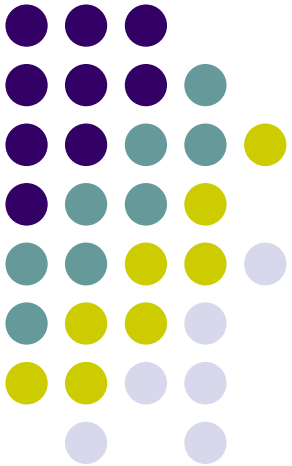
2018
Practical Parallel Computing (実践的並列コンピューティング) No. 11

GPU Programming (1)

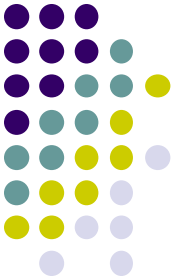


Toshio Endo

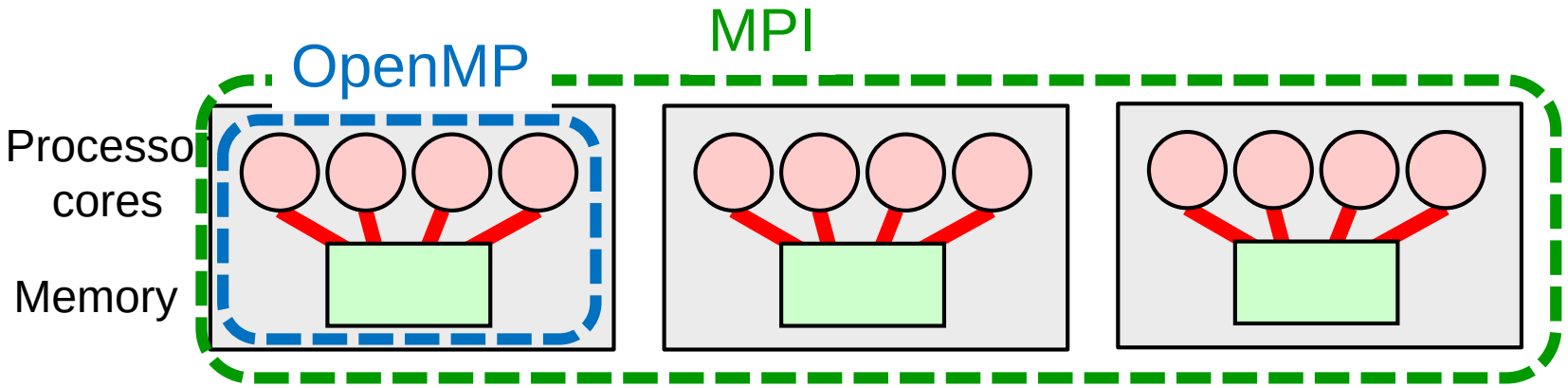
School of Computing & GSIC

endo@is.titech.ac.jp

Parallel Programming using CPUs



- Both OpenMP and MPI uses multiple processor cores in CPUs
  - OpenMP: cores in a single node
  - MPI: we can use cores in multiple nodes



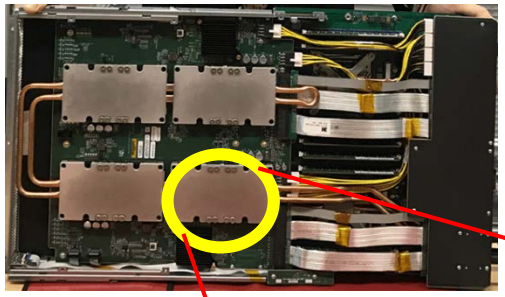
In Part 3, we use other processors than CPUs → GPU

GPU Computing

- Graphic processing units (GPU) have been originally used for computing graphics (including video games)
- A GPU has many (simple) cores
  - CPU: 2 to 32 cores. GPU: >1000 cores

