2017 Practical Parallel Computing (実践的並列コンピューティング) No. 13

GPU Programming with CUDA (2)

Toshio Endo

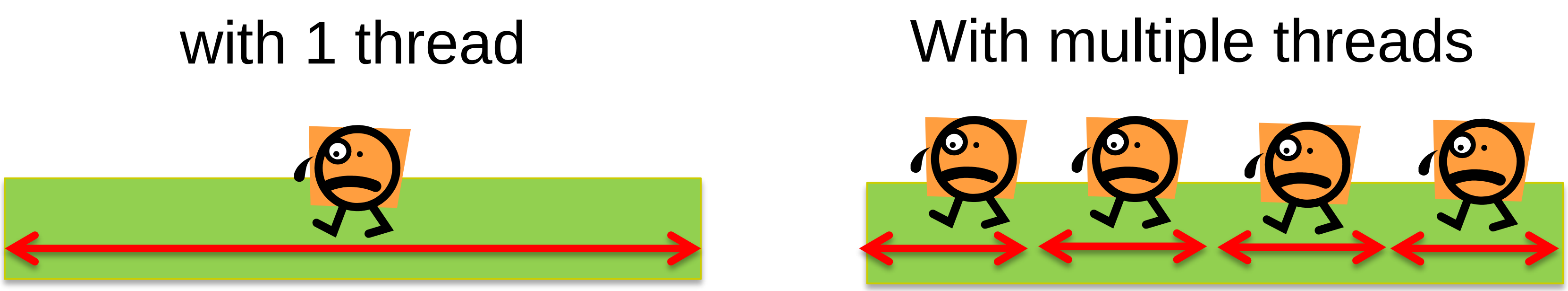
School of Computing & GSIC

endo@is.titech.ac.jp

Parallelization on CUDA

- In order to utilize speed of GPUs, we need to use multiple threads for parallelization
  - "inc_seq" sample program only use 1 thread
  - → The next sample is "inc_par"

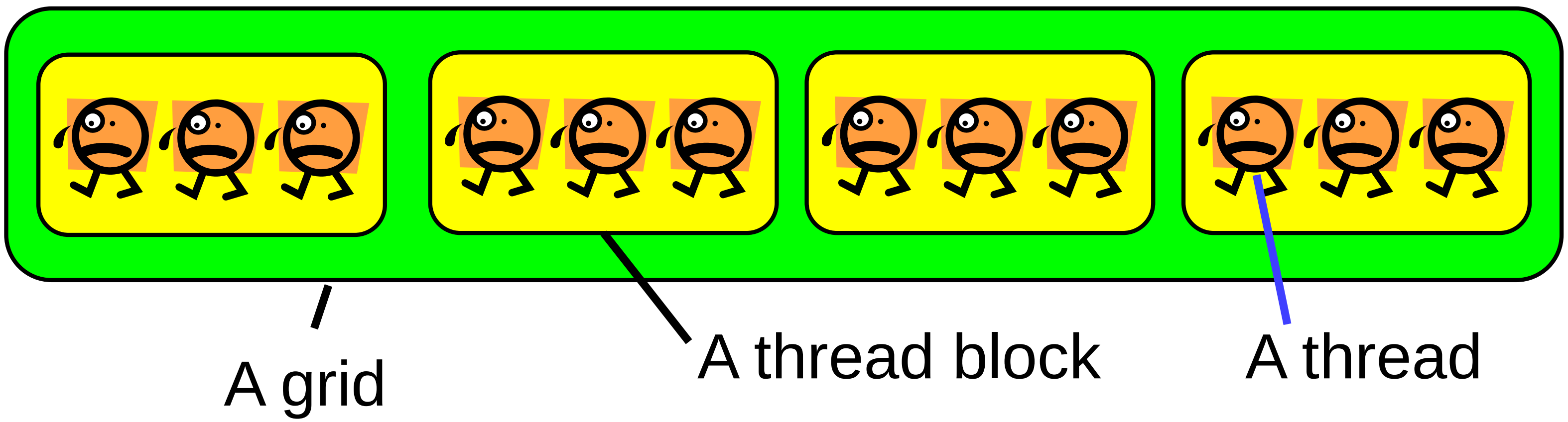
  Available at ~endo-t-ac/ppcomp/17/inc_par



