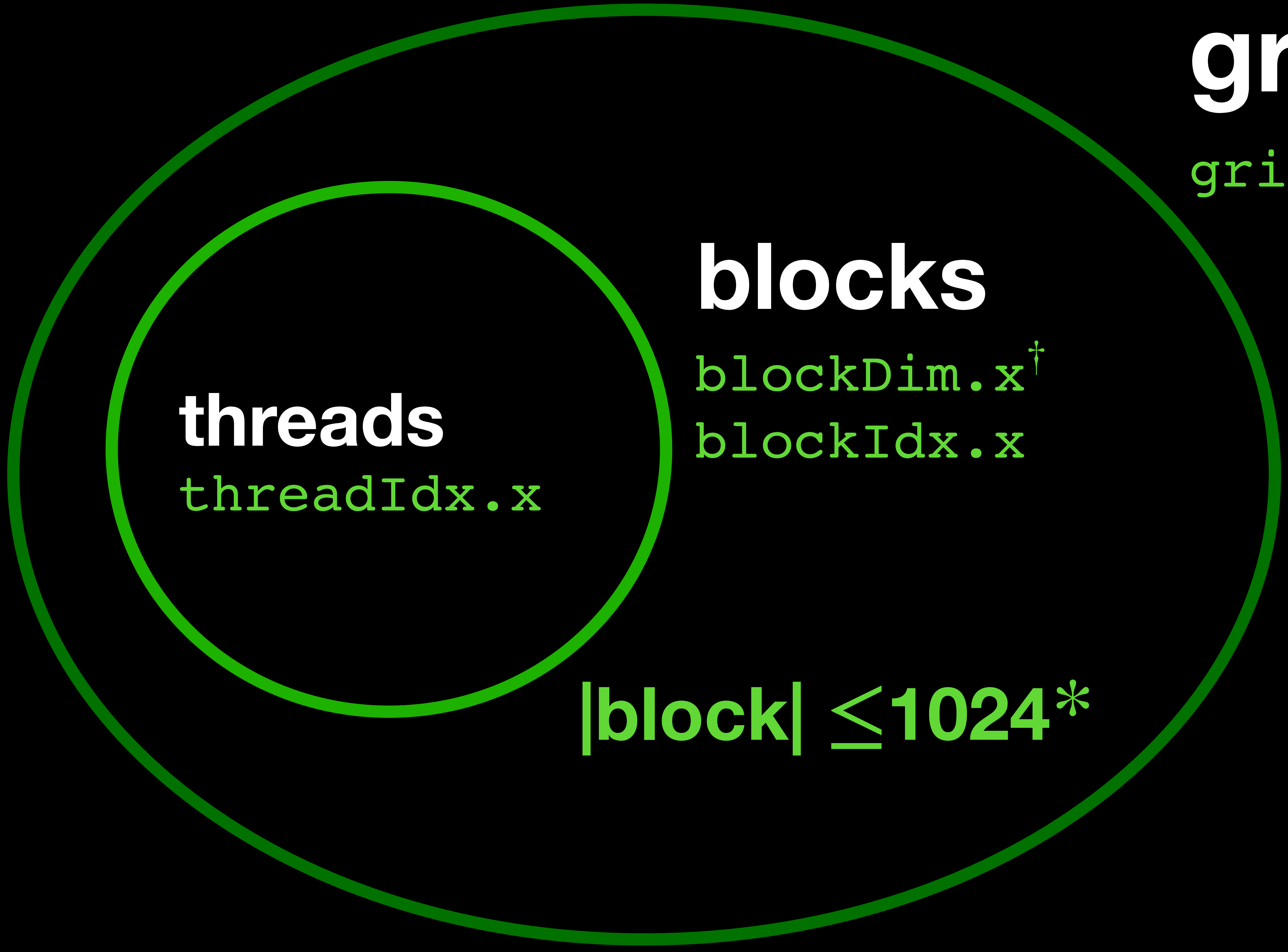
* **unified memory**: implicit memory copy

Structure of Grid

kernel<<< *m*[‡], *n*[†]>>> => creating teams of threads (indexed)

grid

`gridDim.x`[‡]



All threads perform
the same task (*kernel*)
with different parameters (*idx*)

single instruction, multiple threads (SIMT)

One-Dimensional Grid

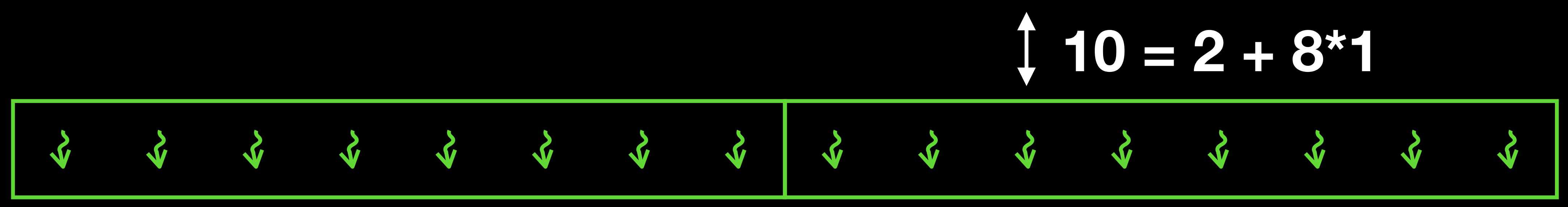
`kernel<<<m,n>>>`

thread identification `tid = threadIdx.x + blockDim.x*blockIdx.x;`

ex) vector of 16 components

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

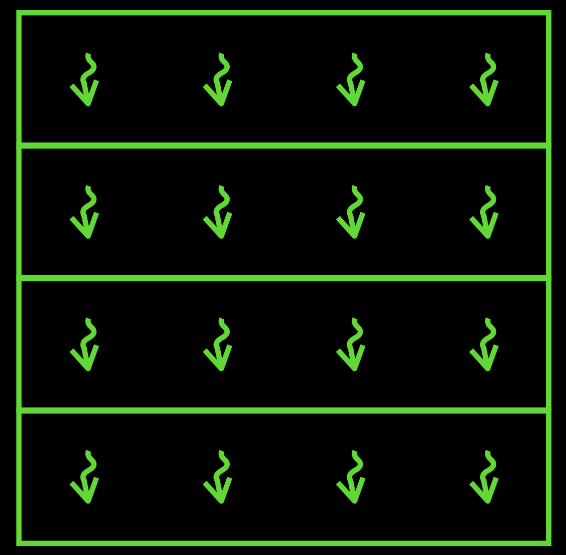
`(m,n)=(2,8)`



ex) 4×4 **matrix**

$\mathbf{M} = (\mathbf{M})_{\mathbf{ij}}$

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15



`(m,n) = (4,4)`

Multi-dimensional Grid

- `blockDim` and `gridDim` can be three dimensional
- Two-dimensional grid block <<<*nBlocks, nThreads*>>> with

