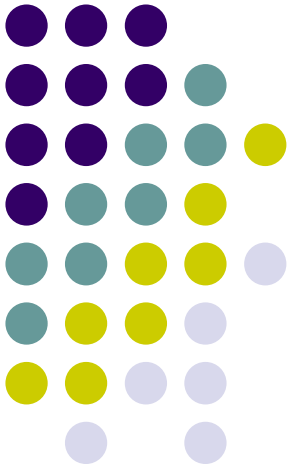
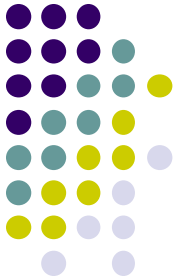
Practical Parallel Computing (実践的並列コンピューティング)

Part2: GPU (2)
June 1, 2020



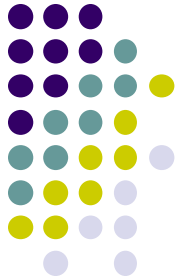
Toshio Endo
School of Computing & GSIC
endo@is.titech.ac.jp

Overview of This Course

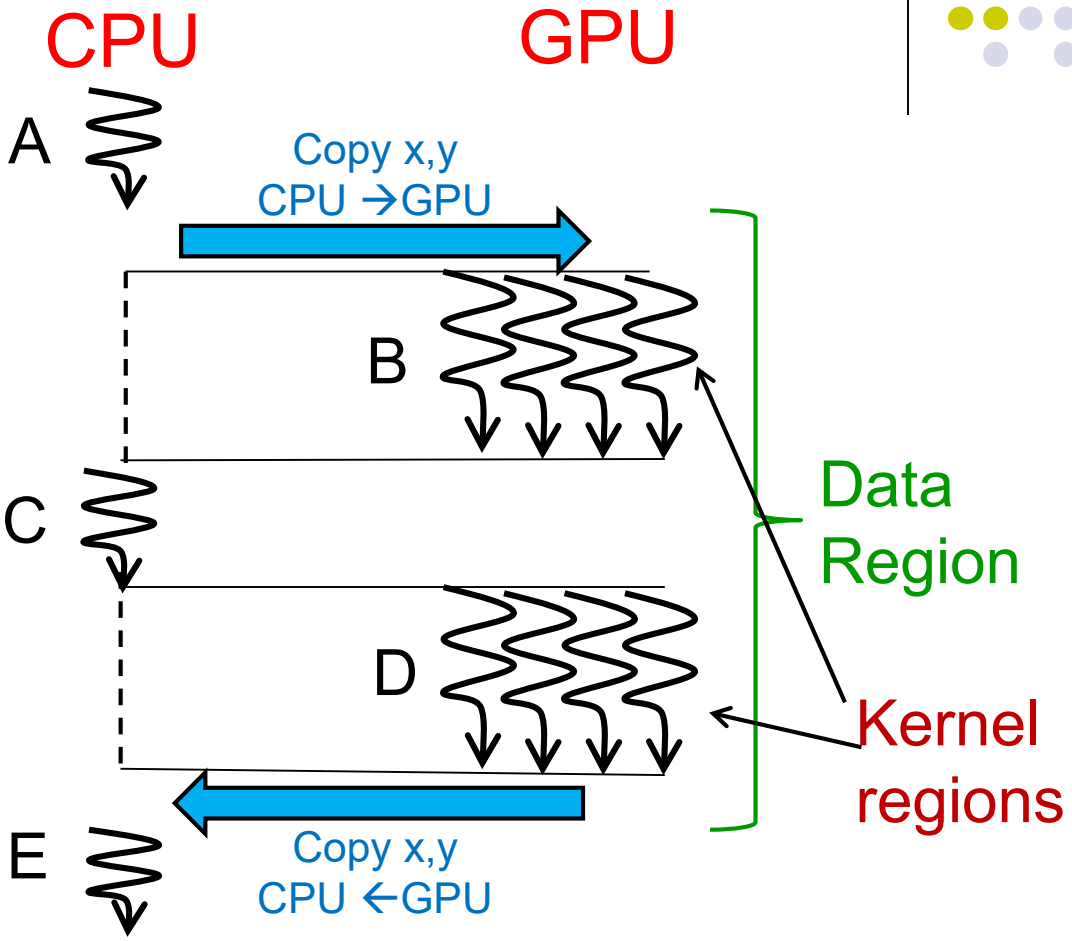


- Part 0: Introduction
  - 2 classes
- Part 1: OpenMP for shared memory programming
  - 4 classes
- Part 2: GPU programming
  - 4 classes ← We are here (2/4)
  - OpenACC (1.5 classes) and CUDA (2.5 classes)
- Part 3: MPI for distributed memory programming
  - 3 classes