Parallelization on CUDA (2)

OpenMP: Specify <u>1 number</u> for number of threads (OMP_NUM_THREADS)

CUDA: Specify <u>2 numbers</u> (at least) for number of threads, when calling a GPU kernel function



cf) func <<< 4, 3 >>> (); → 12 threads

4: Number of thread blocks = gridDim

3: Number of threads per block = blockDim

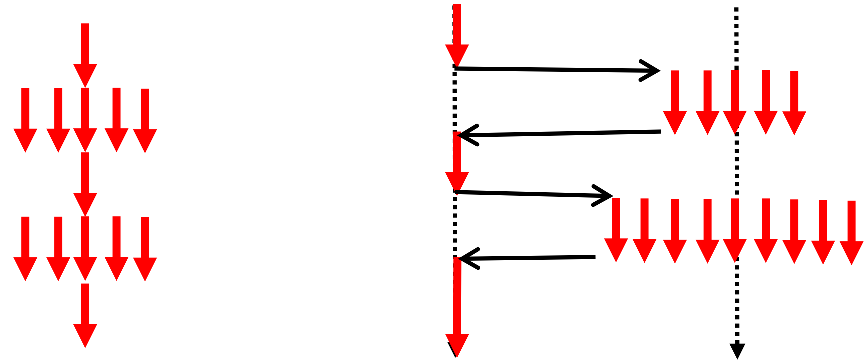
To See Who am I

- By reading the following special variables, each thread can see its thread ID, etc.
- My ID
  - blockIdx.x: Index of the block the thread belong to (≧0)
  - threadIdx.x: Index of the thread (inside the block) (≧0)
- Number of thread/blocks
  - gridDim.x: How many blocks are running
  - blockDim.x: How many threads (per block) are running

Note: In order to see the entire sequential ID, we should compute

blockIdx.x * blockDim.x + threadIdx.x

Differences between OpenMP Threads & CUDA Threads



	OpenMP threads	**CUDA threads**
Run on	CPU	GPU
When the number is specified	Environment var (OMP_NUM_THREADS)	When GPU kernel is called
How the number is specified	1 number	2 numbers at least 6 numbers at most (explained later)
“Desirable” thread numbers	Threads ≦ CPU cores	gridDim ≧ SMXs (=14 on K20X) blockDim ≧ CUDA cores per SMX (=192 on K20X)

Parallel “inc_par” Sample

inc_seq

```
        :


__global__ void inc(int *array, int len)
{
    int i;
    for (i = 0; i < len; i++) {
        array[i]++;
    }
    return;
}

int main(int argc, char *argv[])
{
        :
    inc<<<1, 1>>>(arrayD, N);
        :
}
```

inc_par

```
        :
#define BS (8)

__global__ void inc(int *array, int len)
{
    int id = blockIdx.x * blockDim.x +
        threadIdx.x;
    if (id >= len) return;
    array[id]++;    // ← we can omit loop
    return;
}

int main(int argc, char *argv[])
{
        :
    inc<<<(N+BS-1)/BS, BS>>>(arrayD, N);
        :
}
```

Ideas behind inc_par

- It is ok to make >1000, >10000 threads on CUDA
- We use N threads for N elements computation

```
inc<<<N/BS, BS>>>(.....);
```

gridDim → N/BS

blockDim (=8 in this sample) → BS

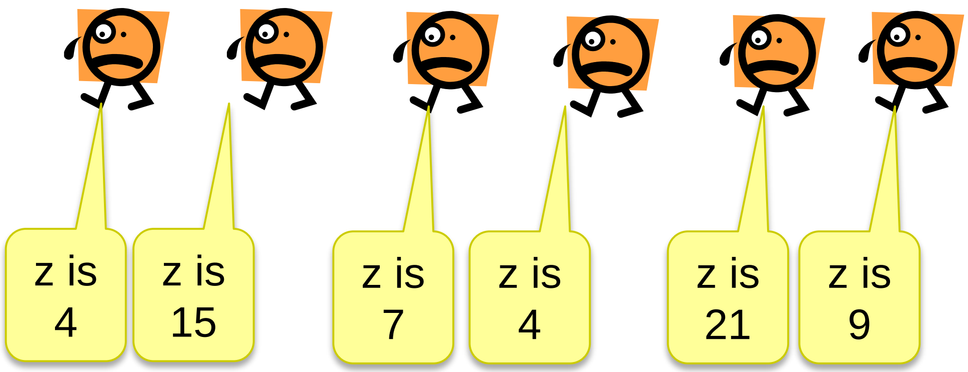
- 1 element for 1 thread → No need of "for" loop in this sample