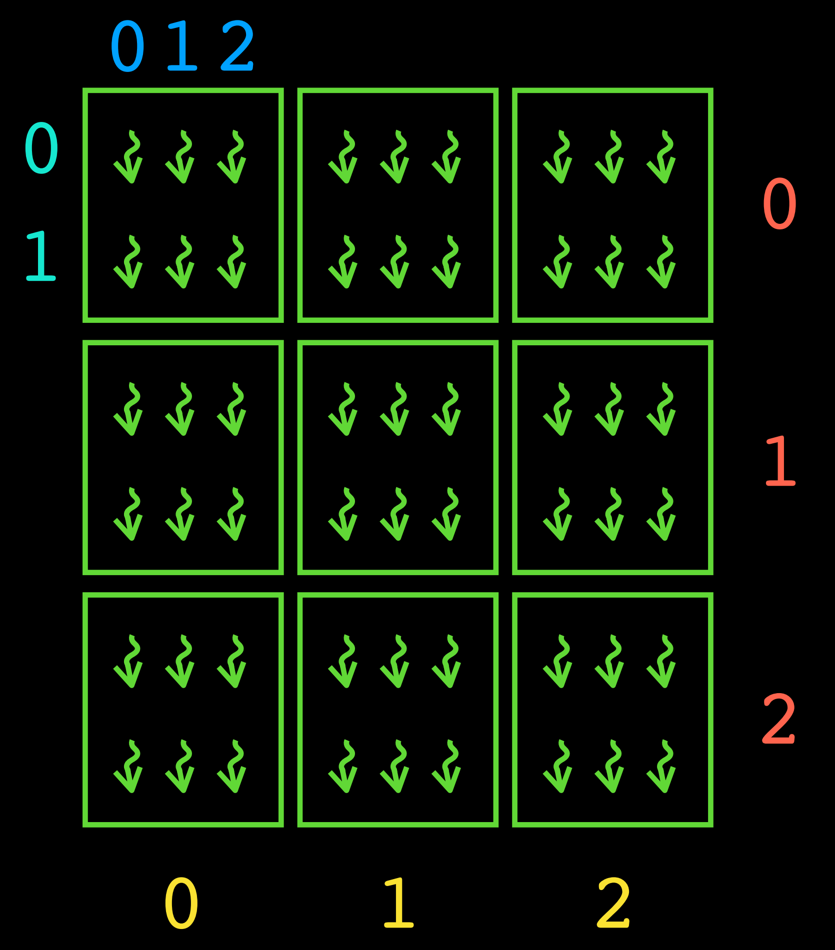
```
dim3 nBlocks(3, 3, 1), nThreads(3, 2, 1);
```

threadIdx.x

threadIdx.y

blockIdx.x

blockIdx.y



thread identification

```
tidx = threadIdx.x + blockDim.x*blockIdx.x;
tidy = threadIdx.y + blockDim.y*blockIdx.y;
```

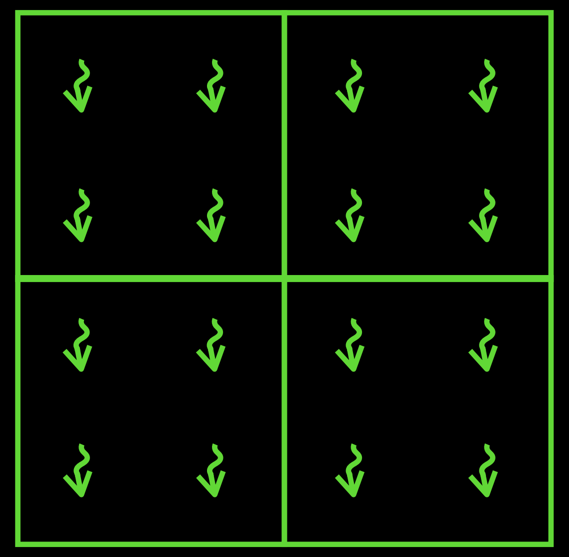
or

```
tid = tidx + (blockDim.x*gridDim.x) * tidy;
```

ex) 4×4 **matrix**

$\mathbf{M} = (\mathbf{M})_{\mathbf{ij}}$

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15



```
dim3 nblocks(2,2,1), nThreads(2,2,1)
```

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

Random Numbers: method 1 (rngHost.cu)

```
$ nvcc rngHost.cu -lcurand_static -lculibos; ./a.out
```

- Using host API: generating RN's in device memory from host

```
#include <curand.h>
```

- choice of PRNG

```
curandGenerator_t gen;
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
```

- initialization with a seed

```
curandSetPseudoRandomGeneratorSeed(gen, seed);
```

- n random numbers at an device array

```
// integers uniform in [0, 2^32)
curandGenerate(gen, devData, n);
// real numbers uniform in (0,1]
curandGenerateUniform(gen, devData, n);
```

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

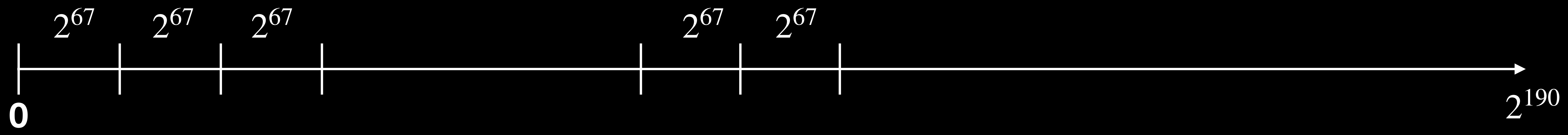
Random Numbers: method 2 (rngDevice.cu)

```
$ nvcc rngDevice.cu -lcurand_static -lculibos; ./a.out
```

- Using Device API inside a kernel

- Set up a PRNG for each thread

```
curandState *rngState;
curand_init(seed, tid, 0, &rngState[tid])
```



- Each thread calls a random number function with its own PRNG

```
curand(&rngState[tid]); //integers uniform in [0,2^32)
curand_uniform(&rngState[tid]); //real uniform in (0,1]
```

