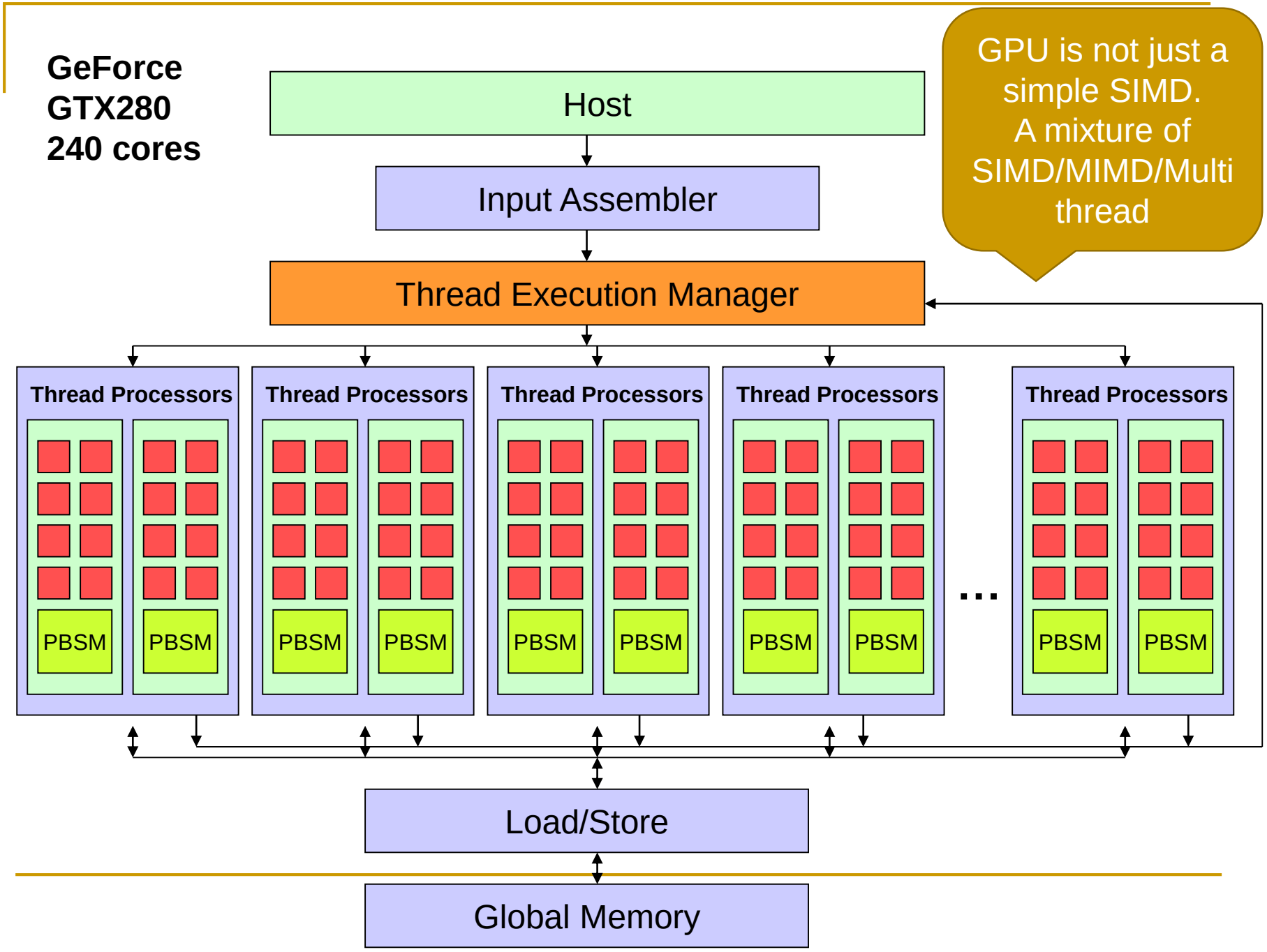
GPUs and their programming

AMANO, Hideharu

GPGPU(General-Purpose computing on Graphic ProcessingUnit)

