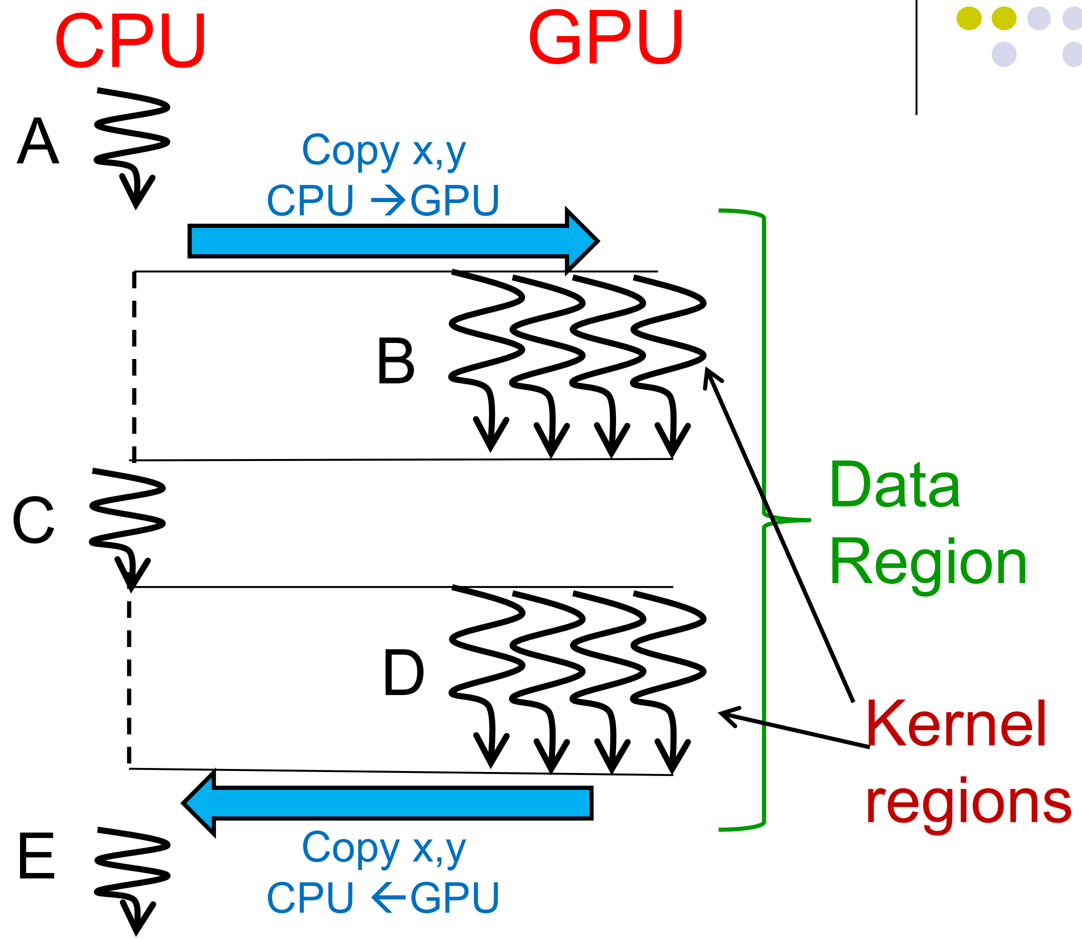
2019 Practical Parallel Computing (実践的並列コンピューティング) No. 12

GPU Programming (2)

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

Data Region and Kernel Region in OpenACC

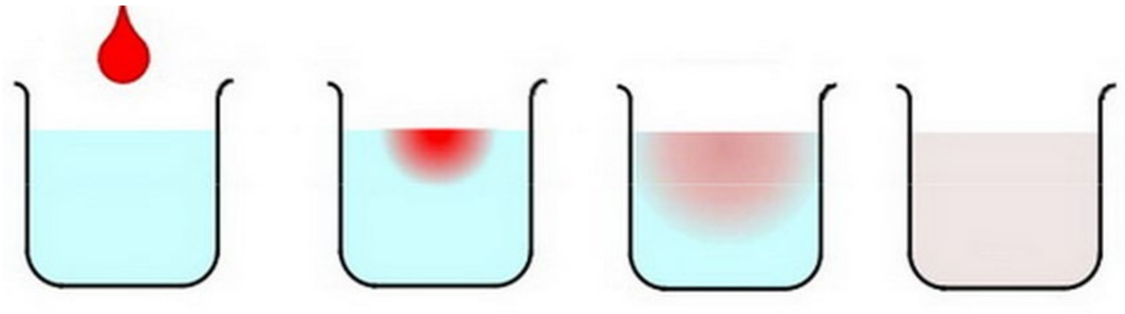
```
int main()
{
    A;
#pragma acc data copy(x,y)
    {
#pragma acc kernels
        {
            B;
        }
        C;
#pragma acc kernels
        D;
    }
    E;
}
```