Seed for PRNG

- 재현 가능한 수치 데이터 ⇒ seed를 기록해둔다.
- 고정 시드 (예) `seed = 123456;`)
- “무작위”적인 시드
  - `unsigned int seed = time(0) or getpid(), …`
  - hardware가 제공하는 랜덤 넘버 (사용 전에 미리 체크 필요)

```
#include <random>
std::random_device rd;
unsigned int seed = rd();
```

Random numbers

- pseudo random number generator from the curand library

- host API† (`#include <curand.h>`)
  generate L^2 random numbers =>
  distribute them to the threads

- device API‡ (`#include <curand_kernel.h>`)
  create L^2 PRNGs and distribute them to the threads =>
  each thread generates its own random numbers

†MC example code adopts this method. **‡MD example code adopts this method.**

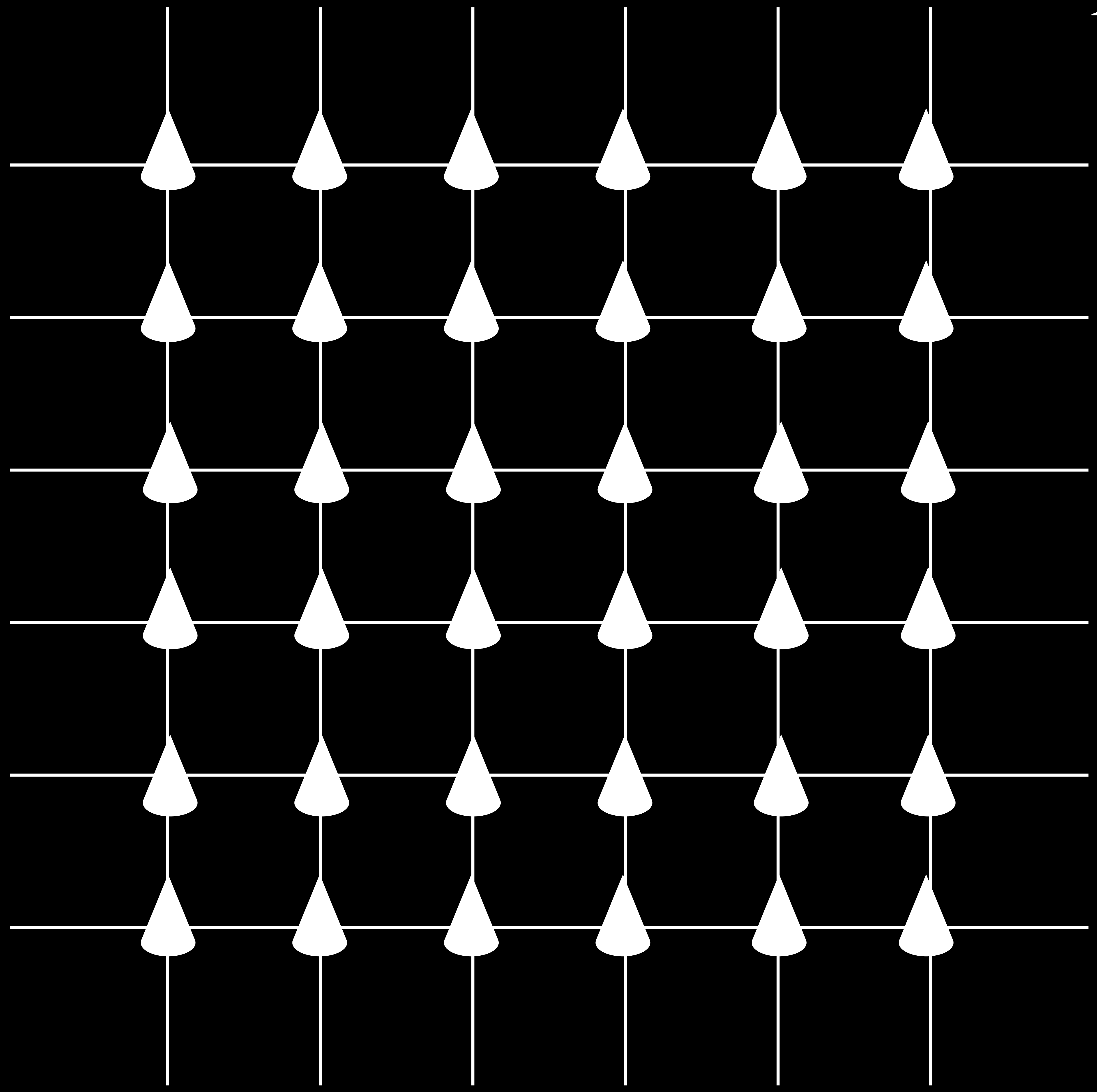
‡Lecture note of Prof. John Hughes (CS@Brown)
http://cs.brown.edu/courses/cs195v/lectures.shtml (Cuda Part 2, page 5-11)

