# Practical Parallel Computing (実践的並列コンピューティング)

Part 2: GPU

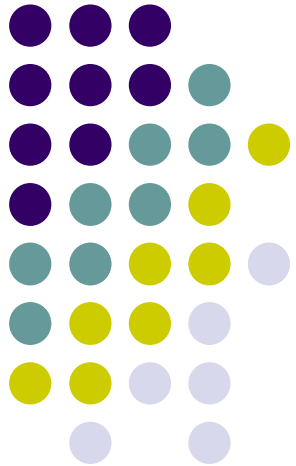
No 1: Overview and OpenACC

May 1, 2023

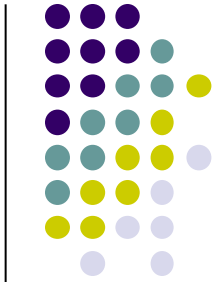
Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp

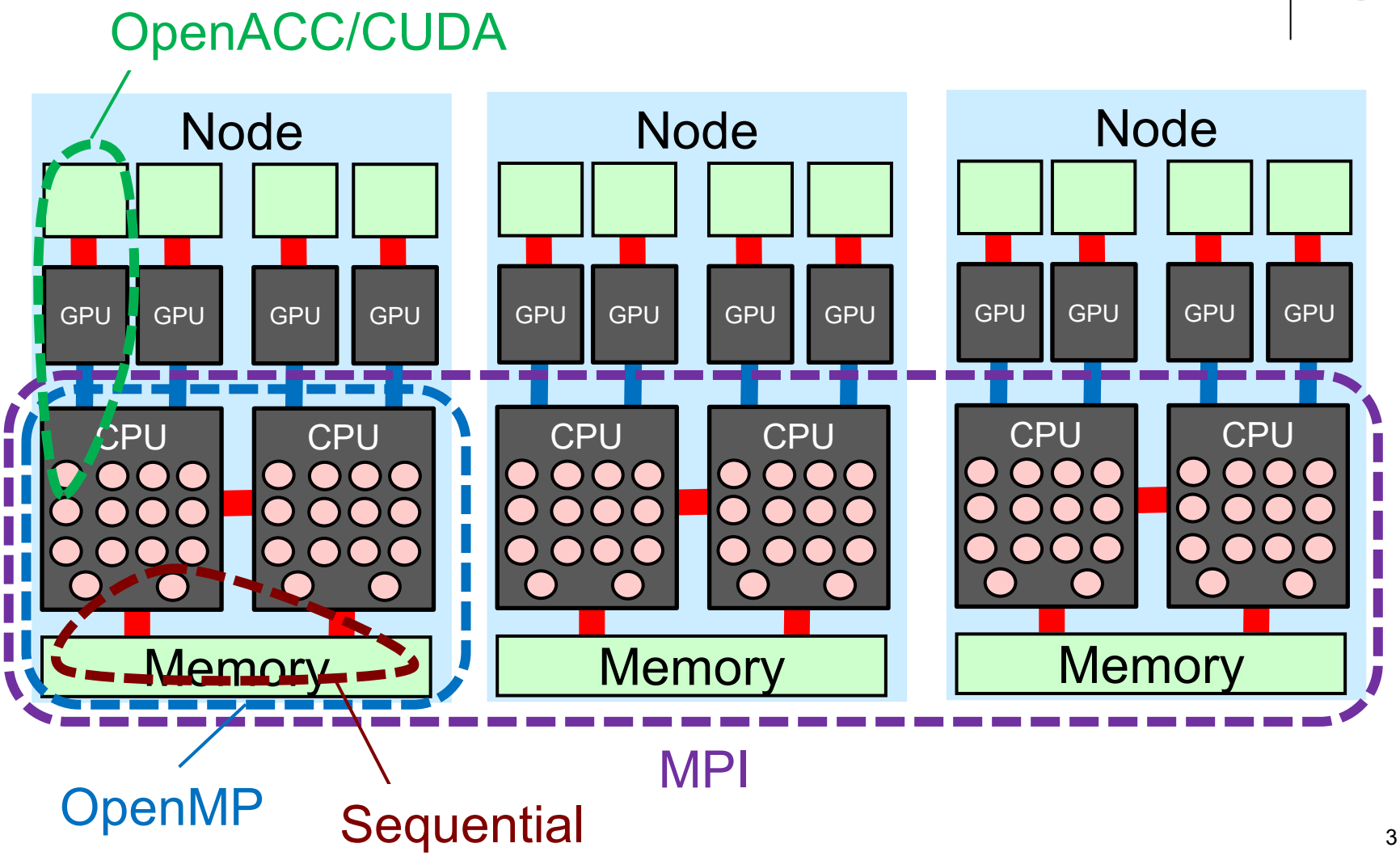


# Overview of This Course



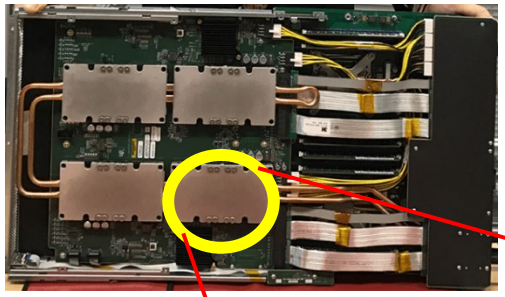
- Part 0: Introduction
  - 2 classes
- Part 1: OpenMP for shared memory programming
  - 4 classes
- Part 2: GPU programming
  - 4 classes ← We are here (1/4)
  - OpenACC (1.5 classes) and CUDA (2.5 classes)