Data Region and Kernel Region in OpenACC



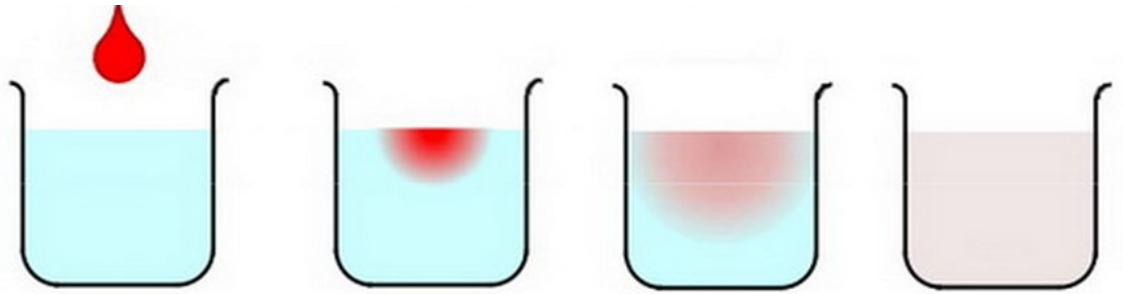
```
int main()
{
    A;
#pragma acc data copy(x,y)
    {
#pragma acc kernels
        {
            B;
        }
        C;
#pragma acc kernels
        D;
    }
    E;
}
```



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

“diffusion” Sample Program
related to [G1]

An example of diffusion phenomena:

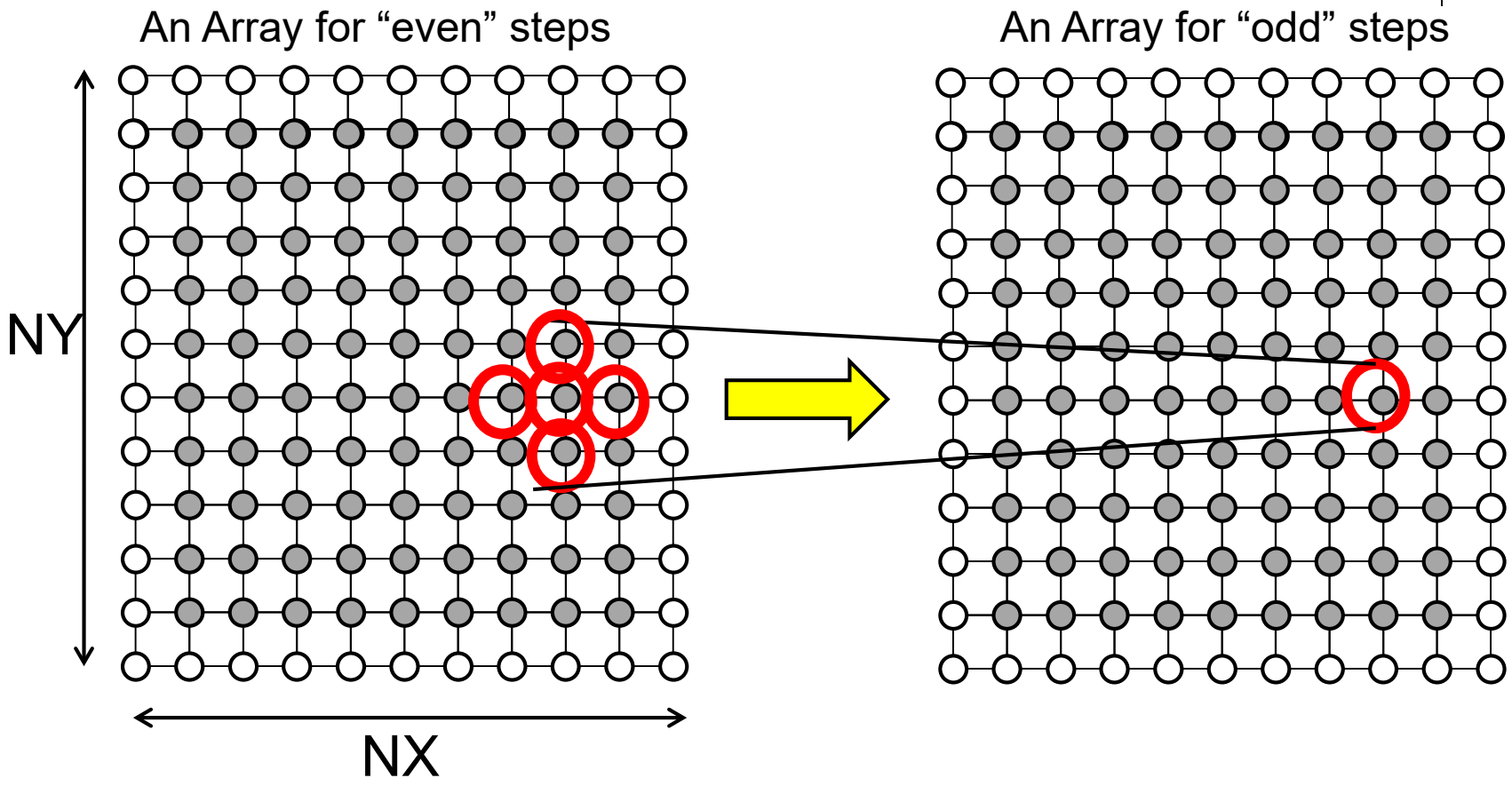


The ink spreads gradually, and finally the density becomes uniform (Figure by Prof. T. Aoki)

Available at /gs/hs1/tga-ppcomp/20/diffusion/

- Execution：./diffusion [nt]
  - nt: Number of time steps

Data Structure in “diffusion”



Consideration of Parallelizing Diffusion with OpenACC
related to [G1]

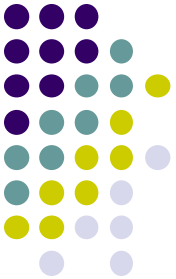
- x, y loops can be parallelized
  - We can use “#pragma acc loop” twice
- t loop cannot be parallelized

```
[Data transfer from CPU to GPU]
for (t = 0; t < nt; t++) {

  for (y = 1; y < NY-1; y++) {
    for (x = 1; x < NX-1; x++) {
          :
    }
  }
}
[Data transfer from GPU to CPU]
```

Kernel region on GPU
Parallel x, y loops

It’s better to transfer data *out of* t-loop

data Clause for Multi-Dimensional arrays



float A[2000][1000]; → an example of a 2-dimension array

…. data copy(A)

→ OK, all elements of A are copied

…. data copy(A[0:2000][0:1000])

→ OK, all elements of A are copied

…. data copy(A[500:600][0:1000])

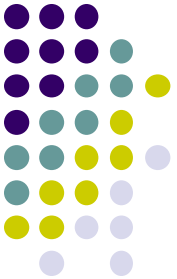
→ OK, rows[500,1100) are copied

…. data copy(A[0:2000][300:400])

→ NG in current OpenACC

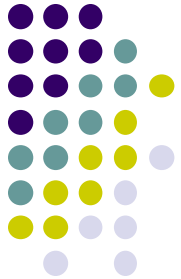
※ Currently, OpenACC does not support non-consecutive transfer

Notes on Assignment [G1]



- You will need compiler options different from the diffusion directory for OpenACC
- You can use files in diffusion-acc directory as basis
  - “Makefile” in this directory supports compiler options for OpenACC
  - Don’t forget “module load cuda pgi” before “make”

Data Update inside Data Region



- Data on GPU can be updated with “acc update” inside data region
  - Also “acc update” can work with “acc enter data” , “acc exit data” (appear later)

- “acc update” is still different from “acc data”
  - “acc data” may create/delete copy on GPU
  - “acc update” does not; it assumes the copy already presents

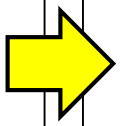
```
[C/C++]
  #pragma acc data copy(x[0:N])
  {
    #pragma acc parallel loop
    for (i=0; i<N; i++) {
      x[i] += ...;  /* GPU */
    }
    #pragma acc update self(x[0:N])

    for (i=0; i<N; i++) {
      x[i] += ...;  /* CPU */
    }
    #pragma acc update device(x[0:N])

    #pragma acc parallel loop
    for (i=0; i<N; i++) {
      x[i] += ...;  /* GPU */
    }
  }
```

※ acc parallel works like acc kernels

By Akira Naruse, NVIDIA

mm-acc/mm.c is Updated
related to [G2]

```
#pragma acc loop independent
  for (j = 0; j < n; j++) {
#pragma acc loop seq
    for (l = 0; l < k; l++) {
      double blj = B[l+j*ldb];
#pragma acc loop independent
      for (i = 0; i < m; i++) {
        double ail = A[i+l*lda];
        C[i+j*ldc] += ail*blj;
      }}}
```