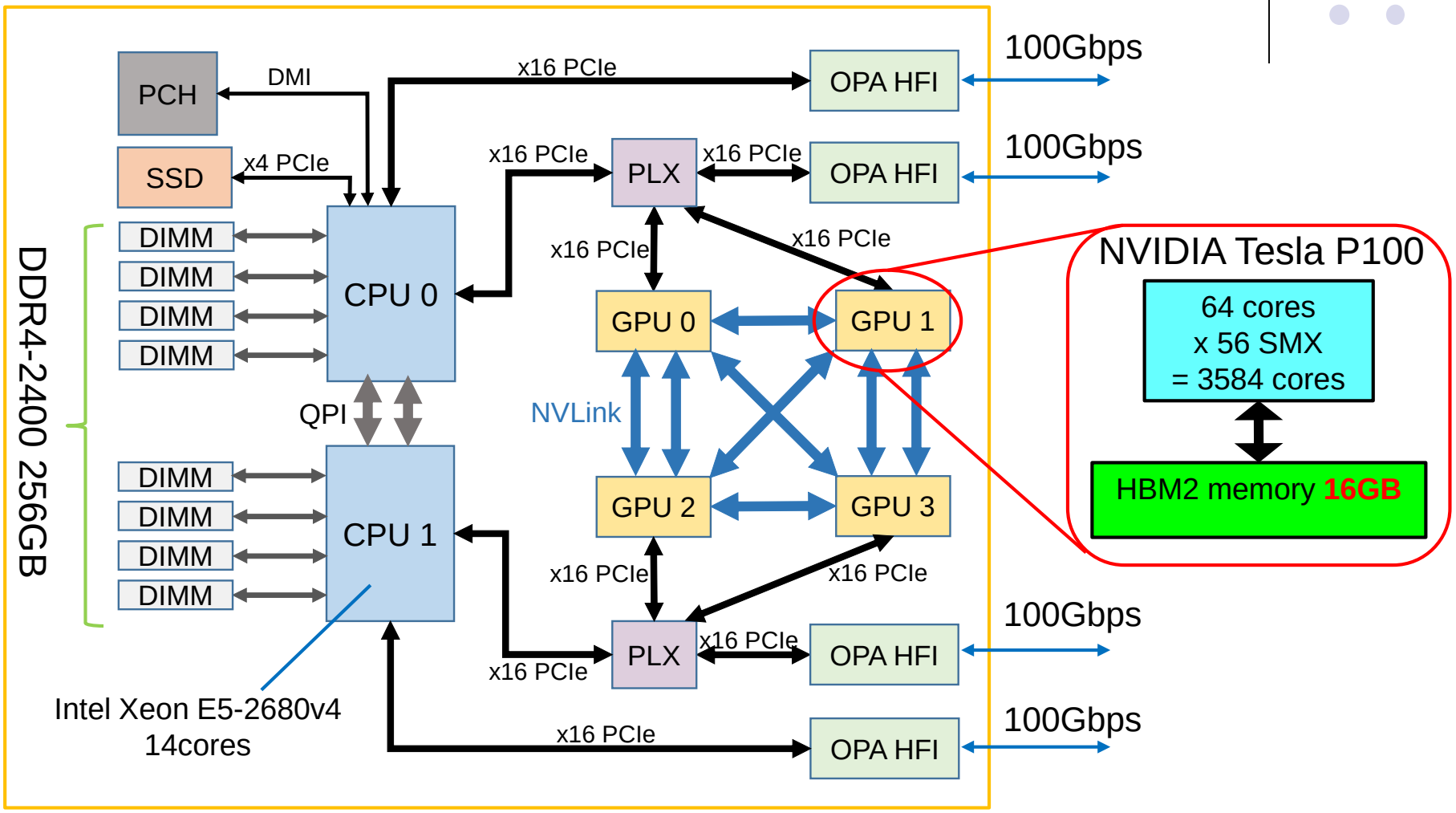
→ Recent GPUs can be used for general applications!