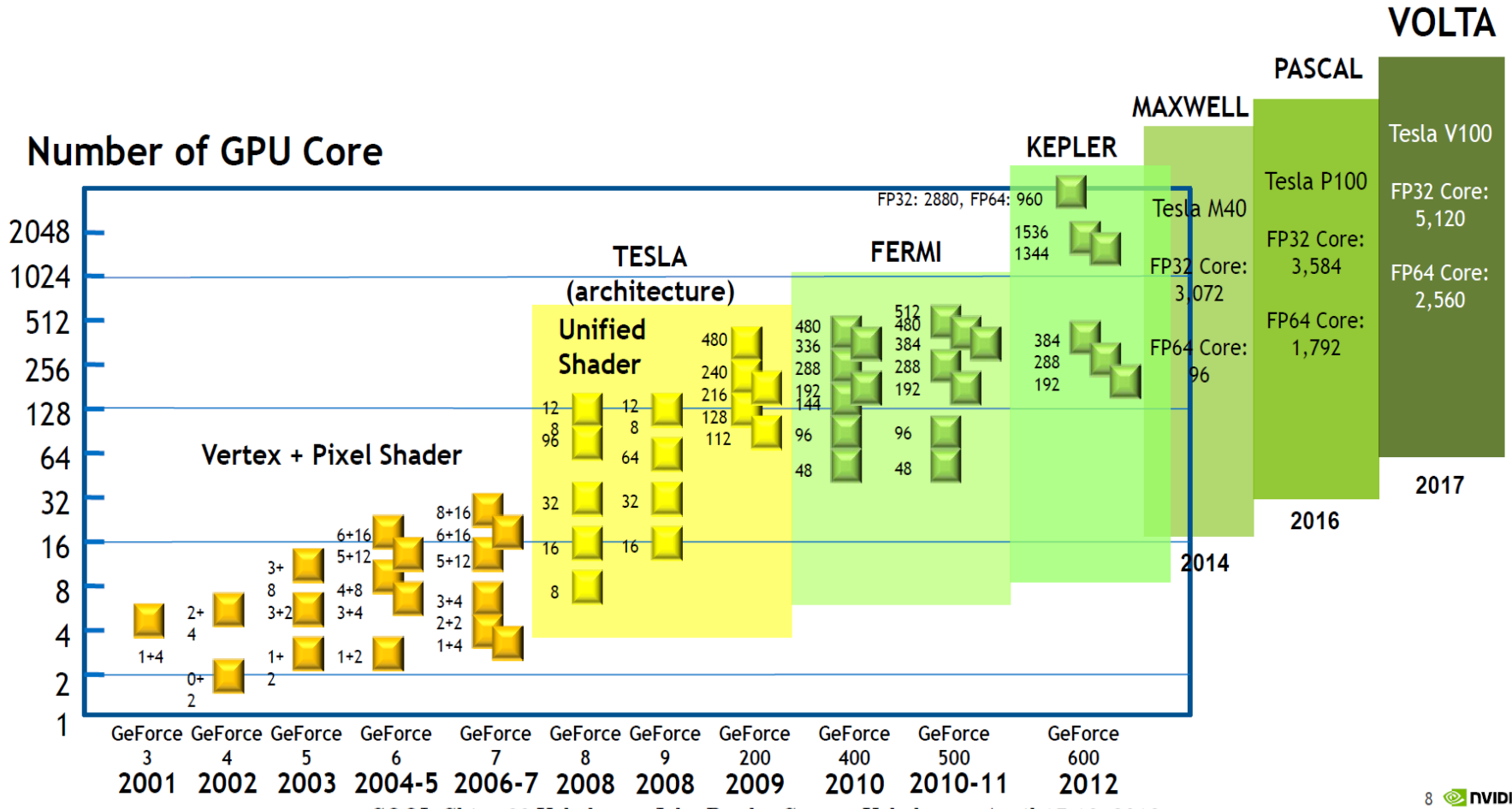
- TSUBAME2.0(Xeon+Tesla,Top500 2010/11 4th)
- 天河一号(Xeon+FireStream,2009/11 5th)


GeForce
GTX280
240 cores
Host
Input Assembler
Thread Execution Manager
GPU is not just a simple SIMD. A mixture of SIMD/MIMD/Multi thread
Thread Processors
PBSM
...
Load/Store
Global Memory