Note: <<<N, 1>>> or <<<1, N>>> also works, but inefficient

Note: To support the case N is indivisible by BS, we actually use <<<(N+BS-1)/BS, BS>>>
→ "Extra" threads (id≧N) should not work
→ if (id >= len) return;

Rules for Memory/Variables

- Variables declared in GPU kernel functions are "thread private"



- Device memory is shared by all CUDA threads
  - Be careful to avoid race condition problem (multiple threads write same address)
  - Reading same address is ok
- Do not forget host memory and device memory are distributed

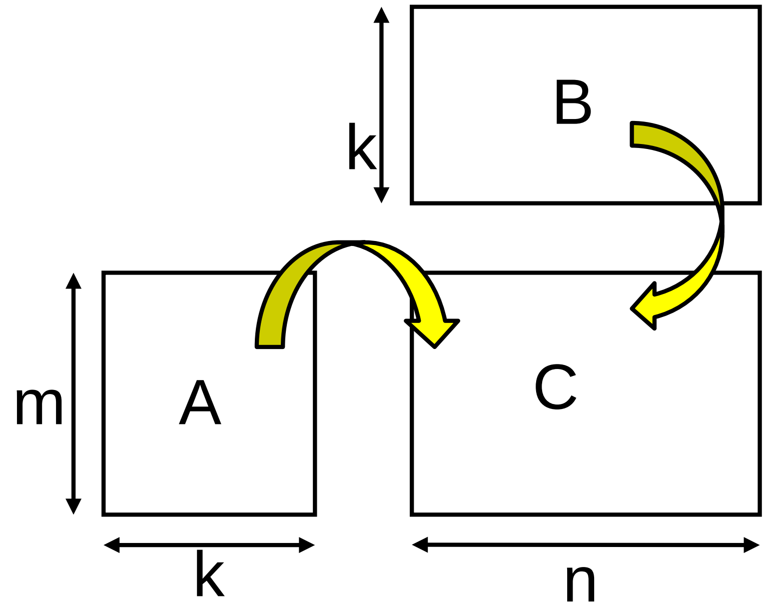
"mm" sample: Matrix Multiply
(Revisited, related to [G2])

CUDA version available at ~endo-t-ac/ppcomp/17/mm-cuda/

A: a (m × k) matrix, B: a (k × n) matrix

C: a (m × n) matrix

$C \leftarrow A \times B$



- Supports variable matrix size.
  - Each matrix is expressed as a 1D array by *column-major* format
- Execution:./mm [m] [n] [k]

On CUDA, We need to design
(1) How we parallelize computation
(2) How we put data on host memory & device memory