- Part 3: MPI for distributed memory programming
  - 3 classes

# Parallel Programming Methods on TSUBAME



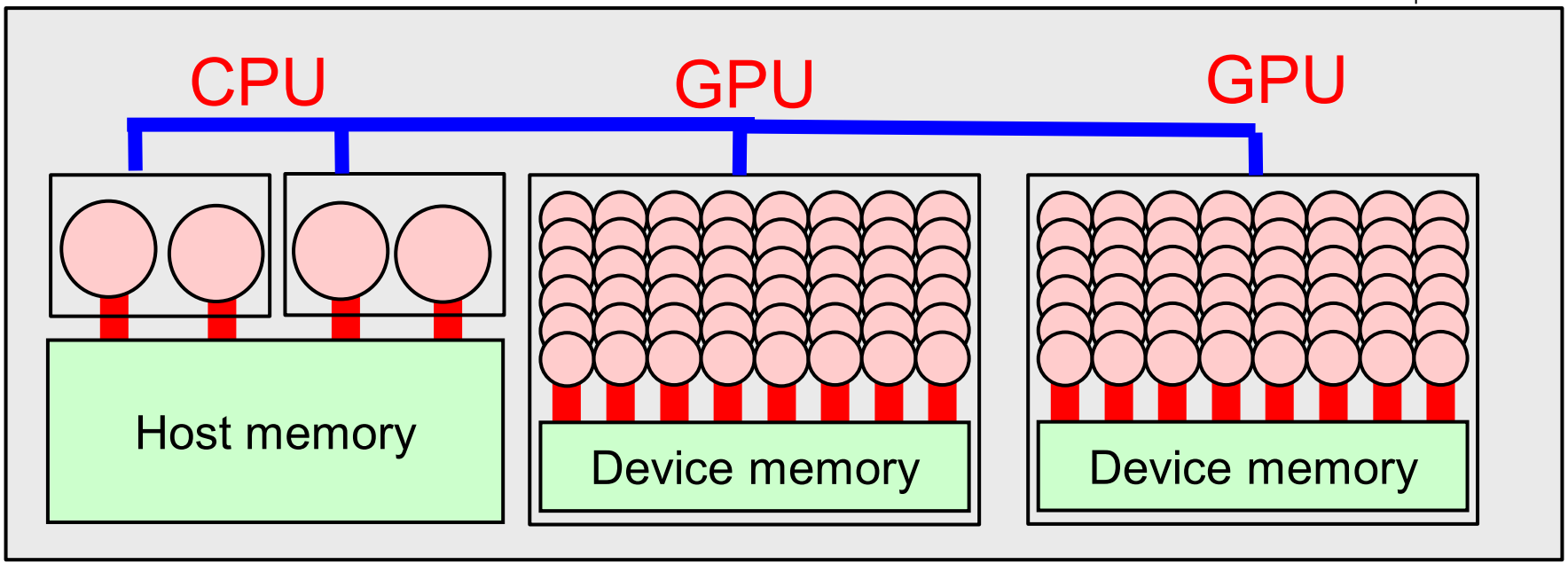
# GPU Computing

- Graphic processing units (GPU) have been originally used for computing graphics (including video games)
- A high performance GPU has many cores
  - CPU: 2 to 32 cores. GPU: >1000 cores
  - The concept is called GPGPU (General-Purpose computing on GPU)
- GPGPU became popular since NVIDIA invented CUDA language in 2007
  - Recently it is popular for deep learning