NVIDIA GPU naming rule is confusing

- Target application
  - Dsktop, Game: GeForce（ジーフォース）
    - GeForce GTX＞GeForce GT＞GeForce
    - TITAN X uses Pascal architecture
    - High performance/cost
  - Quadro
    - Graphics for professional use
  - Mobile：Tegra
    - Automatic driving
    - Tegra X1:Maxwell architecture
    - Tegra K1:Keplar architecture
    - Tegra 3,2: only ARM
  - High performace（AI）：Tesla
    - Tesla P100： uses Pascal architecture
    - Tesla V100： uses Volta architecture
- Architecture
  - Fermi, Maxwell, Kepler, Pascal, Volta

GPU ENJOYED THE TR COUNT INCREASE BY MOOR'S LAW, INCREASING ITS NUMBER OF CORES ACCORDINGLY



AMONG TOP10 FASTEST SUPERCOMUTER IN THE WORLD

5 ARE USING NVIDIA GPU ACCELERATION

NO.1 AND 2 ARE USING NVIDIA GPU

ISC2018(INTERNATIONAL SUPERCOMPUTING CONFERENCE) NOV. 2018日

Rank	Name	Site	Country	Accelerator Co-Processor Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	Accelerator Co-Processor
1	Summit	DOE/SC/Oak Ridge National Laboratory	United States	2,196,480	143,500	200,795	NVIDIA Volta GV100
2	Sierra	DOE/NNSA/LLNL	United States	1,382,400	94,640	125,712	NVIDIA Volta GV100
3	Sunway TaihuLight	National Supercomputing Center in Wuxi	China		93,015	125,436	None
4	Tianhe-2A	National Super Computer Center in Guangzhou	China	4,554,752	61,445	100,679	Matrix-2000
5	Piz Daint	Swiss National Supercomputing Centre (CSCS)	Switzerland	319,424	21,230	27,154	NVIDIA Tesla P100
6	Trinity	DOE/NNSA/LANL/SNL	United States		20,159	41,461	None
7	AI Bridging Cloud Infrastructure (ABCI)	産業技術総合研究所 National Institute of Advanced Industrial Science and Technology (AIST)	Japan	348,160	19,880	32,577	NVIDIA Tesla V100 SXM2
8	SuperMUC-NG	Leibniz Rechenzentrum	Germany		19,477	26,874	None
9	Titan	DOE/SC/Oak Ridge National Laboratory	United States	261,632	17,590	27,113	NVIDIA Tesla K20x
10	Sequoia	DOE/NNSA/LLNL	United States		17,173	20,133	None