  - The concept is called GPGPU (General-Purpose computing on GPU)
- Became popular since NVIDIA invented CUDA language in 2007

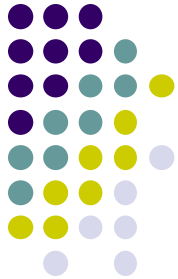


TSUBAME3 node

A TSUBAME3 Node with GPUs



Characteristics of GPUs (1)



- A GPU is a board or a card attached to computers

  It cannot work alone. Driven by CPUs

Comparing Xeon E5-2680 v4 (TSUBAME3's CPU) and Tesla P100 (TSUBAME3's GPU)

	1 CPU		1 GPU
Number of cores	14	<<<	3584 CUDA cores (=64 x 56SMXs)
Clock Frequency	2.4GHz (AVX clock 1.9GHz)	>	1.48GHz
Peak Computation Speed (double precision)	425GFlops	<<	5300GFlops
Memory Capacity	128GB (256GB shared by 2CPUs)	>>	16GB

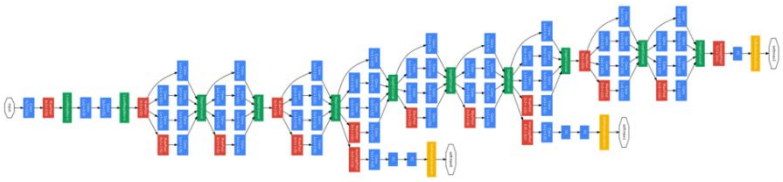
Characteristics of GPUs (2)

- CUDA cores are not perfectly independent with each other

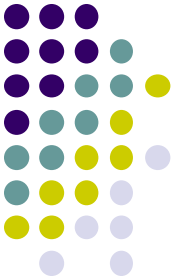
- Irregular algorithms are harder to implement ☹
  - Searching, algorithms with trees…
- Appropriate for algorithms with regular structures ☺
  - Matrix(linear) computations, stencil computations…

- Why are GPUs very popular in deep learning?
  - Convolution computation ⇔ Stencil computation
  - Fully connected layer ⇔ Linear algebra computation

TensorFlow Chainer Caffe2 PYTORCH

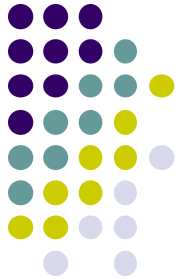


Several Groups of GPUs (and GPU-like Processors)



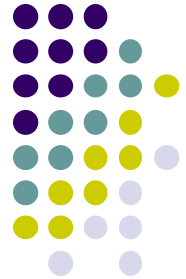
- NVIDIA GPUs: most popular
  - Tesla series, GeForce series…
  - Programming with CUDA, OpenACC, OpenCL etc
- AMD (ATI) GPUs
  - Radeon series, Firepro series…
  - Programming with OpenCL etc
- Intel Xeon Phi
  - 64-72 cores, 1.3-1.5GHz (Knights Landing series)
  - Programming with OpenMP, MPI…
  - OS directly works on Xeon Phi!
- In old days, 9-core Cell processors surprised the world (PlayStation3 in 2006)

Programming Environments for GPUs



- CUDA
  - Most popular and suitable for higher performance
  - Regions executed by GPU must be functions (called kernel functions)
  - Use “nvcc” command for compile
    - module load cuda
    - nvcc … XXX.cu
- OpenACC
  - C/Fortran + directives (#pragma acc …), Easier programming
  - I recommend PGI compiler
    - module load pgi
    - pgcc –acc … XXX.c
  - Basically for data parallel programs with for-loops
- OpenMP 4.0, OpenCL…

OpenACC Programs Look Like



C/C++/Fortran + directives

```
   int a[100], b[100], c[100];
   int i;
#pragma acc data copy(a,b,c)
#pragma acc kernels
#pragma acc loop independent
  for (i = 0; i < 100; i++) {
       a[i] = b[i]+c[i];
  }
```