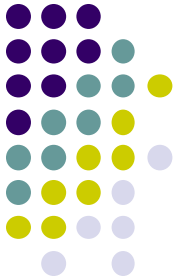


TSUBAME3 node

# A Compute Node with GPU




- A GPU has its distinct memory (device memory)
  - CPU memory is called host memory
- Many cores in a GPU share its device memory



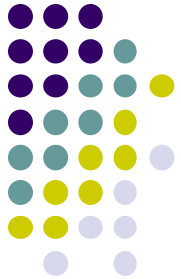
# Characteristics of GPUs

A GPU is a board or a card attached to computers
→ It cannot work alone. Driven by CPUs
→ Different programming methods

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

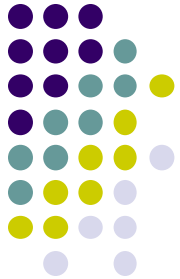
| | 1 CPU | | 1 GPU |
|---|---|---|---|
| Number of cores | 14 cores<br>(28 cores with 2CPUs) | <<< | 3584 CUDA cores<br>(=64 x 56SMXs) |
| Clock Frequency | 2.4GHz | > | 1.48GHz |
| Peak Computation Speed (double precision) | 425GFlops | << | 5300GFlops |
| Memory Capacity | 128GB<br>(256GB shared by 2CPUs) | >> | 16GB |

# Programming Environments for NVIDIA GPUs



- CUDA ← We will use after OpenACC
  - The most famous environment, designed by NVIDIA
  - C/Fortran + new syntaxes
  - Use “nvcc” command for compile
    - module load cuda
    - nvcc … XXX.cu
  - For more general programs than OpenACC ☺
- OpenACC ← Today’s topic
  - C/Fortran + directives (#pragma acc …), Easier programming ☺
  - Supported by NVIDIA HPC SDK
    - module load nvhpc
    - pgcc -acc … XXX.c
  - For parallel programs with for-loops
- OpenMP 5, OpenCL…

# An OpenACC Program Looks 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

OpenACC is not so popular as OpenMP, unfortunately☹

- gcc 4.8.5 (TSUBAME's default compiler) does not support it
- On TSUBAME3, we use NVIDIA HPC SDK

# OpenACC Version of “mm” sample

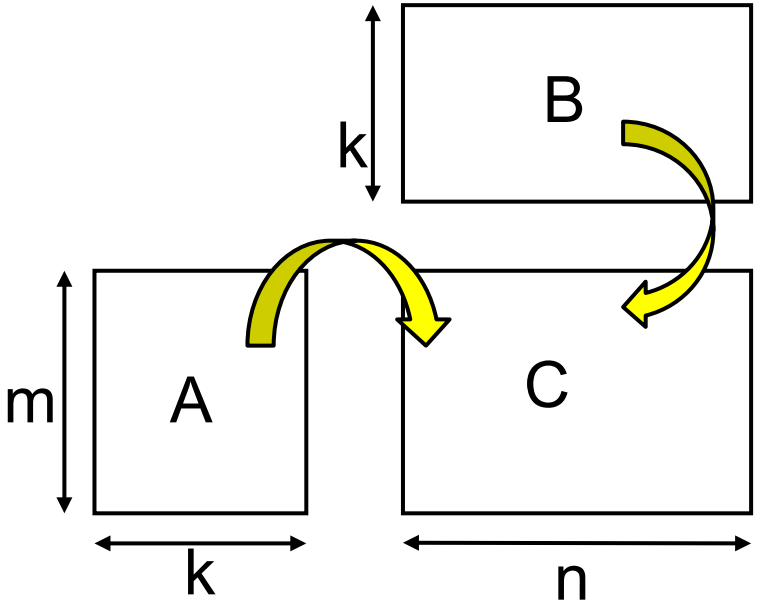
Available at /gs/hs1/tga-ppcomp/23/mm-acc/