Let’s CUDA
Monte Carlo Simulation

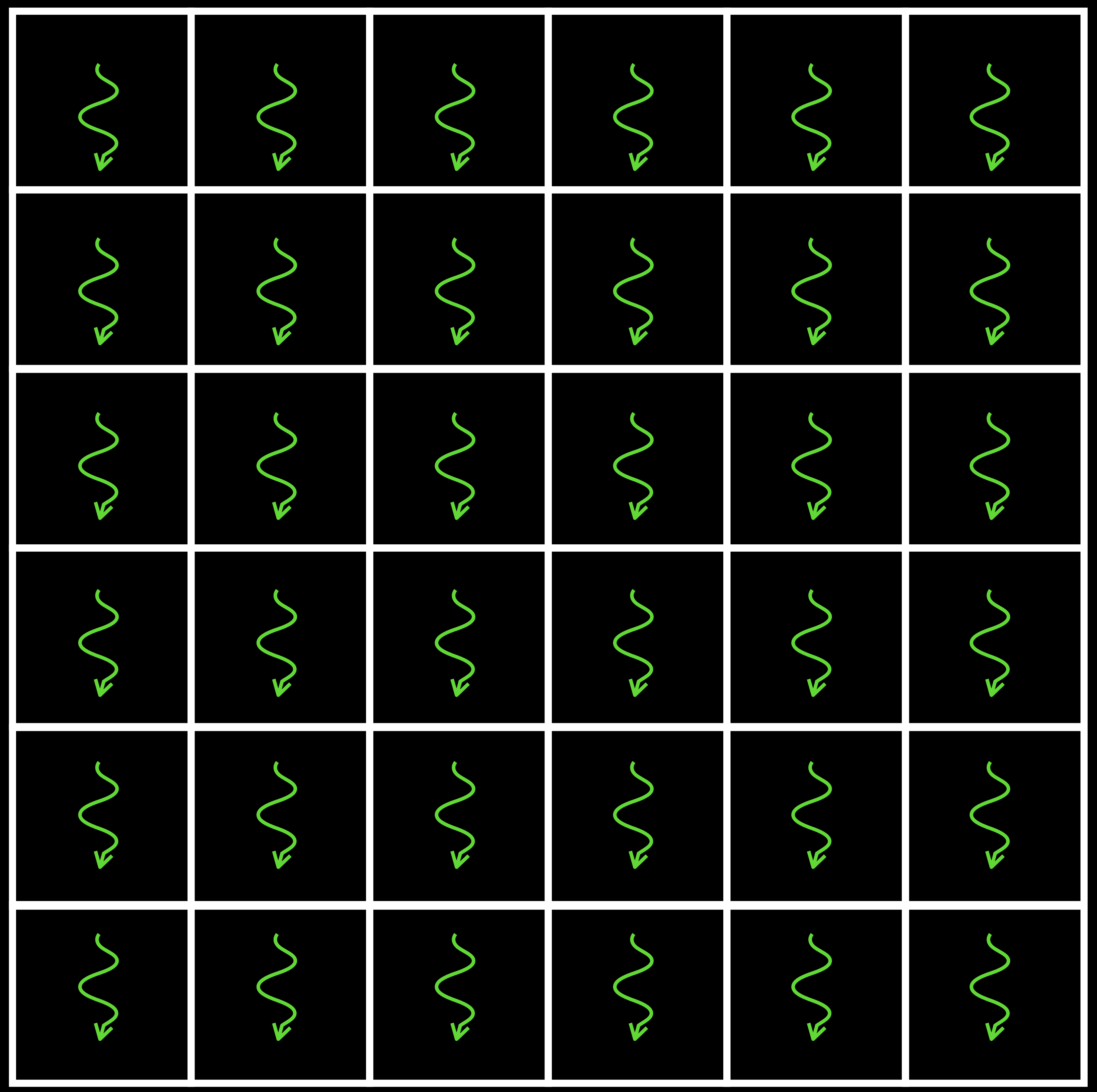
https://github.com/jdnoh/MonteCarloIsing

2D Ising model

$$E = -J \sum_{\langle i,j \rangle} s_i s_j$$



spin ⇔ thread



Monte Carlo Method for Ising model

- Select a site i at random
- Energy difference ΔE under a spin flip $s_i \to -s_i$
- Accept the spin flip with probability $P_{Metropolis}$ or else

```
int *spin;
__global__ void spinFlip_naive(…)
{
  int tid = …;
  int s = spin[tid];
  float dE = E(-s) - E(s);
  if(rand < exp(-β dE))
   spin[tid] *= -1
}
MC_naive<<<Ly, Lx>>>(…)
```

What's wrong?

RACE CONDITION

Random sublattice update

0	0	1	1	2	2
3	3	4	4	5	5
6	6	7	7	8	8
9	9	10	10	11	11
12	12	13	13	14	14
15	15	16	16	17	17

```
int *spinA, *spinB;
__global__ void spinFlip_even(…) {…}
__global__ void spinFlip_odd(…) {…}
void sublattice_update()
{
    if(rand<0.5)
        spinFlip_even<<<Ly, Lx/2>>>(…);
    else
        spinFlip_odd<<<Ly, Lx/2>>>(…);
}
```

2D Square lattice $L_x \times L_y$

=

even sublattice $(L_x/2) \times L_y \oplus$ **odd sublattice** $(L_x/2) \times L_y$

Sublattice Update

0	0	1	1	2	2
3	3	4	4	5	5
6	6	7	7	8	8
9	9	10	10	11	11
12	12	13	13	14	14
15	15	16	16	17	17

neighbors of a spin at (x, y)

$$(x, y),\ (x, y+1),\ (x, y-1),\ \left(x-(-1)^y, y\right)$$

neighbors of a spin at (x, y)

$$(x, y),\ (x, y+1),\ (x, y-1),\ \left(x+(-1)^y, y\right)$$

in summary,

$$(x, y) \Rightarrow (x, y),\ (x, y \pm 1),\ \left(x-(-1)^{p+y}, y\right)$$

parity variable $p = 0(even)\ or\ 1(odd)$

Don't forget the periodic boundary condition

Romeo et el, https://doi.org/10.1016/j.cpc.2020.107473 (Sec. 3.1 and Fig. 1, 2)

Grid setting <<<$L_y, L_x/2$>>>

initialization of 2 PRNGs and system configurations

t-loop

prepare N/2 random numbers using hostAPI

with prob. 1/2, A- or B-sublattice update

order parameters measurement

Order parameters

- magnetization $M = \sum_{i=0}^{L^2-1} s_i$: nontrivial task in parallel processing

- **thrust** library is extremely useful

```
#include <thrust/reduce.h>
M = thrust::reduce(spin, spin+N, 0, thrust::plus<long int>())
=> spin[0]에서부터 spin[N-1]까지의 값을 long int 변수로 간주하고 모두 더하라. 단, 초기값은 0이다.
```