Screen Clipping

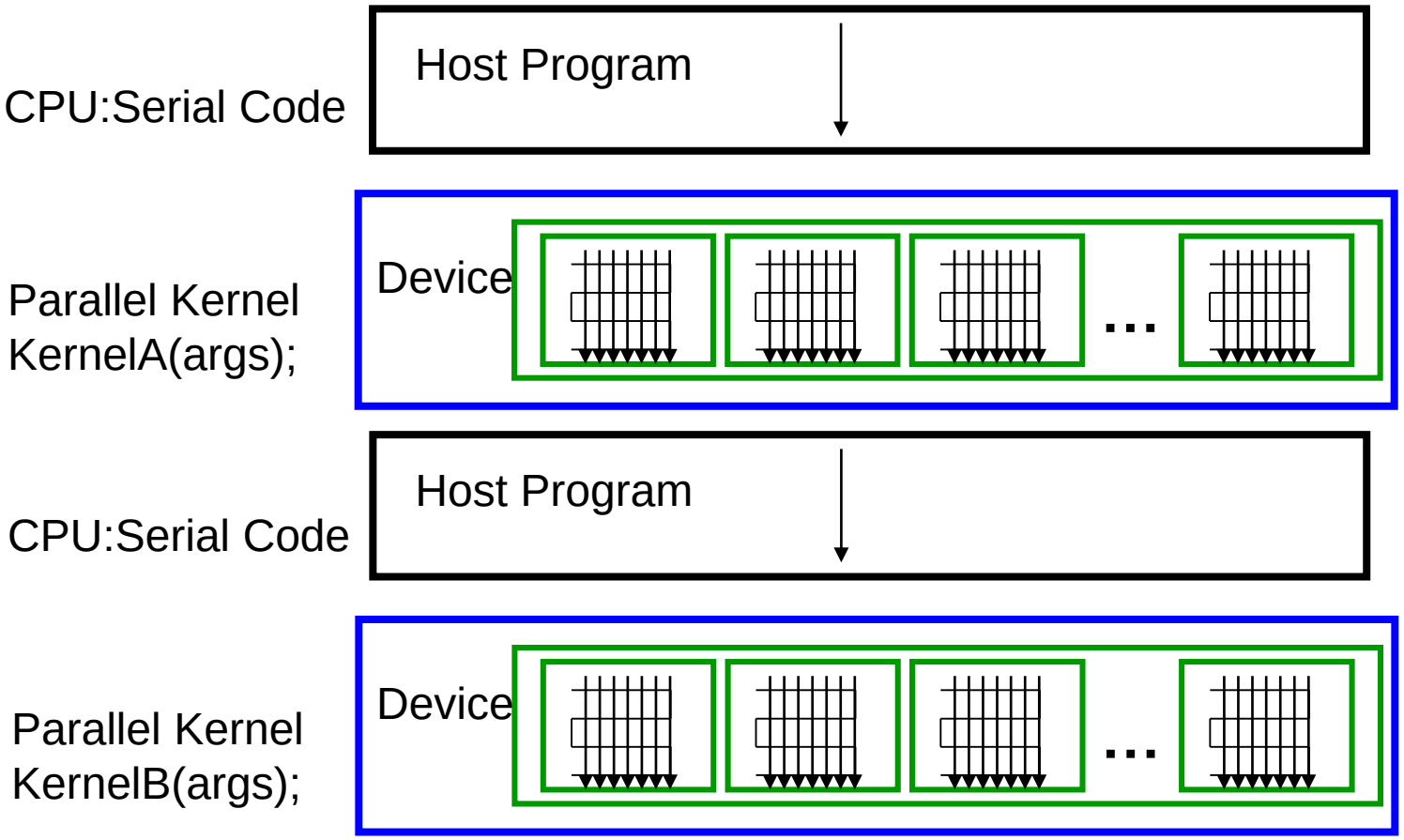
Take a look!

- https://http.download.nvidia.com/developer/cuda/jp/CUDA_Programming_Basics_PartI_jp.pdf
- https://http.download.nvidia.com/developer/cuda/jp/CUDA_Programming_Basics_PartII_jp.pdf

CUDA/OpenCL