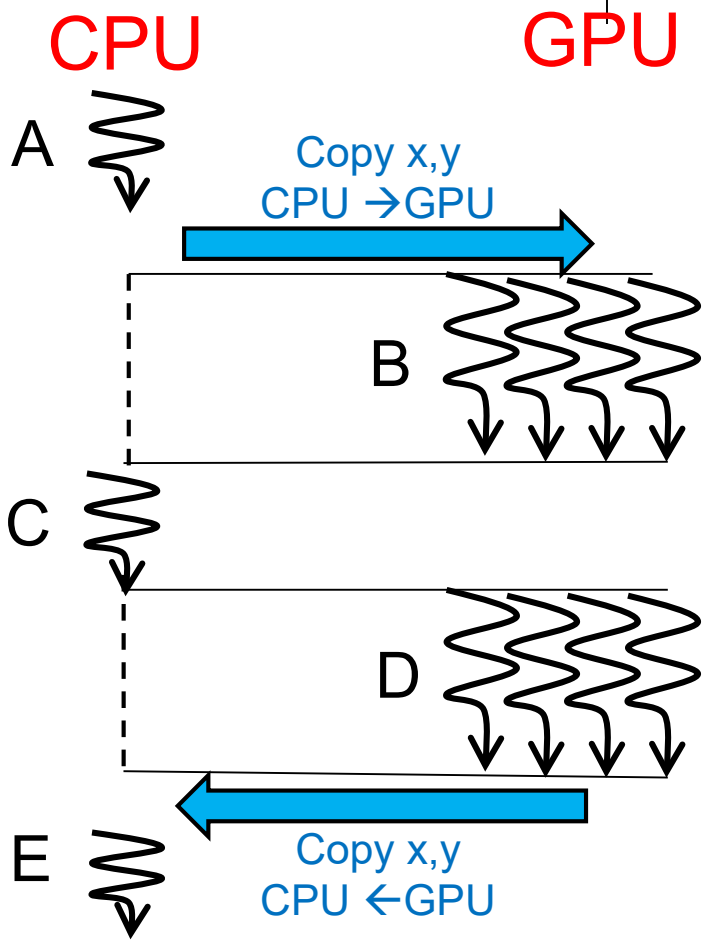


```
#pragma acc loop independent
  for (j = 0; j < n; j++) {
#pragma acc loop seq
    for (l = 0; l < k; l++) {
#pragma acc loop independent
      for (i = 0; i < m; i++) {
        double ail = A[i+l*lda];
        double blj = B[l+j*ldb];
        C[i+j*ldc] += ail*blj;
      }}}
```

- The new version is around 3 times faster, please use this version in [G2]
  - and faster than mm-jil-acc
  - Currently I cannot explain the reason ☹

Data Transfer Costs in GPU Programming

- In GPU programming, data transfer costs between CPU and GPU have impacts on speed
  - Program speed may be slower than expected ☹



Let's discuss impacts of transfer in mm-acc

Speed of GPU Programs:
case of mm-acc

In mm-acc, speed in Gflops is computed by

$$S = 2mnk / T_{total}$$

T_{total} includes both computation time and transfer

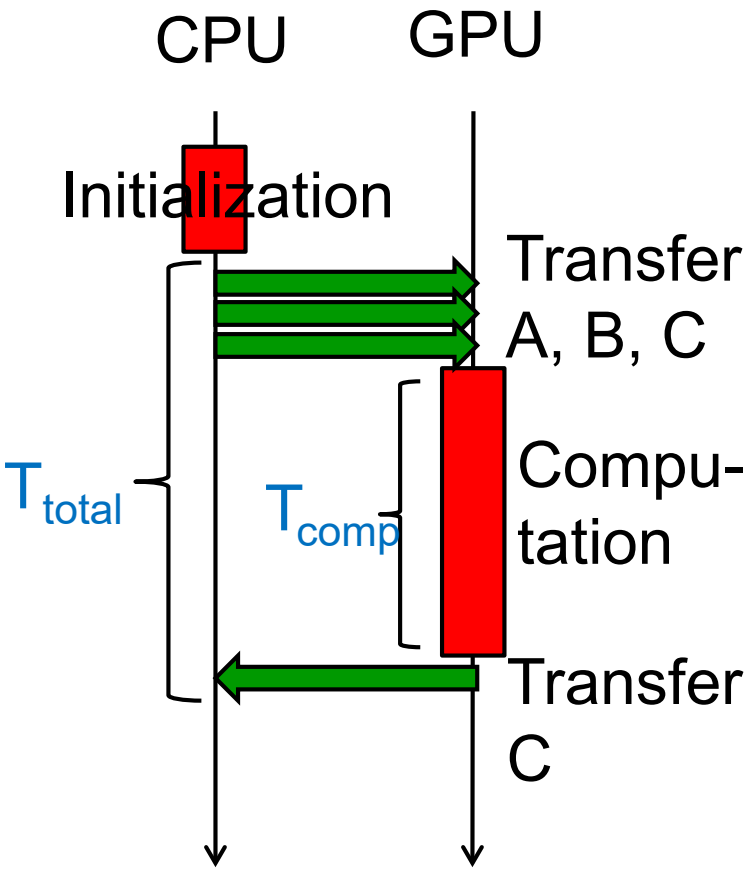
→ S counts slow-down by transfer

To see the effects, let's try another sample

/gs/hs1/tga-ppcomp/20/mm-meas-acc

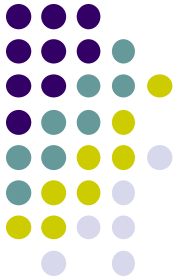
which outputs time for

- copyin (transfer A, B, C)
- computation
- copyout (transfer C)