`* spin` (device memory) → `thrust::device_ptr<int>(spin)`

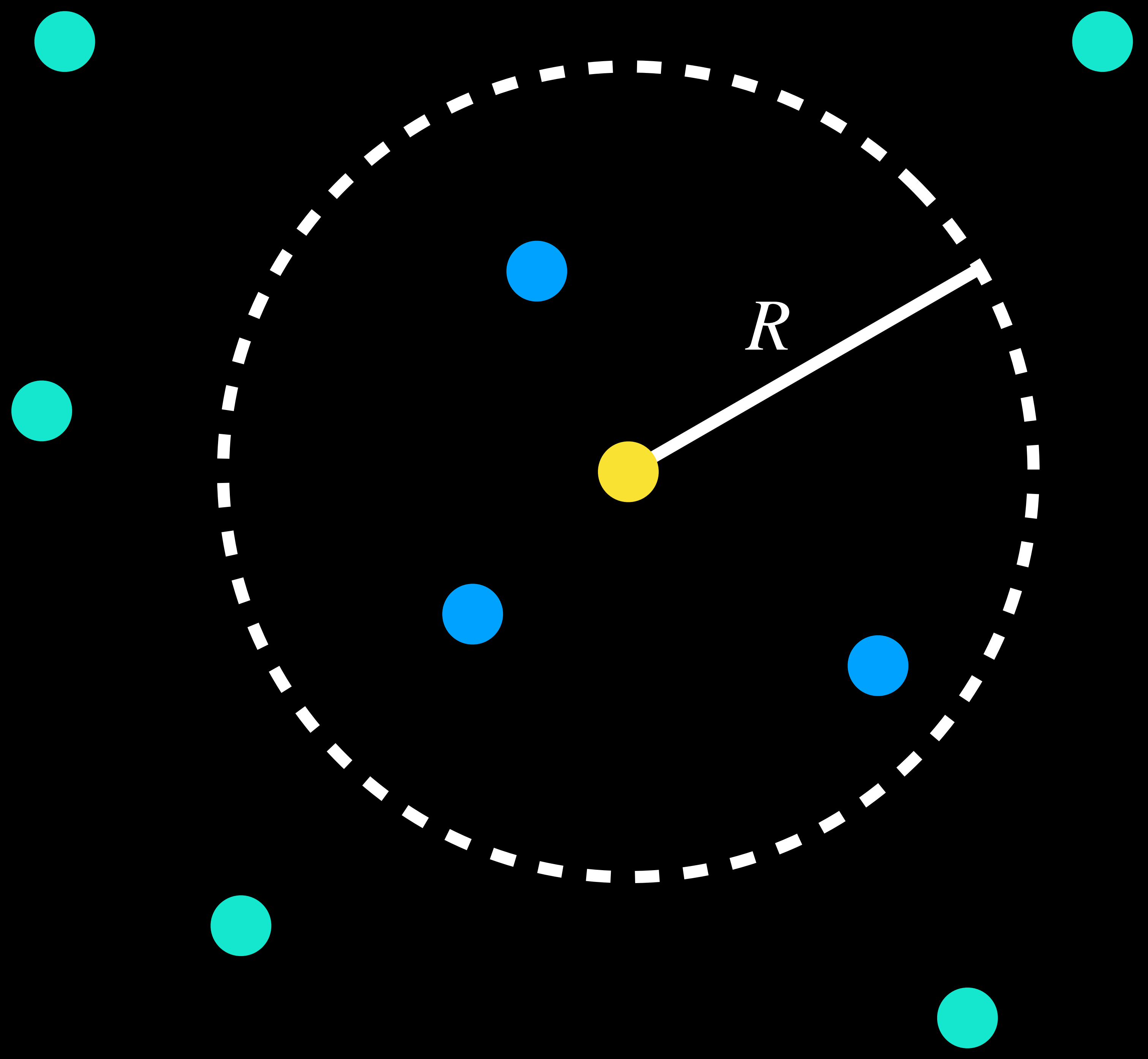
Run

- Compile: cudaIsing_AB.cu와 subIsing_AB.c를 같은 폴더에 놓고
  `$ nvcc -O2 cudaIsing_AB.cu -lm -lcurand_static -lculibos`
- Run: `$ ./a.out 1024 1000 100`
- 격자 크기를 바꿔가며 run time 비교
- 그리드 크기를 바꿔가며 run time 비교
- Profiling: `$ nsys nvprof ./a.out 1024 1000 100`