Examples of OpenACC *directives*

In this case, each directive has an effect on the following block/sentence

“mm” sample: Matrix Multiply

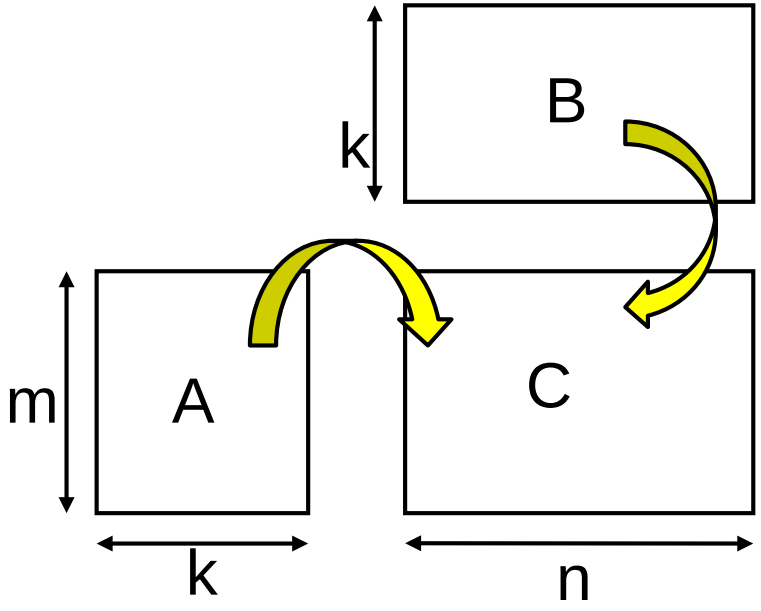
Available at ~endo-t-ac/ppcomp/18/mm-acc/

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

C: a (m × n) matrix

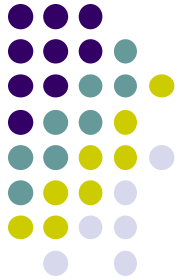
C ← A × B

- Algorithm with a triple for loop
- Supports variable matrix size.
  - Each matrix is expressed as a 1D array by *column-major* format



- Execution: ./mm [m] [n] [k]

Compiling OpenACC Programs



Not so popular as OpenMP, unfortunately☹

- PGI compiler
  - module load pgi, and then use pgcc
  - -acc option in compiling and linking
  - -Minfo option outputs many information on parallelization

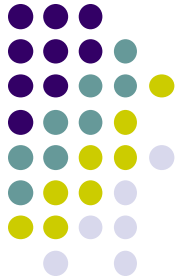
Example of output

```
          :
   47, Generating copyin(A[:m*k])
       Generating copy(C[:m*n])
       Generating copyin(B[:k*n])
   50, Loop is parallelizable
          :
```

- Very new gcc supports OpenACC (gcc 6 or later), but not available on TSUBAME3 ☹

Also see outputs of “make” in sample directory

Submitting a GPU Job



- Sequential version
  - see mm directory
- OpenACC version
  - see mm-acc directory
  - To use a GPU, use q_node type
  - (h_node or f_node types for multi-GPU)

mm/job.sh

```
#!/bin/sh
#$ -cwd
#$ -l s_core=1
#$ -l h_rt=00:10:00

./mm 1000 1000 1000
```