In [G2], please evaluate effects of transfer costs

Another Description Way for Data Copy

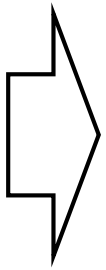


How can we measure transfer time?

- With “data” directive, copy timing is restricted

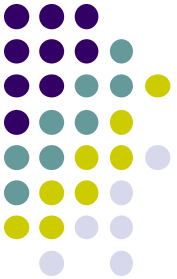
→ We can copy data anytime by “acc enter data”, ”acc exit data” directives

```
// x,y are on CPU

#pragma acc data copy(x,y)
   {
      // x,y are on GPU
   }

// x,y are on CPU
```



```
// x,y are on CPU

#pragma acc enter data copyin(x,y)

   // x,y are on GPU

#pragma acc exit data copyout(x,y)

// x,y are on CPU
```

Discussion on Data Transfer Costs



- Time for data transfer $T_{trans} \fallingdotseq M / B + L$
  - M: Data size in bytes
  - B: “Bandwidth” (speed)
  - L: “Latency” (if M is sufficiently large, we can ignore it)
- In a P100 GPU,
  - Theoretical computation speed is 5.3TFlops
  - Theoretical bandwidth B is 16GB/s (2G double values per second)

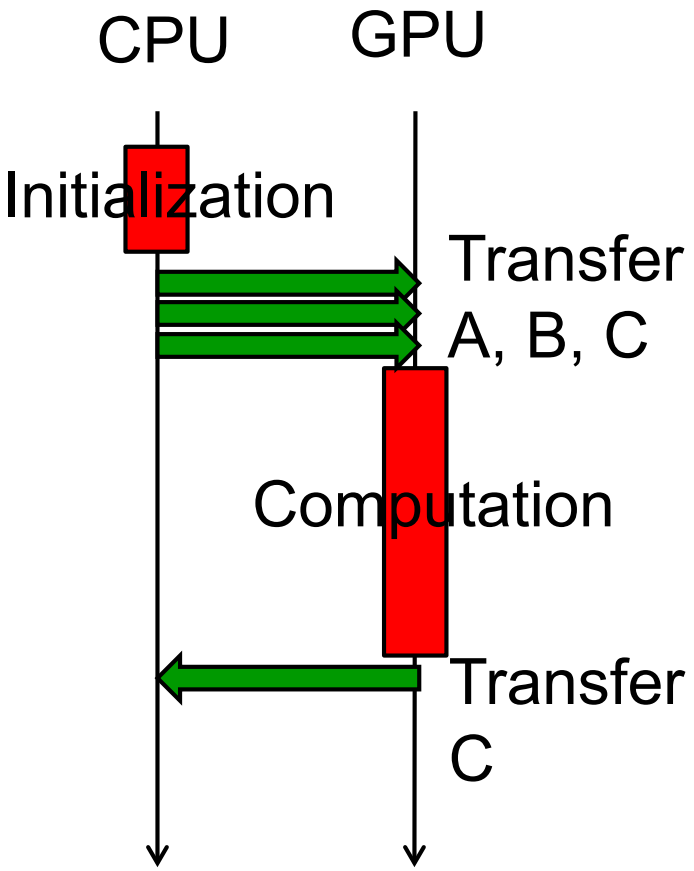
  → Transfer of values is much slower than computation ☹

Discussion on Computation and Transfer Costs

In mm-acc,

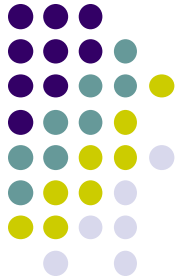
- Computation amount: O(mnk)
- Data transfer amount:
  - A, B, C: CPU → GPU: O(mk+kn+mn)
  - C: GPU → CPU: O(mn)

Transfer costs are relatively smaller with larger m, n, k



In diffusion-acc [G1], how can we reduce transfer costs?