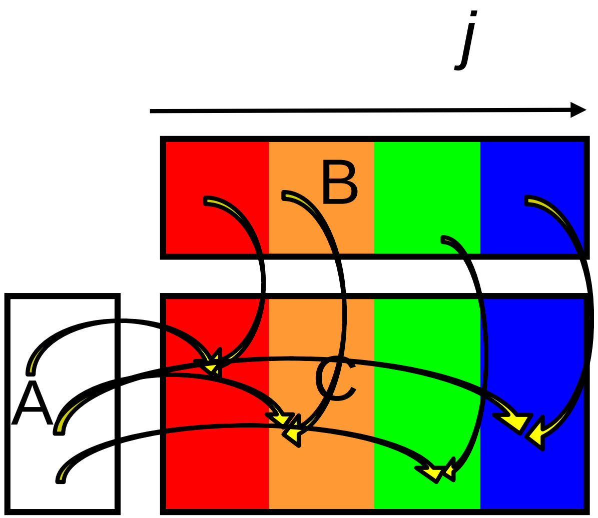
How We Parallelize Computation

In mm, we can compute different C elements in parallel

- On the other hand, it is harder to parallelize dot-product loop

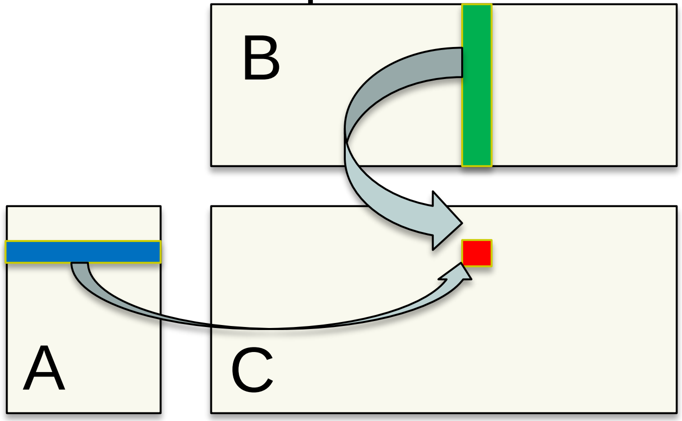
OpenMP

- Parallelize column-loop (or row-loop)



CUDA

- We can create too many threads
  → M x N threads are ok!!
- Parallelize row&column of C
- 1 thread computes 1 element