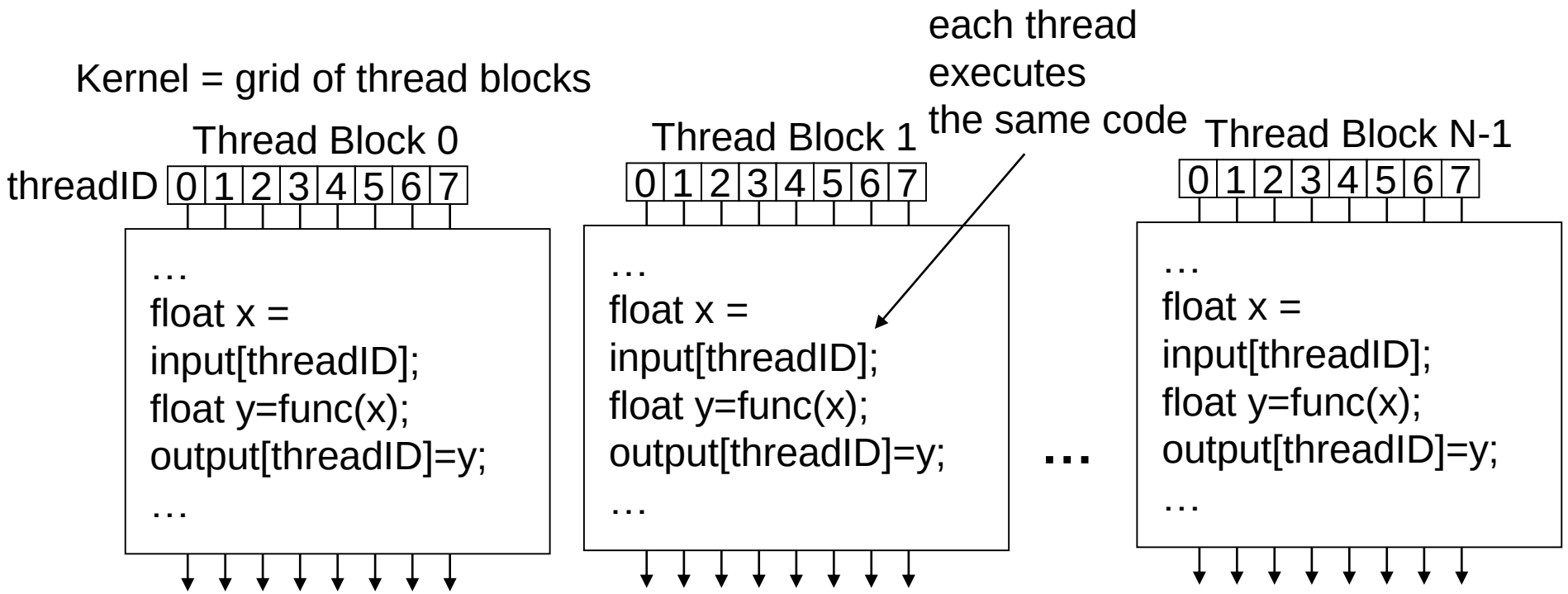
- CUDA is developed for GPGPU programming.
- SPMD(Single Program Multiple Data)
- 3-D management of threads
- 32 threads are managed with a Warp
  - SIMD programming
- Architecture dependent memory model
- OpenCL is standard language for heterogeneous accelerators.