resource type and count

maximum run time

mm-acc/job.sh

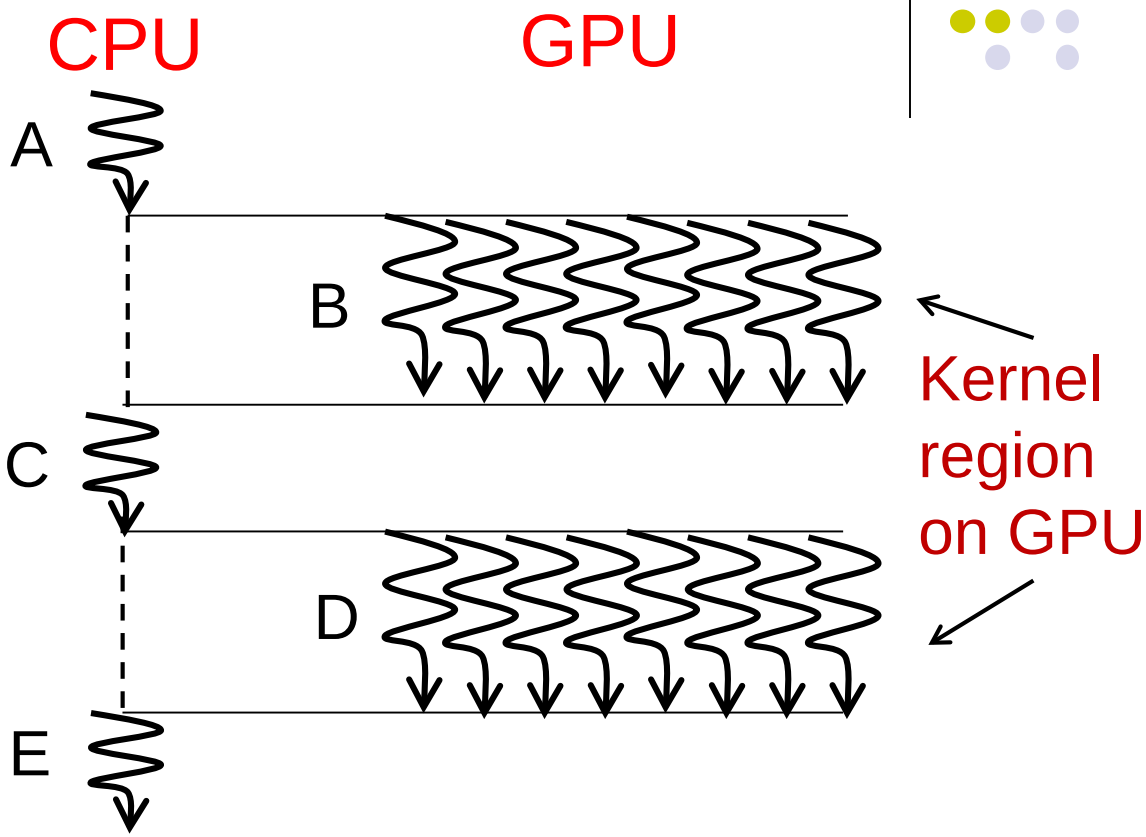
```
#!/bin/sh
#$ -cwd
#$ -l q_node=1
#$ -l h_rt=00:10:00

./mm 1000 1000 1000
```

- Job submission
  - qsub job.sh

Kernel Region in OpenACC

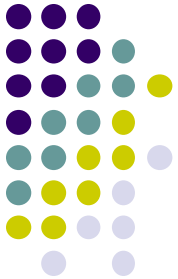
```
int main()
{
    A;
#pragma acc kernels
    {
        B;
    }
    C;
#pragma acc kernels
    D;
    E;
}
```



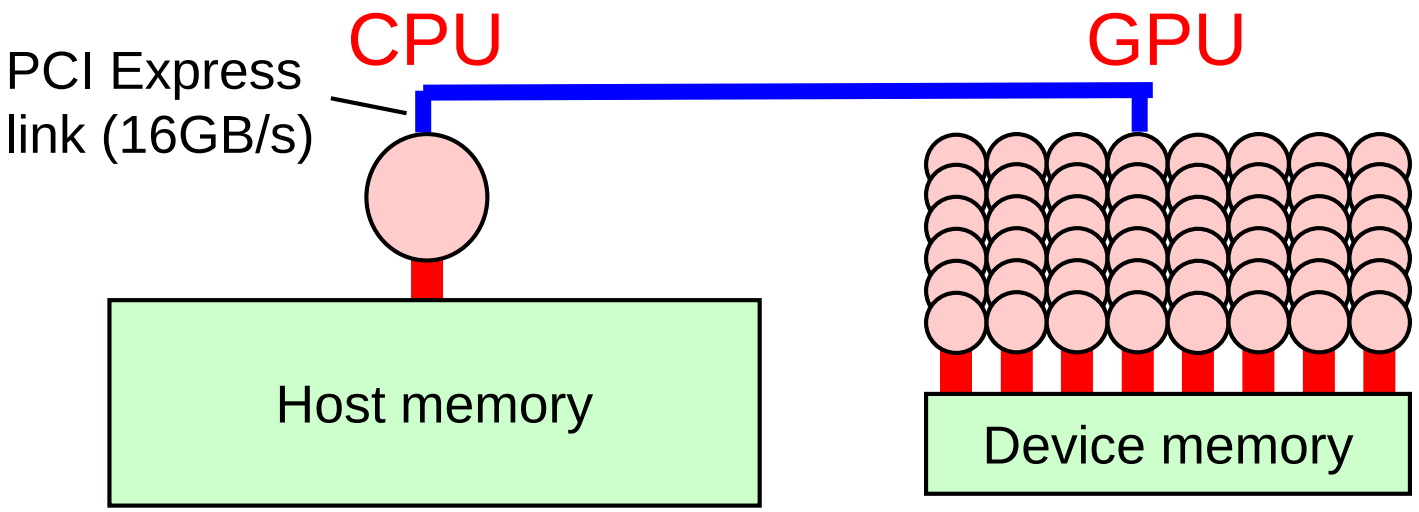
A sentence/block immediately after #pragma acc kernels is called a kernel region, executed on GPU