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

C: a (m × n) matrix

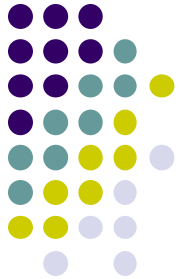
C ← A × B

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



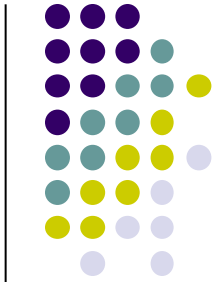

- Execution: ./mm [m] [n] [k]

# Using mm-acc Sample



*[make sure that you are at a interactive node (r7i7nX) ]*
module load gcc nvhpc *[Do once after login]*
cd ~/t3workspace *[Example in web-only route]*
cp -r /gs/hs1/tga-ppcomp/23/mm-acc .
cd mm-acc
make
*[You will see some messages, and an executable file "mm" is created]*
./mm 1000 1000 1000

# Notes on Compiling OpenACC Programs



- NVIDIA HPC SDK on TSUBAME3.0
  - module load gcc nvhpc, and then use pgcc command
  - Use -acc option in compiling and linking
  - -Minfo=accel option outputs many information on parallelization

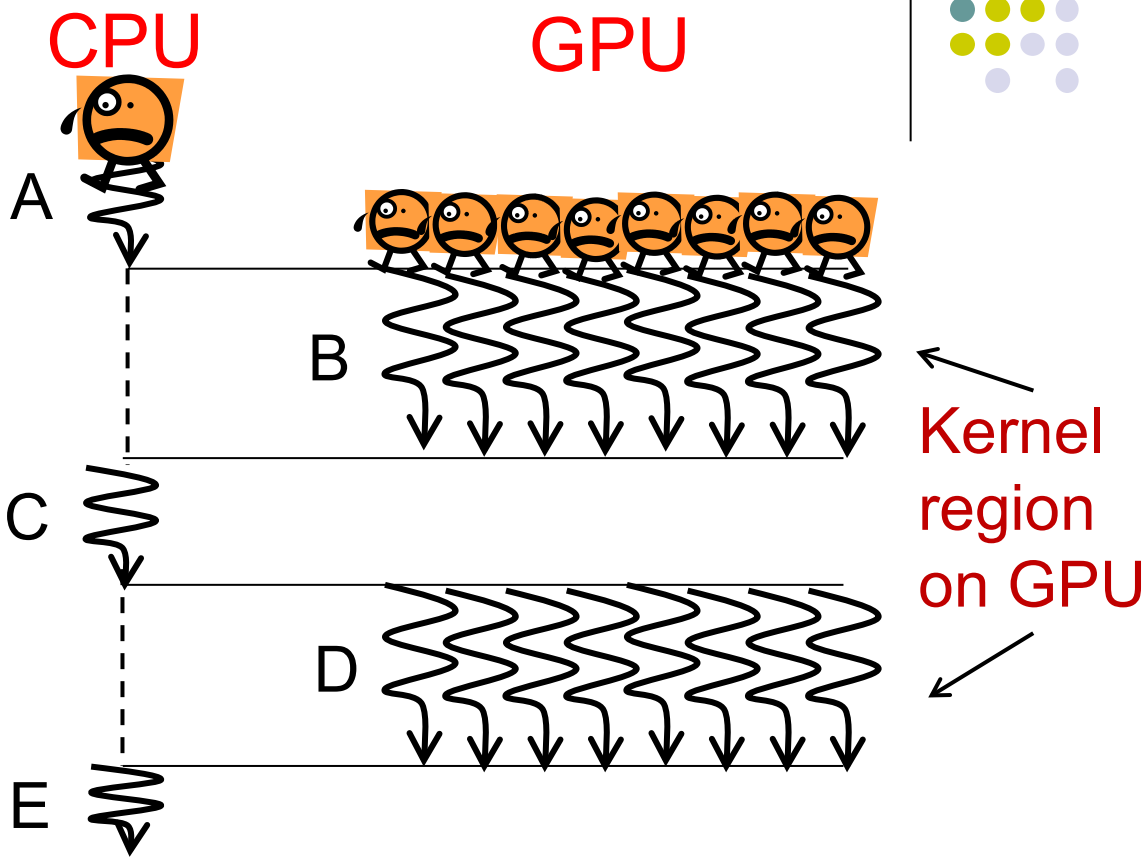
Example of output

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

They are not errors

# Kernel Region in OpenACC

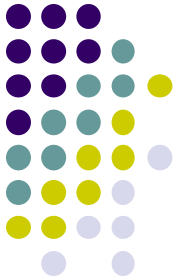
```
int main()
{
    A;
#pragma acc kernels
    {
        B;
    }
    C;
#pragma acc kernels
    D;
    E;
}
```