Heterogeneous Programming with CUDA



CPU code is called a host program
GPU is called a device, and its code is called a kernel

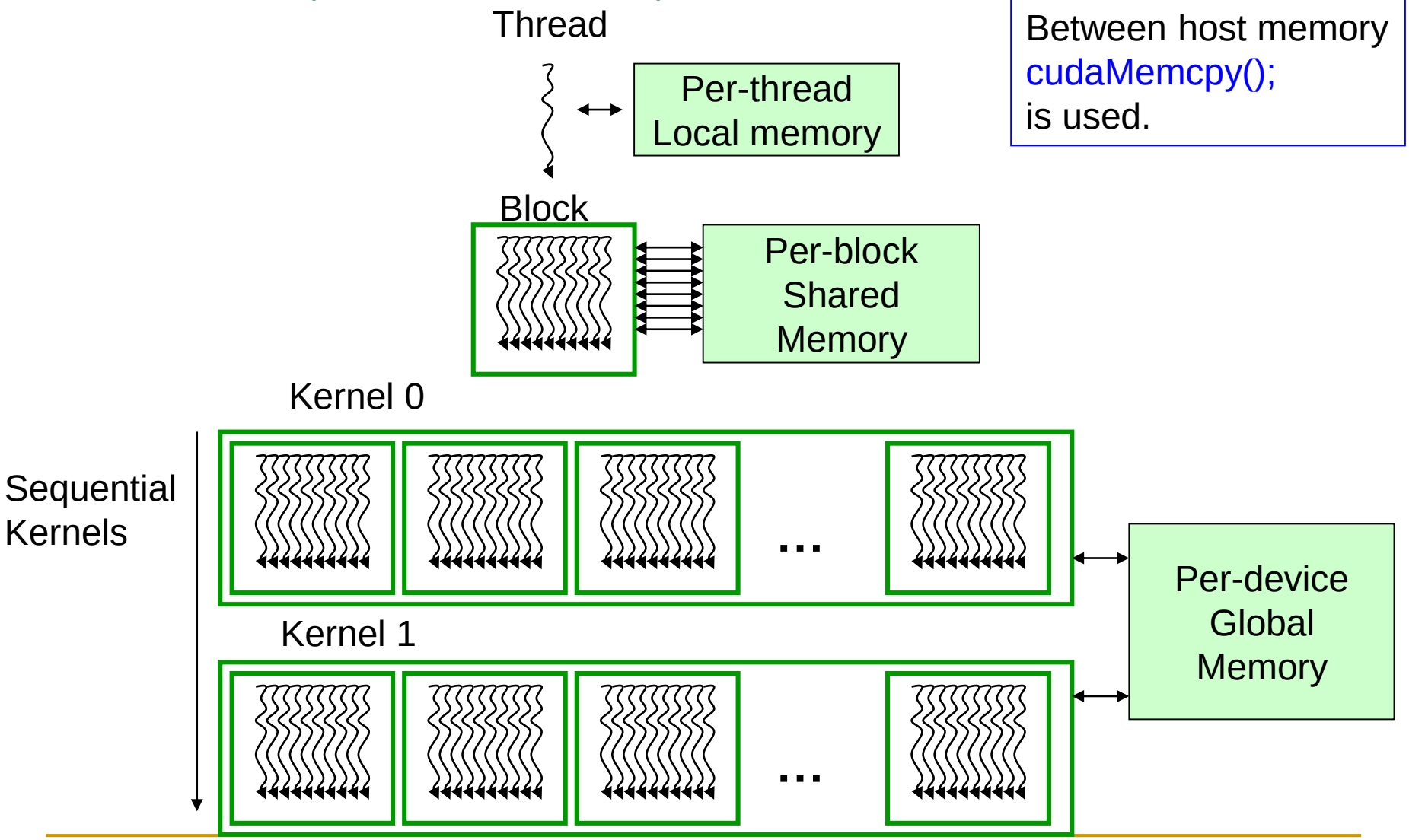
Threads and thread blocks



Threads in the same block may synchronize with barriers.
_syncthreads();
Thread blocks cannot synchronize
-> Execution is depending on machines.

CUDA thread is assigned to each data and handled by its identifier.

Memory Hierarchy



Between host memory
cudaMemcpy();
is used.

Sample program (sample1.cu, sample_kernel1.cu)

- Sum of two arrays of floating point numbers
- Flow of the program:

1. Preparation in the host program
    1. Memory allocation in the device
    2. Data transfer from the host to the device
2. Kernel call
3. Post processing in the host program
    1. Data transfer from the device to the host
    2. Processing in the host
    3. Release the memory in the device

How to login and execute

Get cuda_ex1.tar in your machine.

- Login to comparc{01,02} or nova.
  - ssh exXX@comparc{01,02}.am.ics.keio.ac.jp –XY
  - ssh exXX@nova.am.ics.keio.ac.jp –XY
- File transfer
  - scp cuda_ex1.tar exXX@comparc{01,02}.am.ics.keio.ac.jp:~/.
  - scp exXX@comparc{01,02}.am.ics.keio.ac.jp:~/ex1/ex1_kernel.cu .
- tar xvf cuda_ex1.tar
- cd ex1
- make sample1
  - nvcc sample1.cu sample1_kernel.cu –o sample1
- ./sample1
- make sample1_time generates a version with time measurement.

Target GPU：GeForce GTX790

Architecture：Maxwell
Cuda core：1660
Core clock：1050MHz
GPU memory：4GB

host: sample.cu
Initialization in the host

```
#include <stdio.h>
#include <stdlib.h>
#include "header.h"  // Library files
int main(int argc, char **argv) {
float *h_A, *h_B, *h_C; // variables in the host
float *d_A, *d_B, *d_C; // variables in the device
float result = 0.0f; // results
dim3 dim_grid(LENGTH/BLOCK_SIZE, 1, 1); // For kernel call
dim3 dim_block(BLOCK_SIZE, 1, 1); //
//  Allocation in the host memory and Generation of array
h_A = (float *)malloc(sizeof(float) * LENGTH);
h_B = (float *)malloc(sizeof(float) * LENGTH);
h_C = (float *)malloc(sizeof(float) * LENGTH);
for (int i = 0; i < LENGTH; ++i) {
h_A[i] = 1.0f; h_B[i] = 2.0f; h_C[i] = 0.0f;  }
```

Memory Allocation and Copy

```
// Allocation in the device
cudaMalloc((void **)&d_A, sizeof(float) * LENGTH);
cudaMalloc((void **)&d_B, sizeof(float) * LENGTH);
cudaMalloc((void **)&d_C, sizeof(float) * LENGTH);
// Copy data to the device
cudaMemcpy(d_A, h_A, sizeof(float) * LENGTH,
                    cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, sizeof(float) * LENGTH,
                     cudaMemcpyHostToDevice);

/* cudaMemcpy(void *dst, const void *src, size_t count,
           cudaMemcpyKind kind)
kind: cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost,...
*/
```

Kernel Call (host: sample1.cu)