Let’s CUDA
Molecular Dynamics

https://github.com/jdnoh/VicsekModel

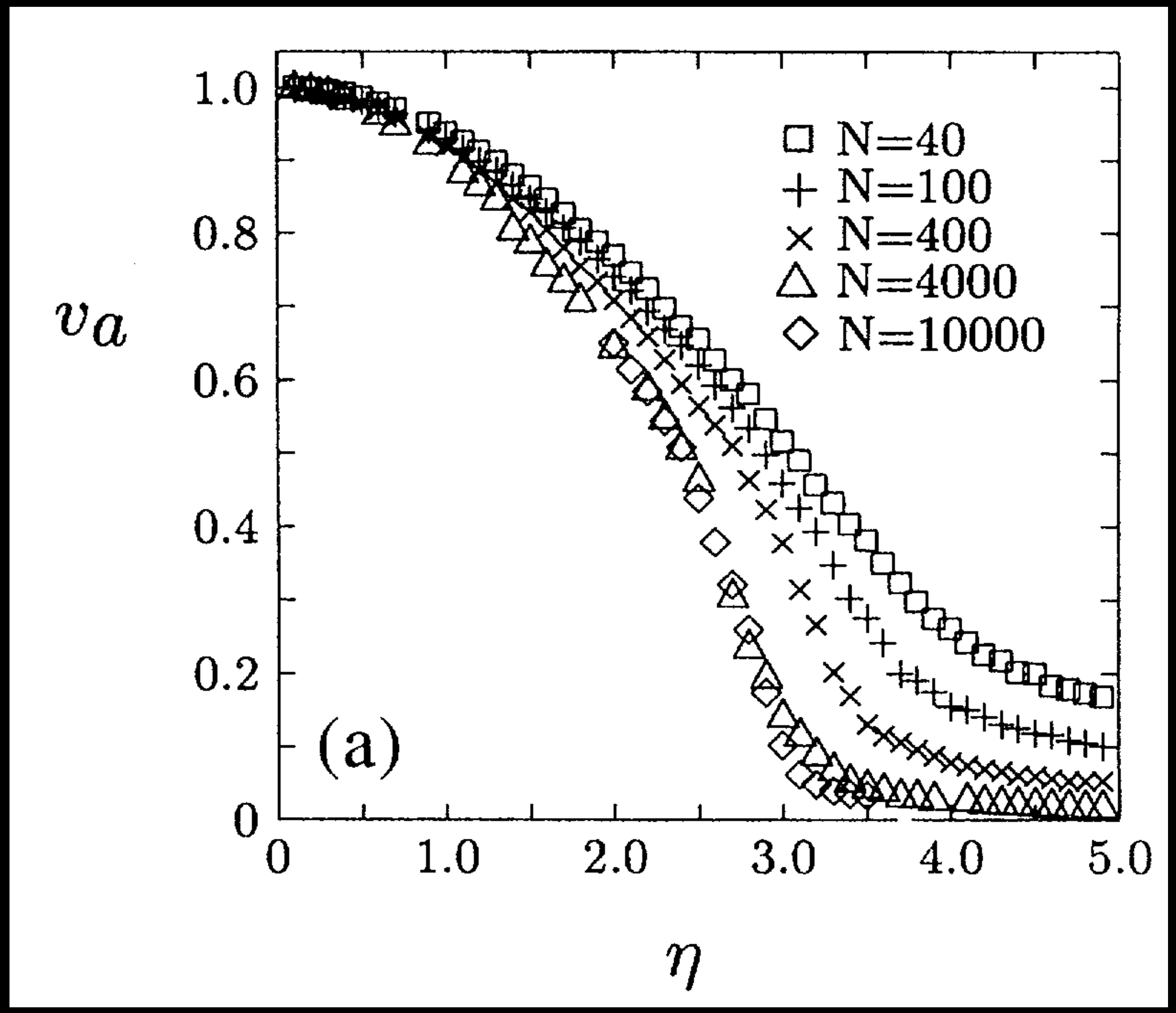
Vicsek model



active particles at constant speed

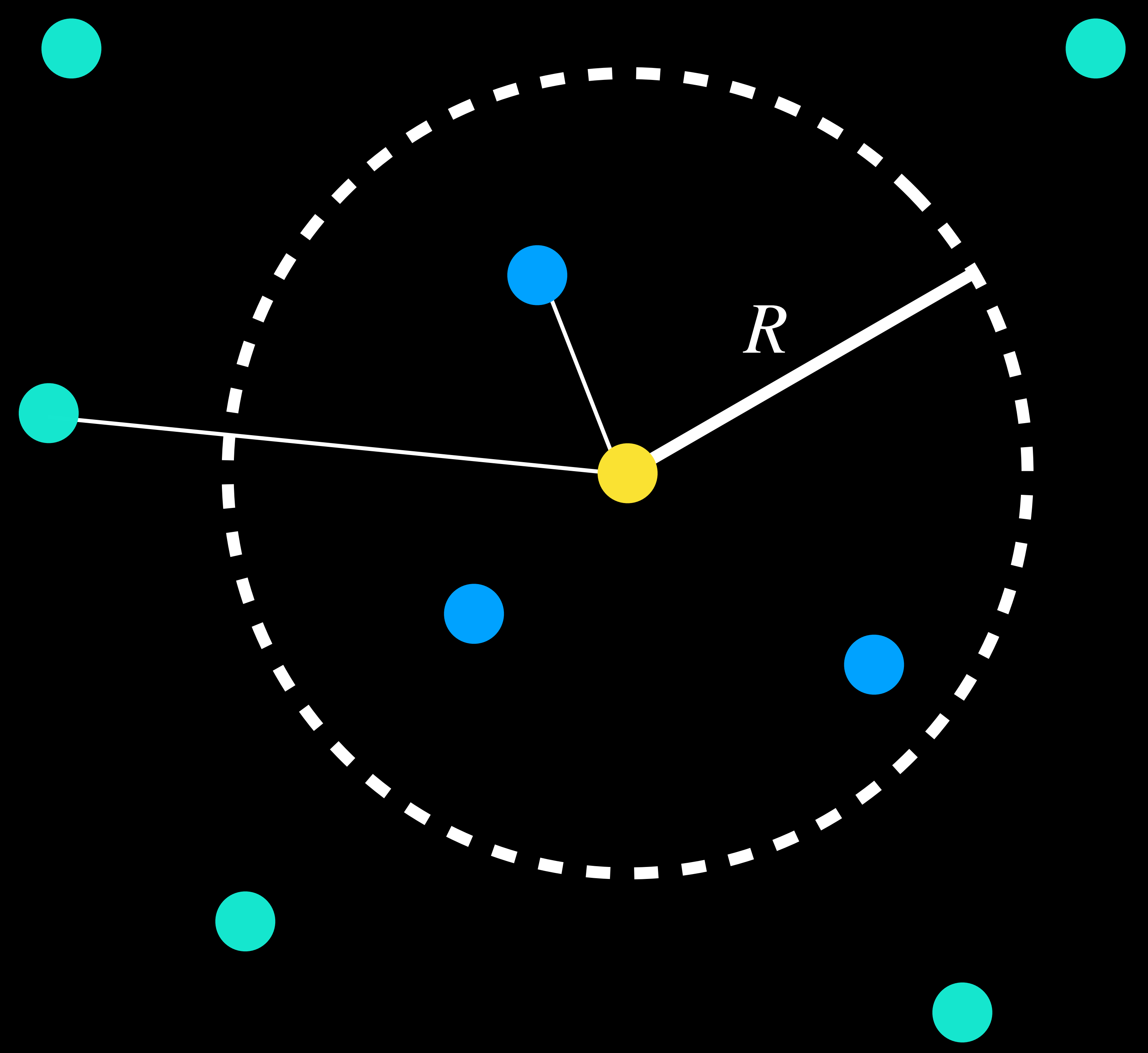
$$\mathbf{r}_i = (x_i, y_i), \ \ \mathbf{v}_i = v_0(\cos\theta_i, \sin\theta_i)$$

noisy velocity aligning interaction



https://doi.org/10.1103/PhysRevLett.75.1226

Vicsek model



active particles moving with constant speed

$$\mathbf{v}_i = v_0(\cos\theta_i, \sin\theta_i), \quad \mathbf{r}_i(t+1) = \mathbf{r}_i(t) + \mathbf{v}_i(t)$$