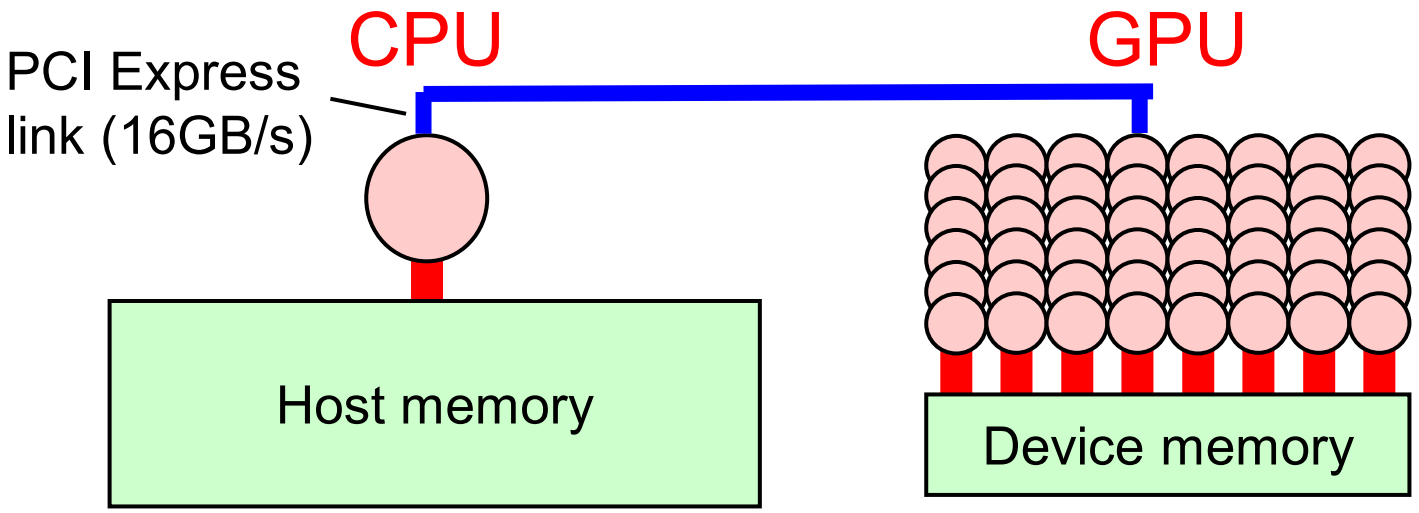

A sentence/block immediately after #pragma acc kernels is called a kernel region, executed on GPU

- We don't need to specify number of threads (it is hard to specify explicitly)

# Data Movement between CPU and GPU

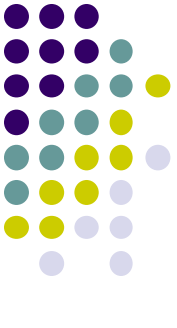


- We need to move data between CPU and GPU
  - Host (CPU) memory and Device (GPU) memory are distinct, like distributed memory
  - Threads on a GPU share the device memory