- We don't need to specify number of threads (we also can)
- Also #pragma acc parallel works similarly (not same)

Data Movement between CPU and GPU



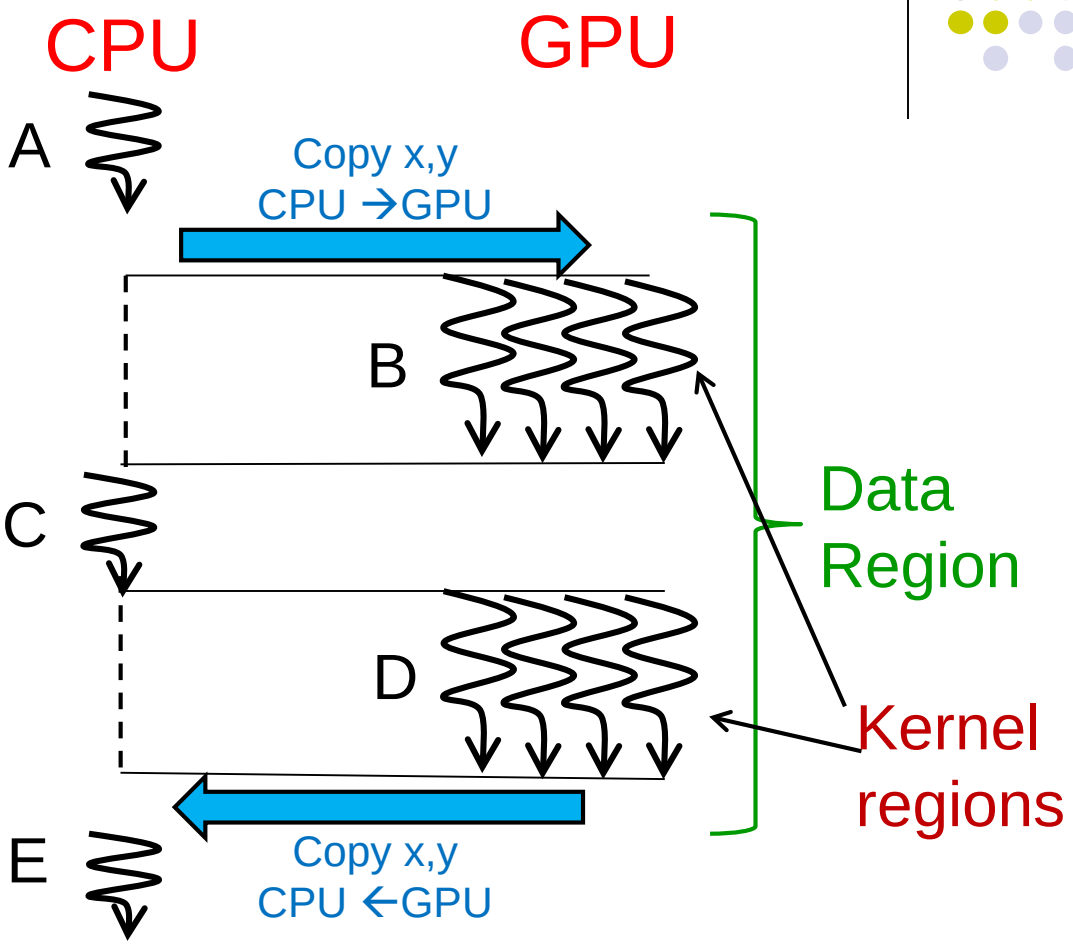
- We need to move data between CPU and GPU
  - Host (CPU) memory and Device (GPU) memory are distinct, like distributed memory
  - Threads on a GPU share the device memory