noisy velocity aligning interaction

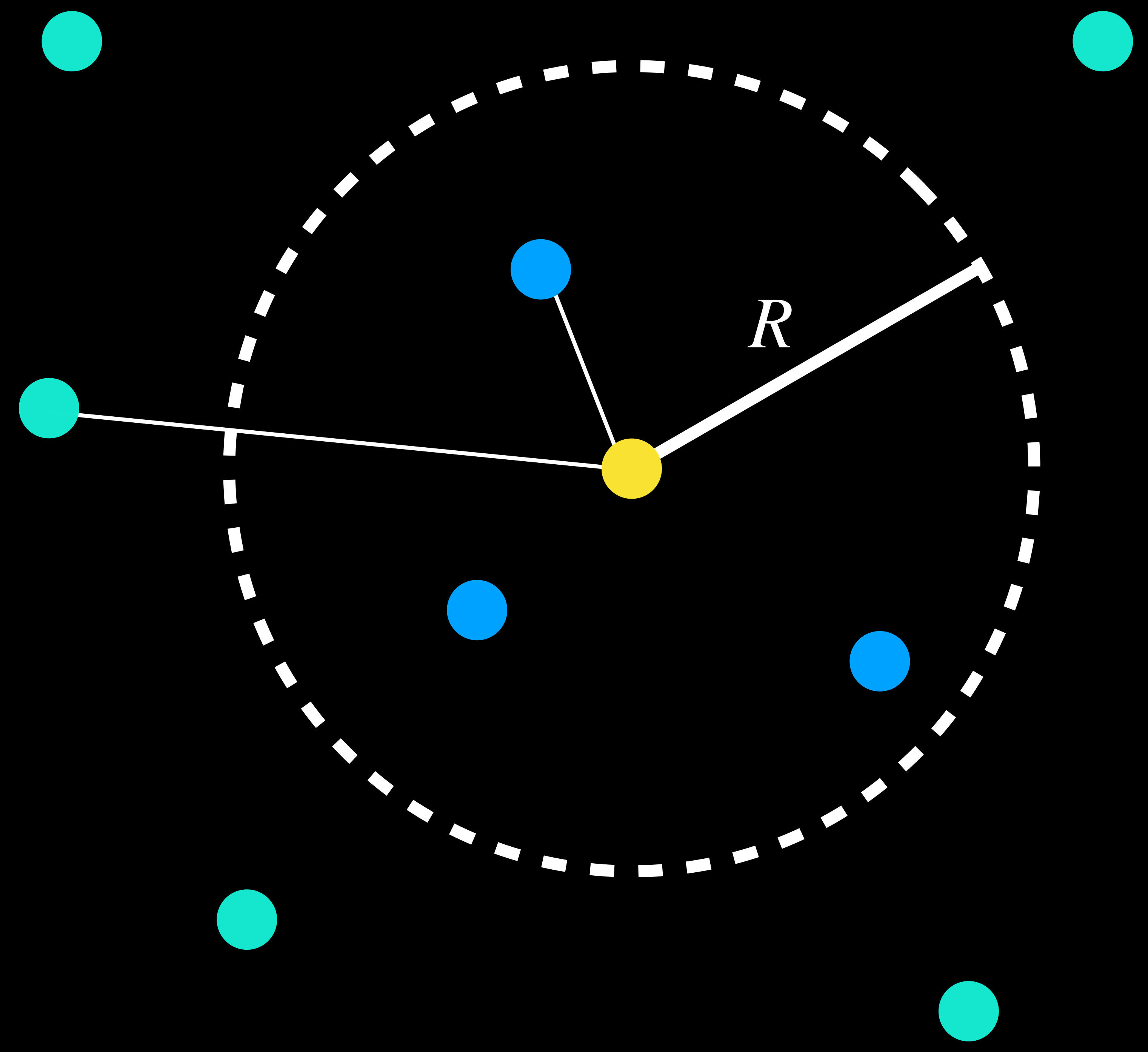
local average velocity

$$\mathbf{V}_i = \sum_{|\mathbf{r}_j - \mathbf{r}_i| < R} \mathbf{v}_j \propto (\cos\Theta_i, \sin\Theta_i)$$

$$\theta_i \to \Theta_i + \eta r \text{ with } r \in (-1:1)$$

order parameter = $\dfrac{1}{v_0 N}\left|\sum_i \mathbf{v}_i\right|$

“Force” calculations



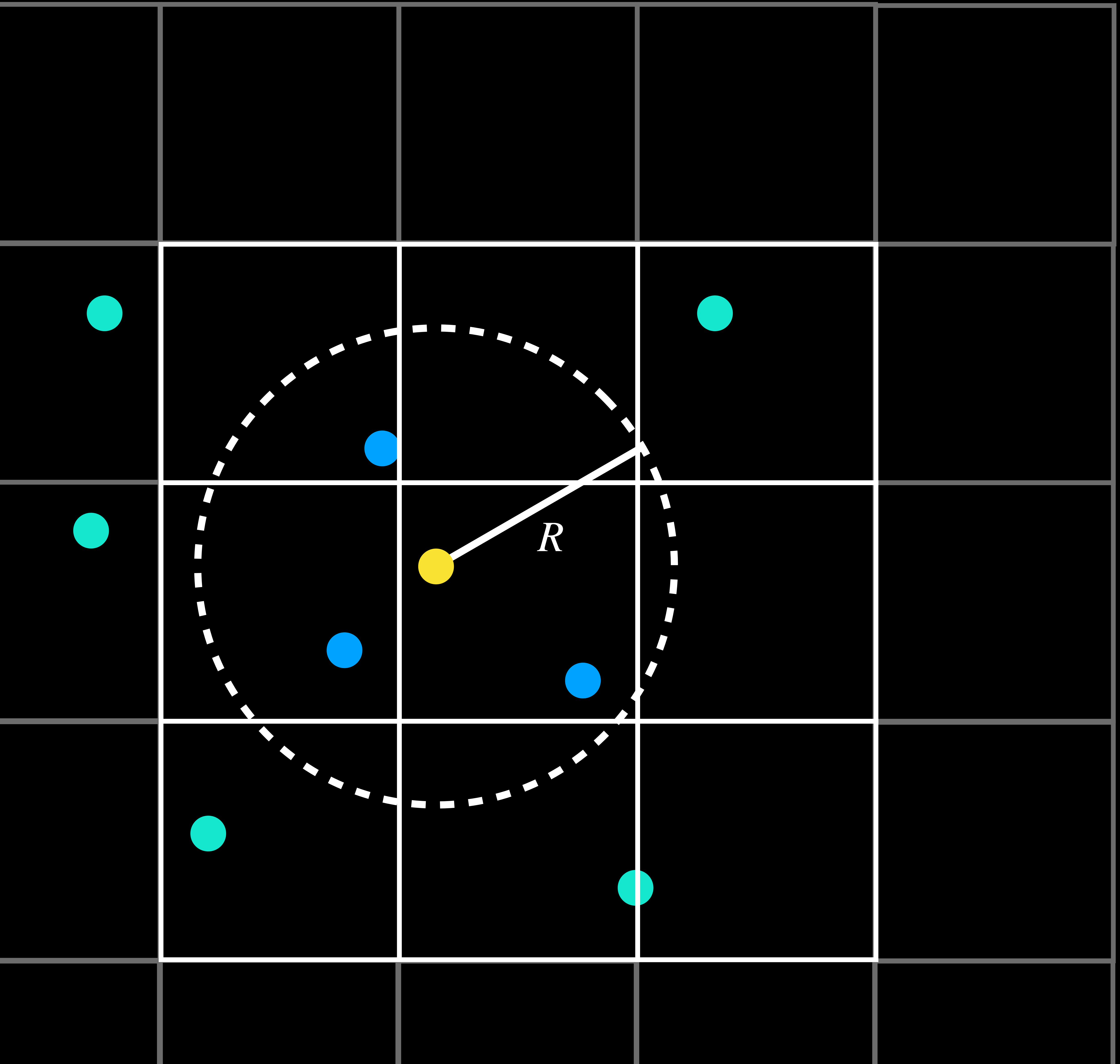
in general MD calculations,

F_i : “force” on a particle i

```
for(all j=1,…,N)
    if(d_ij ≤ R) F_i += f_ij
```