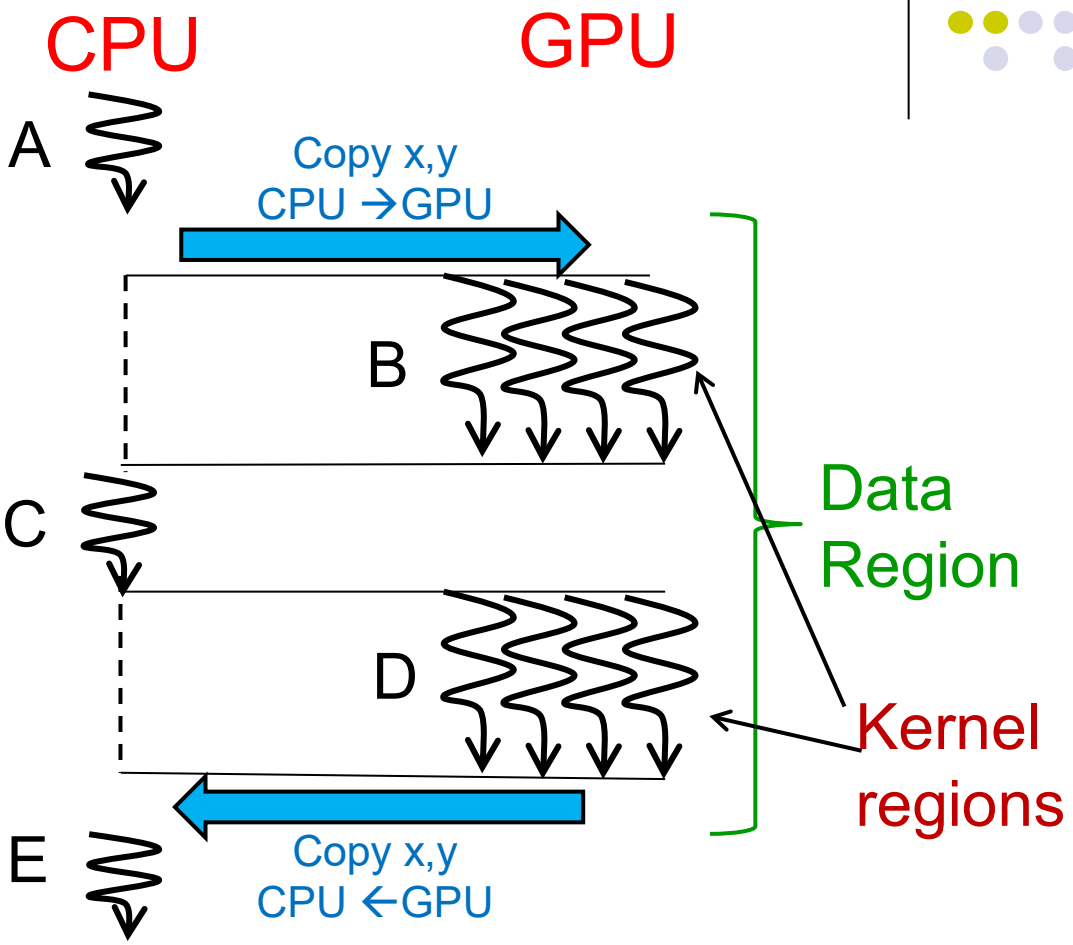



For this purpose, we use #pragma acc data directive
→ This defines a data region

# 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 (1)

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

cf) b [ 0 : 20 ]

start index  number of elements

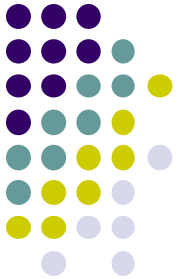
  - Partial copying like b[10:5] or a[4:4] work
- Scalar variables (like x):
  - You can omit copy(x) ➔ The compiler detects automatically ☺

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

Same meaning

```
#pragma acc data copy(a[0:10], b[0:20])
```

# Data Directive (2)



- Directions of copying
  - … data copyin(…): Copy CPU→GPU at the begininng
  - … data copyout(…): Copy GPU→CPU at the end
  - … data copy(…): Do both

  *Optimization of data movement will help speedup*

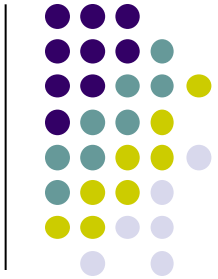
# Loop Directive

```
   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 kernels
#pragma acc loop independent
  for (j = 0; j < n; j++) {
#pragma acc loop seq
    for (l = 0; l < k; l++) {
#pragma acc loop independent
      for (i = 0; i < m; i++) {
        Ci,j += Ai,l * Bl,j;
      } } }
```

We can omit GPU➔CPU copy of A,B

←For each column in C

←For dot product

←For each row in C

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

# Different Loop Orders

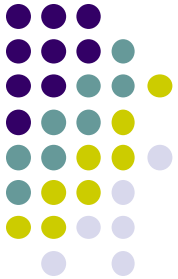
- mm-acc uses JLI nested loop
- mm-jil-acc uses JIL nested loop

→ Both have the same amount of computations. How are speeds?

There are $P_3$=6 variations of triply nested loop

- IJL, ILJ, JIL, JLI, LIJ, LJI
- Which is the fastest? And how about on CPUs?