For this purpose, we use #pragma acc data directive
→ Data region

Data Directives to use GPU memory

```
int main()
{
    A;
#pragma acc data copy(x,y)
    {
#pragma acc kernels
        {
            B;
        }
        C;
#pragma acc kernels
        D;
    }
    E;
}
```



- Data region may contain 1 or more kernel regions
- Data movement occurs at beginning and end of data region

Data Directive

- Arrays:
  - we can write array names if the sizes are statically declared → entire array is copied
- Pointers as arrays:

  cf) b [0 : 20]

  start index　number of elements
  - Partial copying like b[10:5] or a[4:4] are ok

```
int x;
float a[10];
double *b = (double*)
    malloc(20*sizeof(double));
      :
#pragma acc data copy(x, a, b[0:20])
      :
```

- One direction or both directions?
  - … data copyout(…): Copy CPU→GPU at the begininng
  - … data copyin(…): Copy CPU←GPU at the end
  - … data copy(…): Do both

Optimization of data movement will help speedup

Loop Directive

```
   int a[100], b[100], c[100];
   int i;
#pragma acc data copy(a,b,c)
#pragma acc kernels
#pragma acc loop independent
  for (i = 0; i < 100; i++) {
       a[i] = b[i]+c[i];
  }
```

- #pragma acc loop must be included in "acc kernels" or "acc parallel"
- Directly followed by "for" loop
  - The loop must have a loop counter, as in OpenMP
  - List/tree traversal is NG

- … loop independent: Iterations are done in parallel by multiple GPU threads
- … loop seq: Done sequentially. Not be parallelized
- … loop: Compiler decides

OpenACC Version of mm (mm-acc/mm.c)

```
#pragma acc data copyin(A[0:m*k],B[0:k*n]),copy(C[0:m*n])
#pragma acc loop kernels
#pragma acc loop independent
  for (j = 0; j < n; j++) {
#pragma acc loop independent
    for (i = 0; i < m; i++) {
#pragma acc loop seq
      for (l = 0; l < k; l++) {
        Ci,j += Ai,l * Bl,j;
      } } }
```