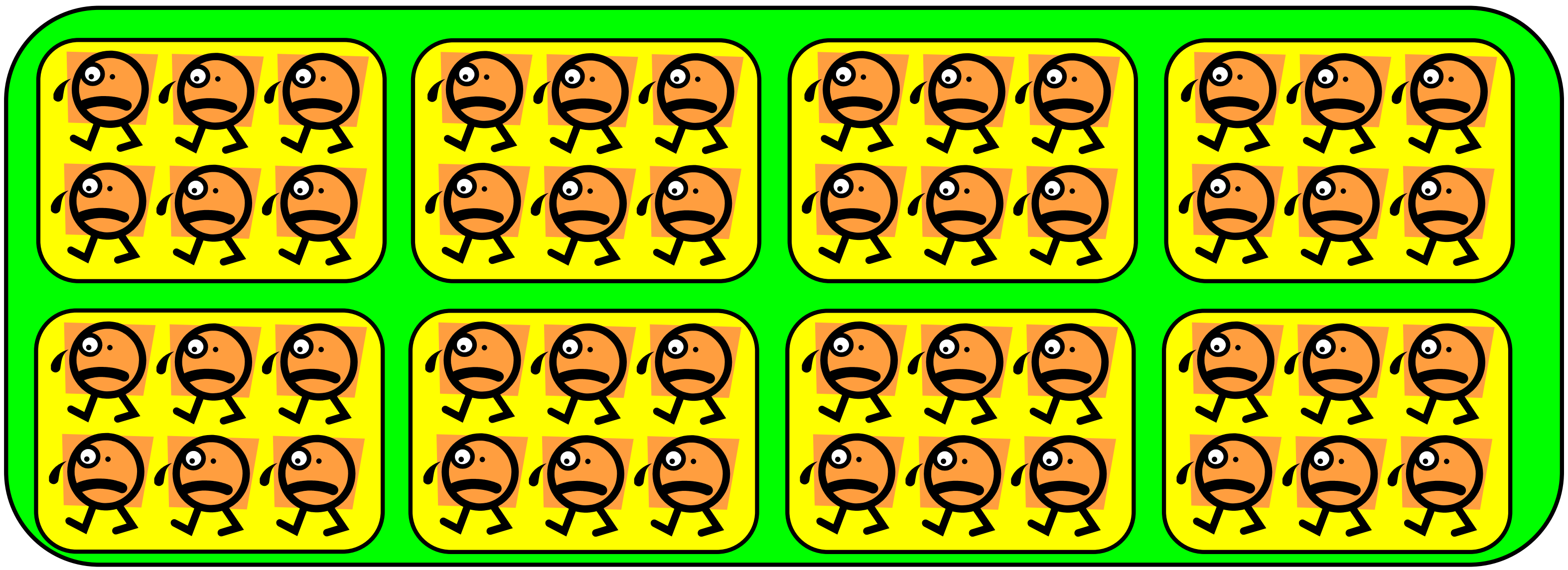


※ This is not the unique way

Creating Many Threads

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

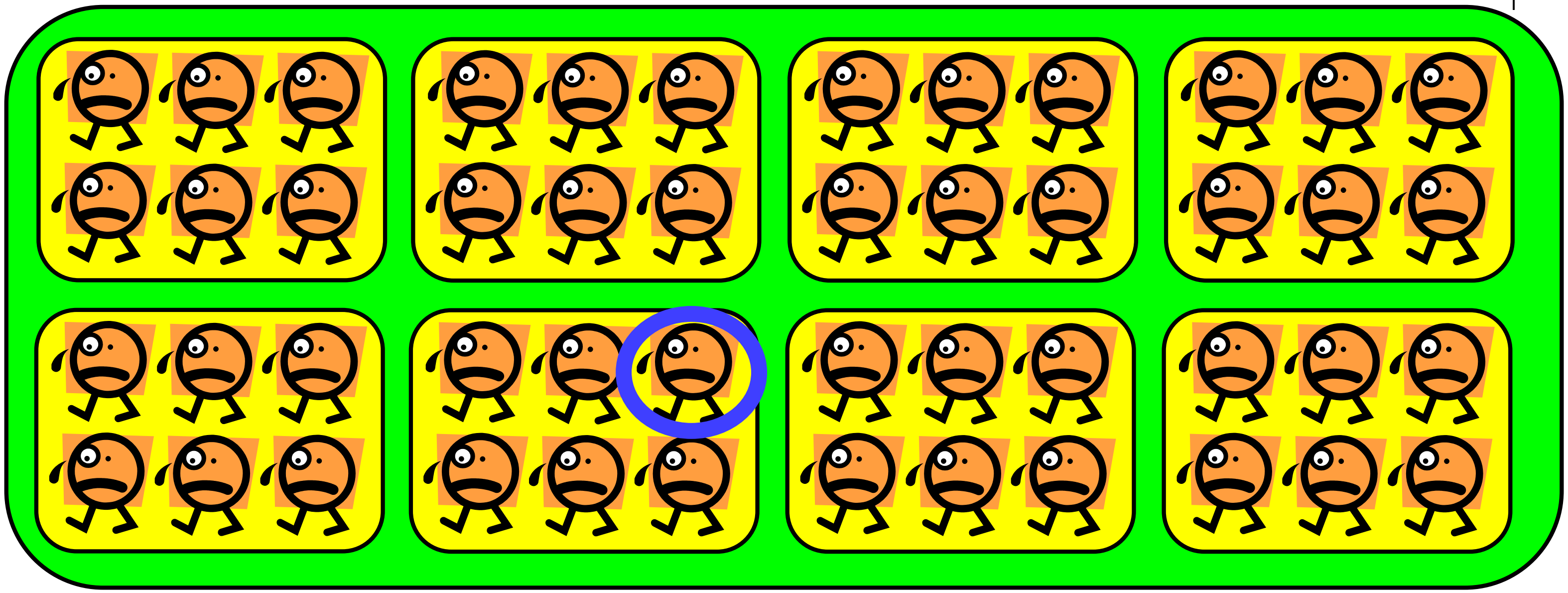
cf) func <<< dim3(4,2,1), dim3(3,2,1) >>> (); → 48 threads