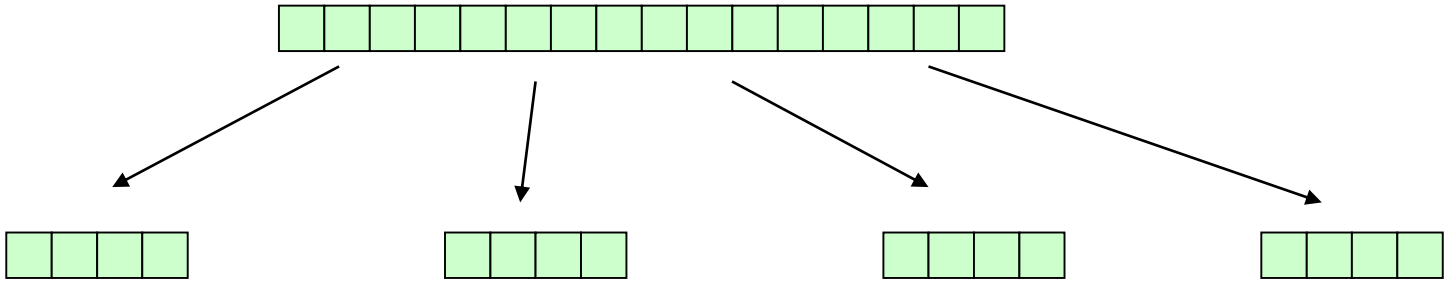
```
dim3 dim_grid(LENGTH/BLOCK_SIZE, 1, 1); // For kernel call
dim3 dim_block(BLOCK_SIZE, 1, 1); //
…..
Sample1Kernel<<<dim_grid, dim_block>>>(d_A, d_B, d_C);
```

dim3 definition and

<<<… >>> is a type of CUDA extension.

Kernel Call

Let's assume LENGTH=16, BLOCK_SIZE=4



blockIdx.x=0
blockDim.x=4
threadIdx.x=0,1,2,3
idx=0,1,2,3

blockIdx.x=1
blockDim.x=4
threadIdx.x=0,1,2,3
idx=4,5,6,7

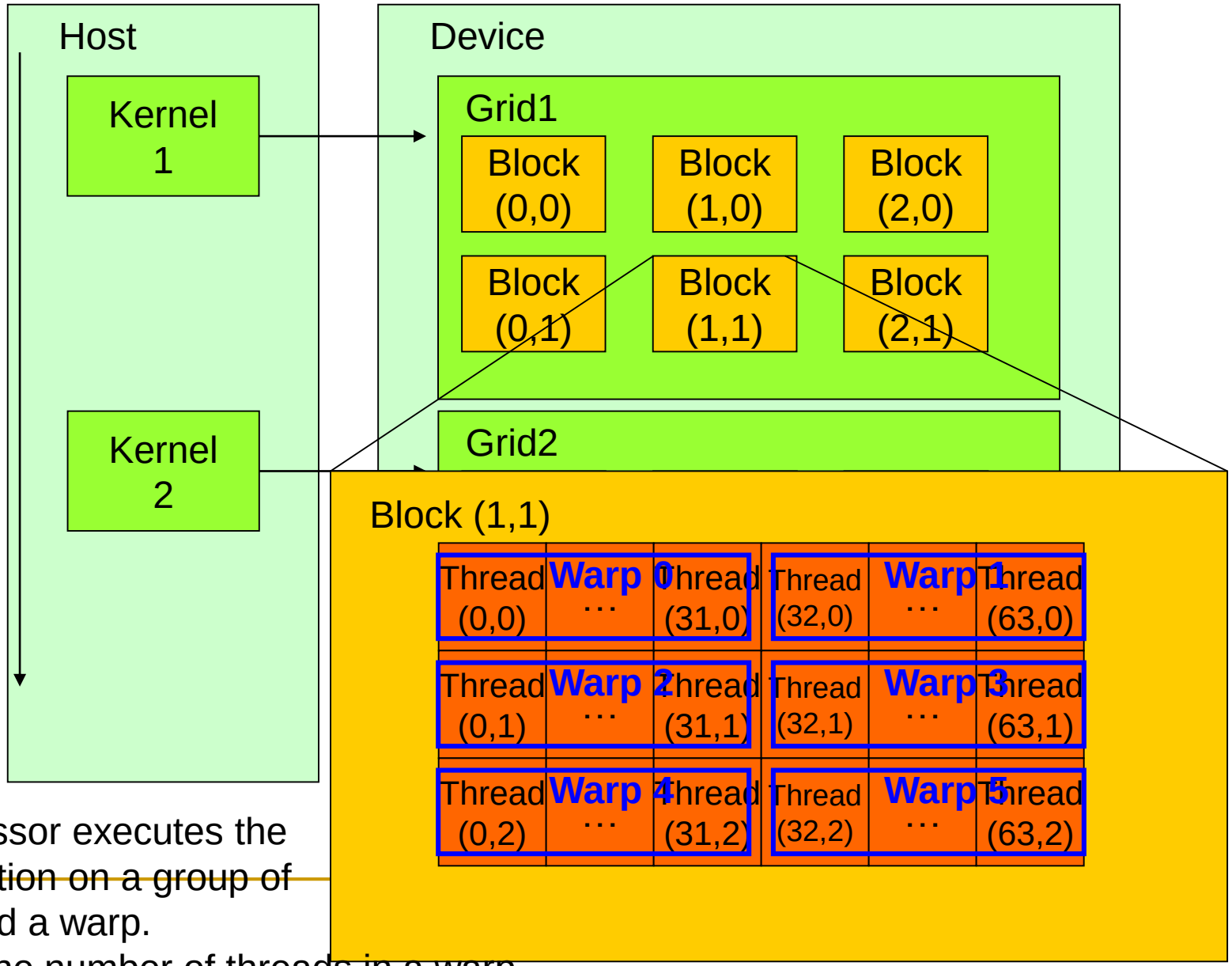
blockIdx.x=2
blockDim.x=4
threadIdx.x=0,1,2,3
idx=8,9,10,11