Function Calls from GPU



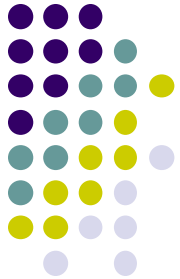
- Kernel region can call functions, but be careful

```
int main()
{

#pragma acc kernels
   {

        … func(A[i]) …

   }

}
```

```
#pragma acc routine
int func(int arg)
{
    :
    :
    return …;
}
```

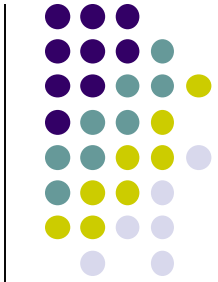
- “routine” directive is required by compiler to generate GPU code

How about Library Functions?



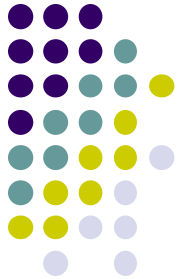
- Available library functions is very limited ☹
- We cannot use strlen(), memcpy(), fopen()… ☹

- Exceptionally, some mathematical functions are ok ☺
  - fabs, sqrt, fmax…
  - #include <math.h> is needed
- Very recently, printf() in kernel regions is ok! ☺



Now explanation of OpenACC is finished; we will go to CUDA

OpenACC and CUDA for GPUs

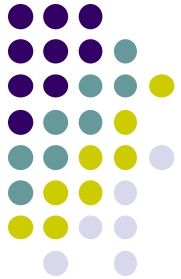


- OpenACC
  - C/Fortran + directives (#pragma acc …), Easier programming
  - PGI compiler works
    - module load pgi
    - pgcc –acc … XXX.c
  - Basically for data parallel programs with for-loops

  → Only for limited types of algorithms ☹
- CUDA
  - Most popular and suitable for higher performance
  - Use “nvcc” command for compile
    - module load cuda
    - nvcc … XXX.cu

  Programming is harder, but more general

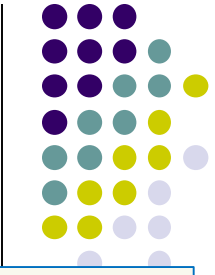
An OpenACC Program Look Like



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

   // After kernel region finishes,
   CPU can access to A[i],B[i]
```

Executed on GPU in parallel

A CUDA Program Look Like



Sample:
/gs/hs1/tga-ppcomp/20/add-cuda/

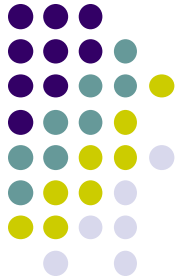
```
int A[100], B[100];
int *DA, *DB;
int i;
cudaMalloc(&DA, sizeof(int)*100);
cudaMalloc(&DB, sizeof(int)*100);
cudaMemcpy(DA,A,sizeof(int)*100,
    cudaMemcpyHostToDevice);
cudaMemcpy(DB,B,sizeof(int)*100,
    cudaMemcpyHostToDevice);

add<<<20, 5>>>(DA, DB);

cudaMemcpy(A,DA,sizeof(int)*100,
    cudaMemcpyDeviceToHost);
```

```
__global__ void add
    (int *DA, int *DB)
{
    int i = blockIdx.x*blockDim.x
          + threadIdx.x;
    DA[i] += DB[i];
}
```

Executed on GPU
(called a *kernel function*)

We have to separate code regions executed on CPU and GPU

Using add-cuda Sample

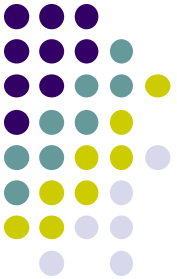


```
[make sure that you are at a interactive node (r7i7nX) ]
module load cuda        [Do once after login]
cd ~/t3workspace        [Example in web-only route]
cp -r /gs/hs1/tga-ppcomp/20/add-cuda  .
cd add-cuda
make
[An executable file “add” is created]
./add
```

※ [Standard route] A log-in node does not have a GPU

→ You can compile the sample there, but the program does not work!

Compiling CUDA Programs/ Submitting GPU Jobs



- Compile .cu file using the NVIDIA CUDA toolkit
  - module load cuda
  - and then use nvcc

Also see Makefile in the sample directory

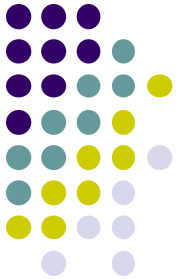
- Job submission method is same as OpenACC version

add-cuda/job.sh

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

./add
```