We can omit CPU←GPU copy of A,B

←For each column in C

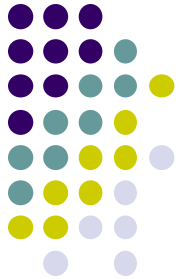
←For each row in C

←For dot product

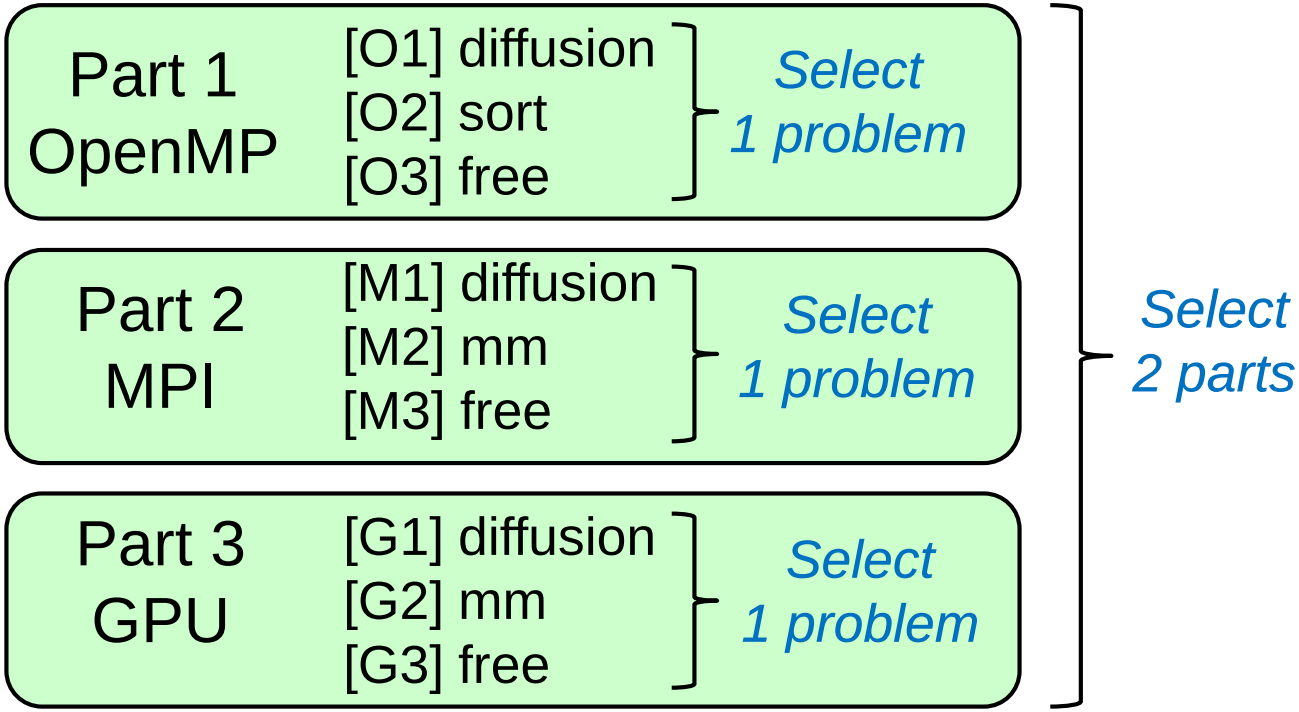
- Each element in C can be computed in parallel (i-loop, j-loop)
- Computation of a single C element is sequential (L-loop)

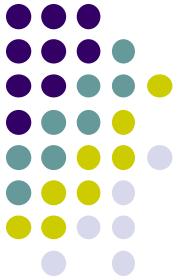
- mm-acc/mm.c includes JLI version and JIL version
  - Both have same computation amount. How are speeds?

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 (1)

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

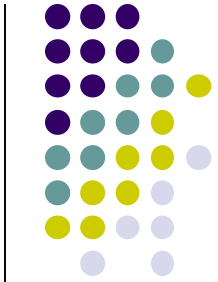
Due date: June 14 (Thursday)

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

Optional:

- To make array sizes variable parameters
- To compare OpenACC vs CUDA
- To improve performance further
  - Different assignment of threads and elements (CUDA)
  - Different num_gang, vector_length, etc (OpenACC)
  - etc.

Assignments in GPU Part(2)



[G2] Improve “mm-acc” or “mm-cuda” to support larger matrices

- In original version, “./mm 2000 500000 2000” is ok, but “./mm 2000 600000 2000” outputs an error

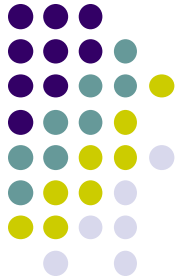
→ Fix this problem

- Consider large n. Small m and k are ok
  - Note: Be careful for too large n. We cannot surpass host memory

Optional:

- To overlap data copy and computation
- To support large m, n, k
- To try “CUDA unified memory”

Assignments in GPU Part (3)



[G3] (Freestyle) Parallelize *any* program by OpenACC or CUDA.

- cf) A problem related to your research
- “pi” sample?
  - Using random number on OpenACC/CUDA is not easy
- “sort” sample on GPU?
  - Other algorithms than quick sort may be appropriate
- More challenging one for parallelization is better
  - cf) Partial computations have dependency with each other

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

Next Class:

- GPU Programming (2)
  - Improving data copy
  - Improving loop parallelization