N^2 operations: high cost

Cell Lists



Divide the space into cells of size R^2

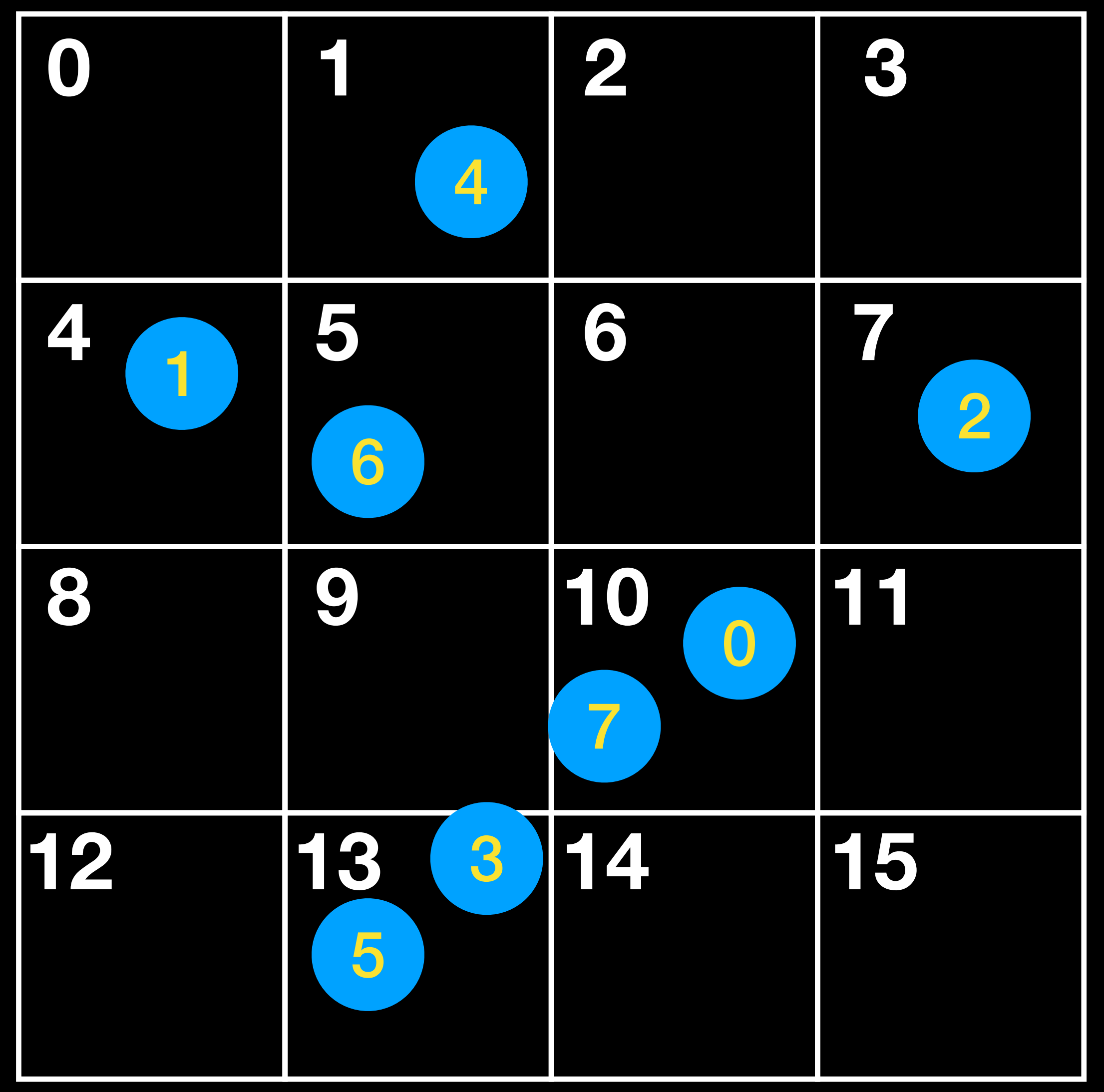
F_i **: force on a particle** i

```
// N operations
for(all j=1,…,N)
    if(d_ij ≤ R) F_i += f_ij
```

$\Rightarrow$

```
// ~9ρ operations
for(only j in 9 cells)
    if(d_ij ≤ R) F_i += f_ij
```

Linked List using Sort



8 particles in 16 cells

ptl.	cell
0	10
1	4
2	7
3	13
4	1
5	13
6	5
7	10

sort with cell idx →

ptl.	cell
4	1
1	4
6	5
2	7
0	10
7	10
3	13
5	13

relabel ptl. idx →

ptl.	cell
0	1
1	4
2	5
3	7
4	10
5	10
6	13
7	13

http://docs.nvidia.com/cuda/samples/5_Simulations/particles/doc/particles.pdf (page 6,7)

Linked List

`sort_by_key` **in the** **`thrust`** **library**

```
#include <thrust/sort.h>

find_address<<<nB, nT>>>(…) // cell index

thrust::sort_by_key(cell_idx, cell_idx+nPtls, particle_idx)

cell_head_tail<<<nB, nT>>>(…) // head and tail of cells
```

⇒ `head[cell]` ≤ `particle_idx[cell]` ≤ `tail[cell]`

Grid setting

initialization of PRNG and system configurations

t-loop

particles move

linked list

force calculation

particles rotate

order parameters measurement

Run

- Compile: cudaVicsek.cu와 subVicsek.c를 같은 폴더에 놓고
  `$ nvcc -O2 cudaVicsek.cu -lm -lcurand_static -lculibos`
- Run: `$ ./a.out 1024 1000 100`
- 격자 크기를 바꿔가며 run time 비교
- 그리드 크기를 바꿔가며 run time 비교
- Profiling: `$ nsys nvprof ./a.out 1024 1000 100`

최적화를 위하여

- 그리드 크기를 잘 선택하자
- 하드웨어, 특히 메모리 구조를 잘 이해하자
- CPU와 GPU를 동시에 돌리자
- multi streams and multi GPU
- 라이브러리 함수를 잘 이용하자 (cuBLAS, cuFFT, cuRAND, Thrust, …)
- profiling tools (Nsight systems,…)과 친해지자