⇨ qsub job.sh

Preparing Data on Device Memory

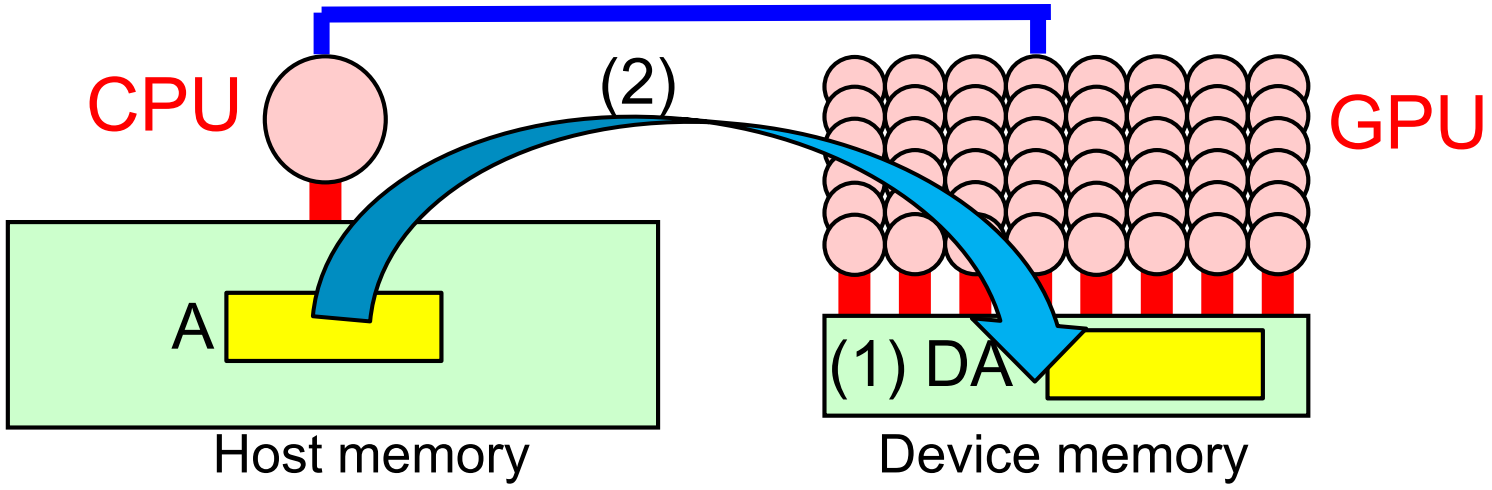


(1) Allocate a region on device memory

cf) cudaMalloc((void**)&DA, *size*);

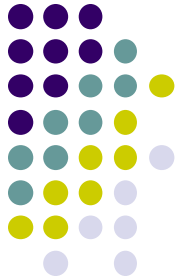
(2) Copy data from host to device

cf) cudaMemcpy(DA, A, *size*, cudaMemcpyDefault);



Note: cudaMalloc and cudaMemcpy must be called on CPU, NOT on GPU

Comparing OpenACC and CUDA



OpenACC

Both allocation and copy are done by acc data copyin

One variable name A may represent both

- A on host memory
- A on device memory

```
  int A[100];  ← on CPU
#pragma acc data copy(A)
#pragma acc kernels
 {
   … A[i] …
 }        ↖ on GPU
```

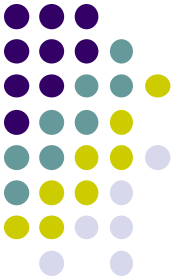
CUDA

cudaMalloc and cudaMemcpy are separated

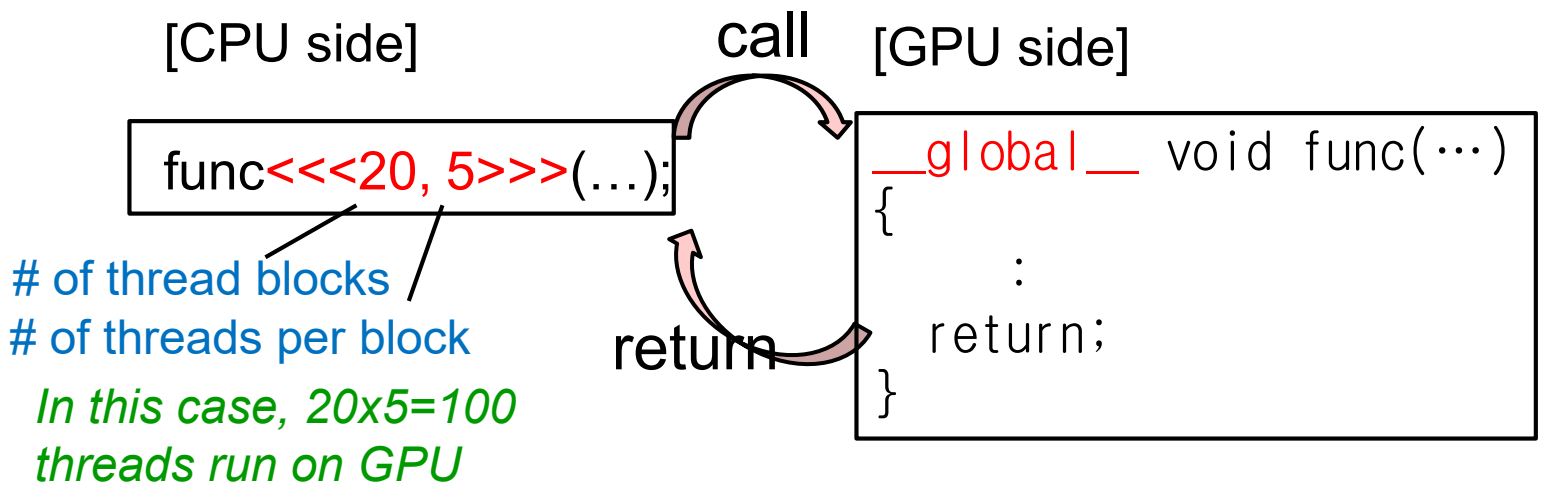
Programmer have to prepare two pointers, such as A and DA

```
int A[100];
int *DA;
cudaMalloc(&DA, …);
cudaMemcpy(DA, A, …, …);
// Here CPU cannot access DA[i]

func<<<…, …>>>(DA, …);
```