# Submitting a GPU Job to the Job Scheduler



- 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

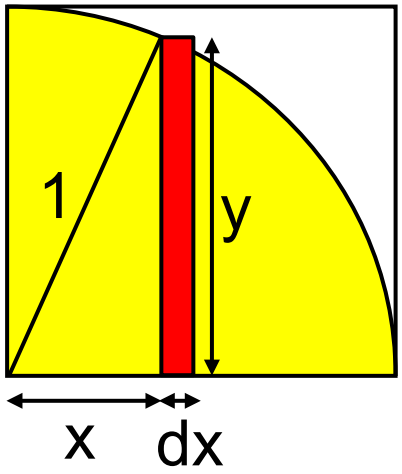
```
#!/bin/sh
#$ -cwd
#$ -l q_node=1
#$ -l h_rt=00:10:00

./mm 1000 1000 1000
```

- Job submission
  - qsub job.sh

# OpenACC version of “pi” sample

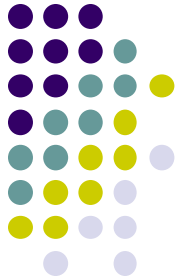
Estimate approximation of π (circumference/diameter) by approximation of integration

- Available at /gs/hs1/tga-ppcomp/23/pi-acc/
- Method
  - Let SUM be approximation of the yellow area
  - 4 x SUM → π
- Execution : ./pi [n]
  - n: Number of division
  - Cf) ./pi 100000000
- Compute complexity : O(n)




dx = 1/n
y = sqrt(1-x*x)

# Algorithm of “pi”



OpenMP

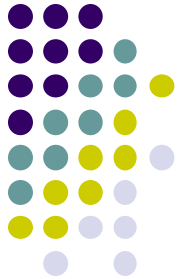
```
double pi(int n) {
    int i;
    double sum = 0.0;
    double dx = 1.0 / (double)n;

#pragma omp parallel
#pragma omp for reduction(+:sum)
    for (i = 0; i < n; i++) {
        double x = (double)i * dx;
        double y = sqrt(1.0 - x*x);
        sum += dx*y;
    }

    return 4.0*sum; }
```

OpenACC

```
double pi(int n) {
    int i;
    double sum = 0.0;
    double dx = 1.0 / (double)n;

#pragma acc kernels
#pragma acc loop independent reduction(+:sum)
    for (i = 0; i < n; i++) {
        double x = (double)i * dx;
        double y = sqrt(1.0 - x*x);
        sum += dx*y;
    }

    return 4.0*sum; }
```