blockIdx.x=3
blockDim.x=4
threadIdx.x=0,1,2,3
idx=12,13,14,15

int idx = blockDim.x * blockId.x + threadIdx.x;
will map from local index threadIdx to global index.

blockDim should be >= 32 in real code!
Using more number of blocks hides the memory latency in GPU.

Hardware Implementation: Execution Model



A multiprocessor executes the same instruction on a group of threads called a warp.

Warp size= the number of threads in a warp

Kernel: sample1_kernel.cu

```
__global__ void Sample1Kernel(float *d_A, float *d_B,
                              float *d_C) {
// Getting its thread id
int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
// Compute sum of array
d_C[thread_id] = d_A[thread_id] + d_B[thread_id];
}
```

A loop structure can be done with threads in parallel.

CUDA runtime

- Device menagement:
  - cudaGetDeviceCount(), cudaGetDeviceProperties();
- Device memory management:
  - cudaMalloc(), cudaFree(),cudaMemcpy()
- Graphics interoperability:
  - cudaGLMapBufferObject(), cudaD3D9MapResources()
- Texture management:
  - cudaBindTexture(), cudaBindTextureToArray()

Post processing (host: sample1.cu)

```
// Copy results from memory to host
cudaMemcpy(h_C, d_C, sizeof(float) * LENGTH,
cudaMemcpyDeviceToHost);
// Release device memory
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
// Print the result
for (int i = 0; i < LENGTH; ++i) result += h_C[i];
result /= (float)LENGTH;
printf("result = %f¥n", result);
// Finalize
free(h_A); free(h_B); free(h_C);
return 0;
}
```

CUDA extensions

- Declaration specifiers :
  - _global_ void kernelFunc(…); // kernel function, runs on device
  - _device_ int GlobalVar; //variable in device memory
  - _shared_ int sharedVar; //variable in per-block shared memory
- Extend function invocation syntax for paralell kernel launch
  - KernelFunc<<<dimGrid, dimBlock>>> // launch dimGrid blocks with dimBlock threads each
- Special variables for thread identification in kernels
  - dim3 threadIDx; dim3 blockIdx; dim3 block Dim; dim3 gridDim;
- Barrier Synchronization between threads
  - _syncthreads();

Example: ex1

- A[i] and B[i] are vectors with 256x256(LENGTH) elements.
- Write the kernel code in ex1_kernel.cu.

```
for (i=0; i<LENGTH; i++)
        C[i]=0.0;
      for(j=0; j<LENGTH; j++)
        C[i] += (A[i]-B[j])*(A[i]-B[j]);
```

Execute ex1.cu by removing a comment for ex1Kernel. The answer must match to C version.