※ 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

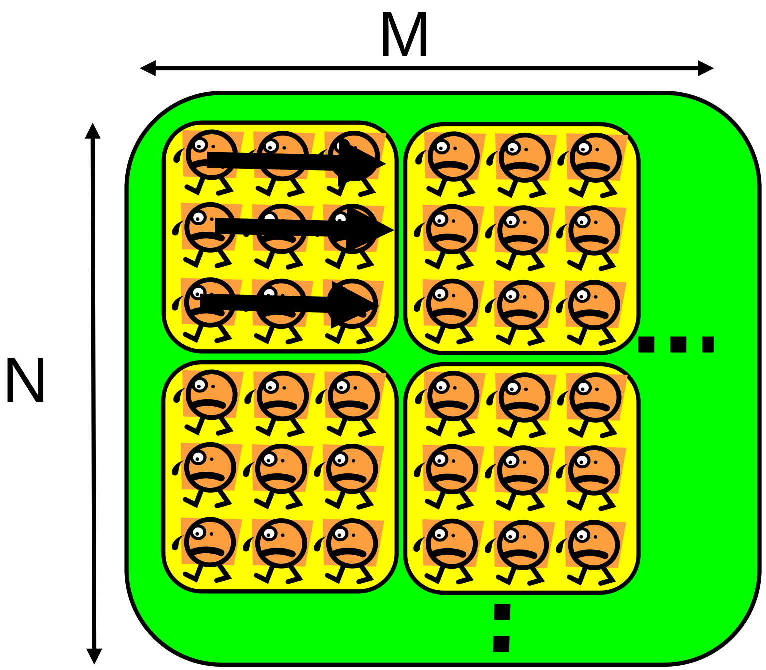
- For the thread with blue mark,

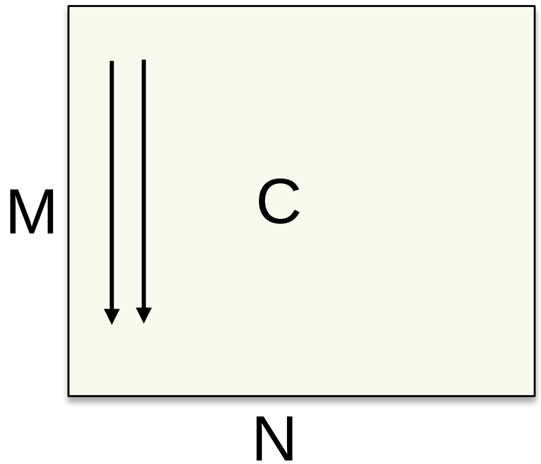
  blockIdx.x=1, blockIdx.y=1, blockIdx.z=0

  threadIdx.x=2, threadIdx.y=0, threadIdx.z=0

Threads in mm-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)
  - gridDim is computed from M, N
- x is mapped to column index, y is mapped to row index (※)





※ reverse mapping is possible, but inefficient (in the next class)