※ For scalar variables, “data copy” is omitted

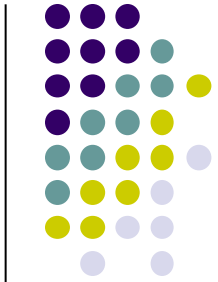
# Notes on Number of Threads



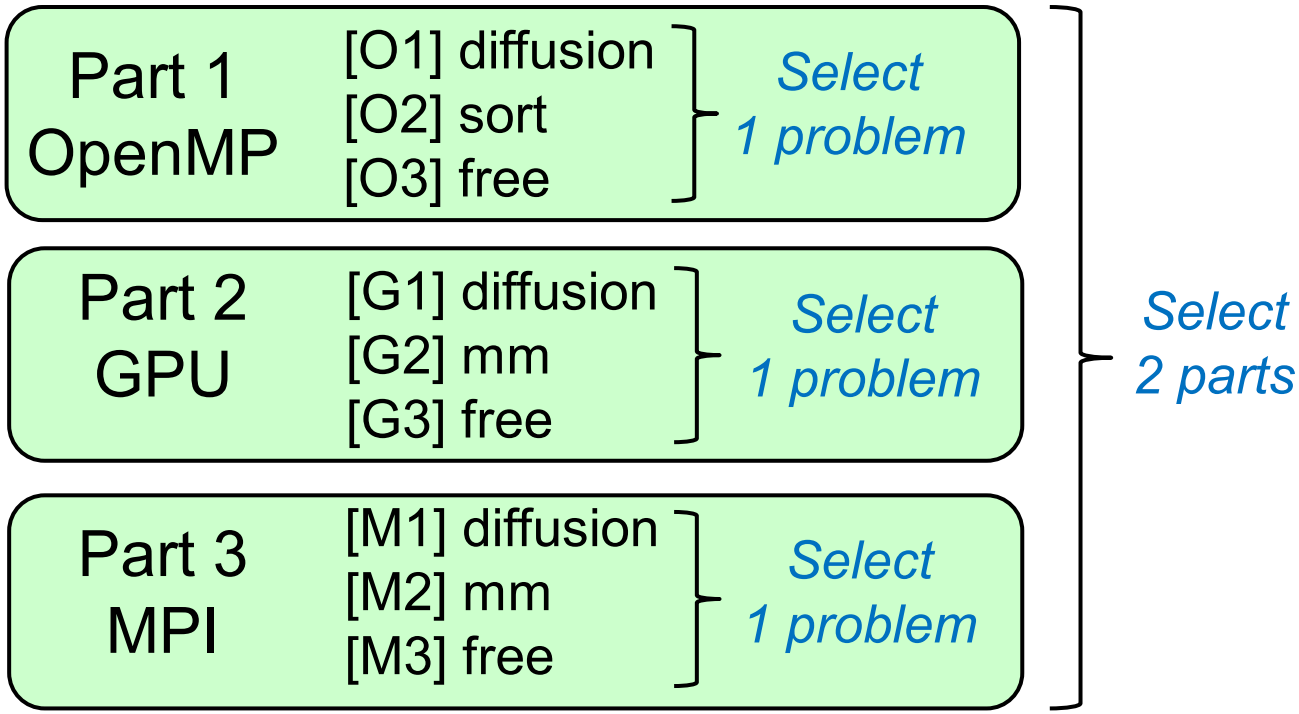
- In OpenMP, the number of threads is set by OMP_NUM_THREADS
- In OpenACC, the number is automatically determined per loop

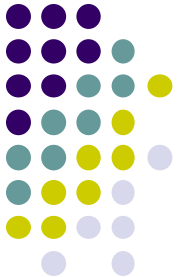
- In OpenMP, thread ID is obtained by omp_get_thread_num()
- In OpenACC, we cannot see thread ID

# Assignments in this Course



- There is homework for each part. Submissions of reports for 2 parts are required





# Assignments in GPU Part (1)

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

Due date: May 25 (Thursday)

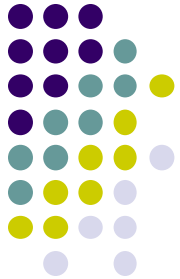
[G1] Parallelize “diffusion” sample program by OpenACC or CUDA

- You can use Makefile in /gs/hs1/tga-ppcomp/23/diffusion-acc/ or /gs/hs1/tga-ppcomp/23/diffusion-cuda/

Optional:

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

# Assignments in GPU Part(2)



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

- Use various matrices sizes
- Evaluate effects of data transfer cost
- Compare with CPU (OpenMP) version

Optional：

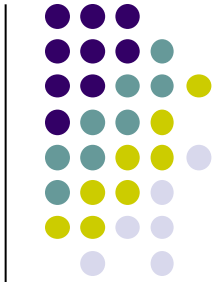
- To use different loop orders
- To evaluate both mm-acc and mm-cuda
- To change/improve the program
  - Different assignment of threads and elements (CUDA) etc

# Assignments in GPU Part (3)

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

- cf) A problem related to your research
- "sort" sample on GPU?
  - The quick sort may be hard on GPU (There is no "task" syntax)
  - → Bitonic sort?
- More challenging one for parallelization is better
  - cf) Partial computations have dependency with each other

# Notes in Report Submission (1)

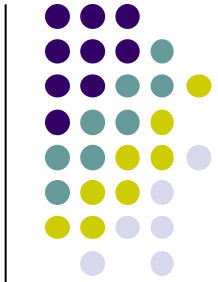


- Submit the followings via T2SCHOLA

  (1) A report document
  - PDF, MS-Word or text file
  - 2 pages or more
  - in English or Japanese (日本語もok)

  (2) Source code files of your program
  - Try “zip” to submit multiple files

# Notes in Report Submission (2)



The report document 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 threads
  - With varying problem sizes
  - Discussion with your findings
  - Other machines than TSUBAME are ok, if available

# Next Class:

- On Monday, May 8
- GPU Programming (2)
  - OpenACC
    - Improving data copy
    - Improving loop parallelization
  - Introduction of CUDA