Calling A GPU Kernel Function from CPU



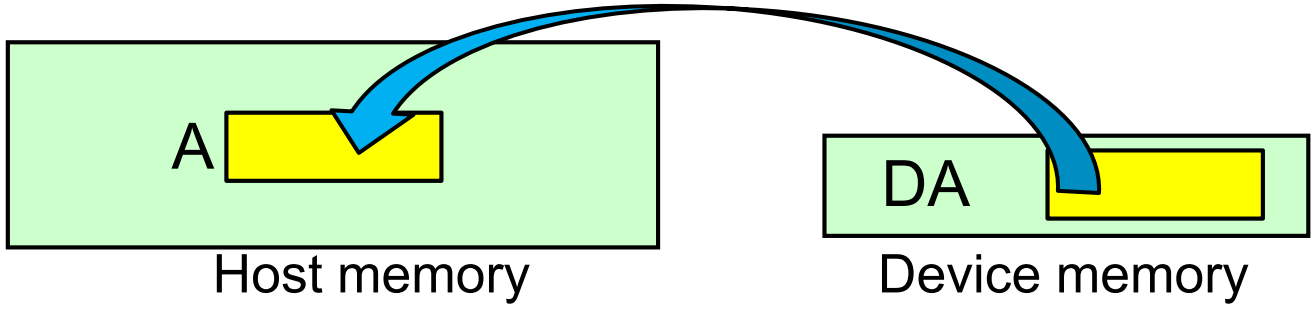
- A region executed by GPU must be a distinct function
  - called a GPU kernel function



A GPU kernel function (called from CPU)

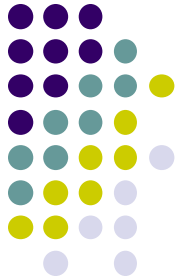
- needs __global__ keyword
- can take parameters
- can NOT return value; return type must be void

Copying Back Data from GPU



- Copy data using cudaMemcpy
  - cf) cudaMemcpy(A, DA, *size*, cudaMemcpyDefault);
  - 4th argument is one of
    - cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice, cudaMemcpyHostToHost
    - cudaMemcpyDefault ← Detect memory type automatically ☺
- When a memory area is unnecessary, free it
  - cf) cudaFree(DA);

Assignments in GPU Part (Abstract)



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

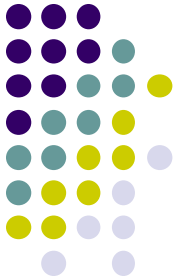
Due date: June 18 (Thursday)

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

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

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

Notes in Report Submission (1)

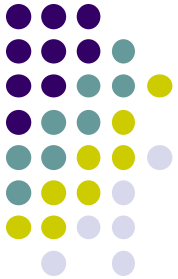


- Submit the followings via OCW-i

  (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

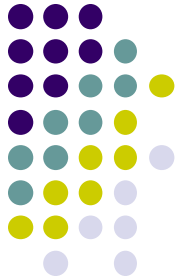
Notes in Report Submission (2)



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

FYI:
Event Announcement



GPU online minicamp

- June 15 (Mon) – 16 (Tue)
- Online (Slack & Zoom)
- Professional mentors, including NVIDIA technical staffs, will help to solve issues on GPU programming

http://gpu-computing.gsic.titech.ac.jp/node/102

- Application deadline: June 8

Next Class:

- GPU Programming (3)
  - Multi-threads on CUDA