Codes in mm-cuda

gridDim blockDim

```
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}

Limitations on Number of Threads

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

- gx: $\leqq 2^{31}-1$
- gy, gz: $\leqq 65535$
- bx, by: $\leqq 1024$
- bz: $\leqq 64$

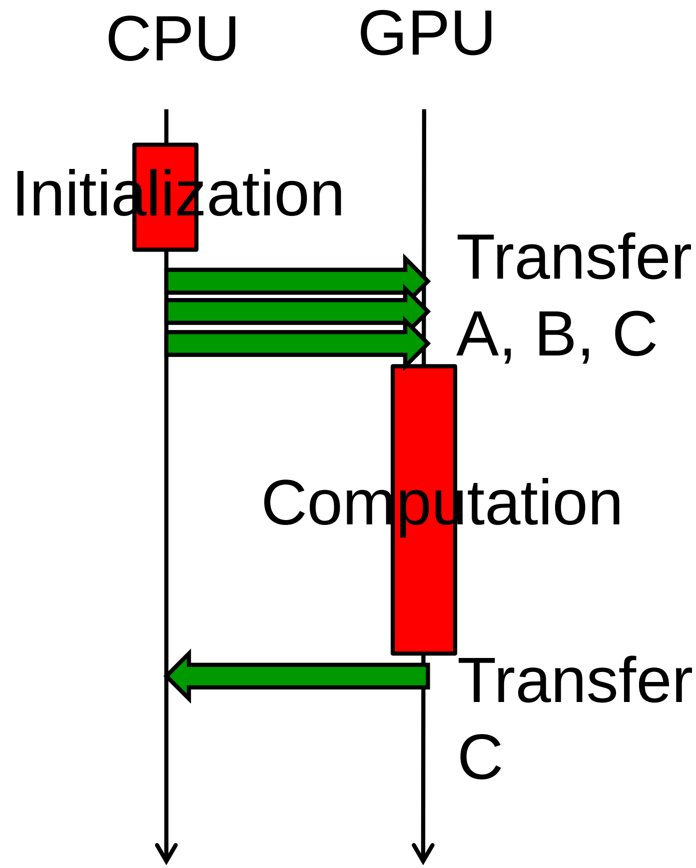
Also, bx*by*bz must be $\leqq 1024$

BlockDim has severe limitation ☹
That is why mm-cuda uses fixed BlockDim (16x16x1)

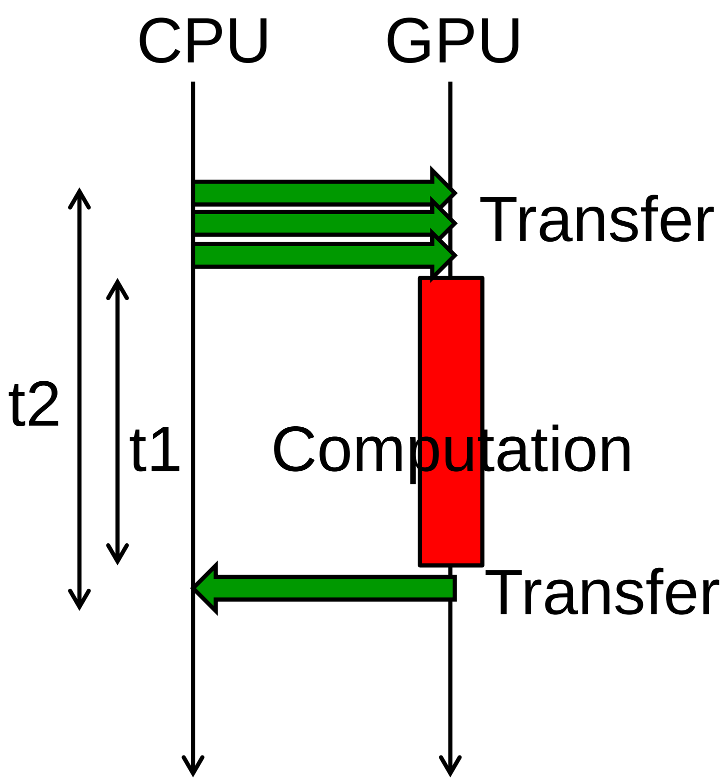
- Limitation depends on GPU types. Refer Appendix G in Programming Guide
  - http://docs.nvidia.com/cuda/
  - K20X has Compute capability 3.5

How We Put Data in mm-cuda Sample

- Consider host memory on CPU and device memory on GPU
- Consideration
  - When computed, all of A, B, C should be on device memory
  - Where are they initialized? CPU or GPU?
  - → In this sample, on CPU
- Current design
  - After initialization of A, B, C, we transfer them from CPU to GPU (by cudaMemcpy)
  - After computation, we transfer C to CPU



Consideration of Computation Speed (related to [G2])



- Computation "speed" is basically (Computation-amount / Time)
- Should "Time" include transfer time?

→ It depends on context. What do we want to measure?

mm-cuda prints both t1 and t2

$t1 \fallingdotseq cMNK$

$t2 \fallingdotseq t1 + d(MK+KN+2MN)$

- c,d is constants that depend on architecture
- This omits "latency"

Notes in Time Measurement

- clock(), gettimeofday() must be called from CPU
- For accurate measurement, we should call cudaDeviceSynchronize() before measurement
  - Actually GPU kernel function call and cudaMemcpy(HostToDevice) are non-blocking
    - “non-blocking” like MPI_Isend, MPI_Irecv

Assignments in this Course

- There is homework for each part. Submissions of reports for 2 parts are required
- Also attendances will be considered

Due Date	Part	Problems	Select
Due Date May 8	Part 1 OpenMP	[O1] diffusion [O2] sort [O3] free	*Select 1 problem*
Due Date May 29	Part 2 MPI	[M1] [M2] [M3]	*Select 1 problem*
Due Date June 12	Part 3 GPU	[G1] [G2] [G3?]	*Select 1 problem*

Select 2 parts

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 (explained later).

[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:

- GPU Programming (3)
  - Optimization techniques in GPU programming
  - Discussion on “diffusion” on CUDA
    - related to assignment [G1]