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

"diffusion" Sample Program
related to [G1]

An example of diffusion phenomena:

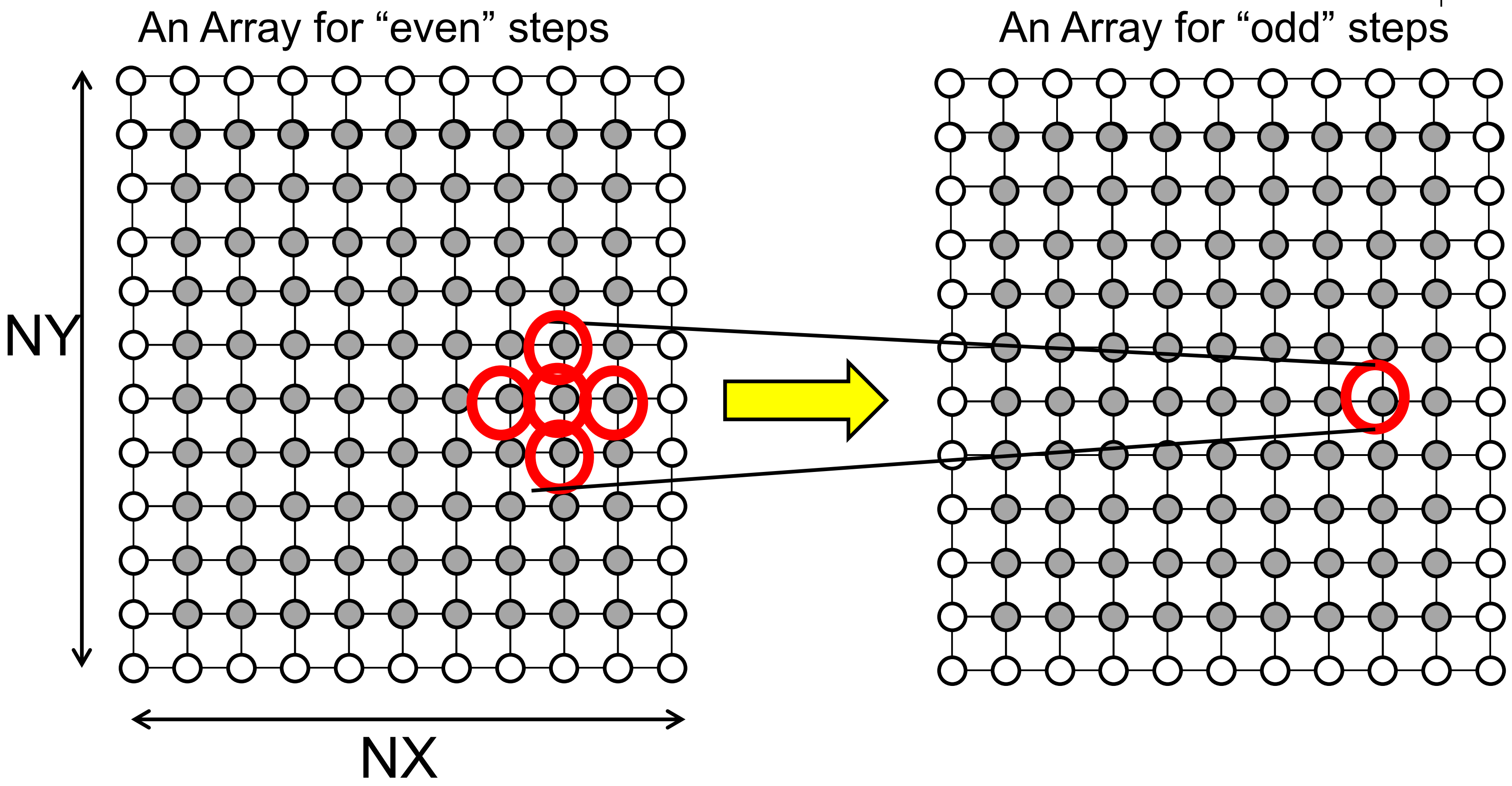


The ink spreads gradually, and finally the density becomes uniform   (Figure by Prof. T. Aoki)

Available at ~endo-t-ac/ppcomp/19/diffusion/

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

Data Structure in “diffusion”



Parallelizing Diffusion with OpenACC

- x, y loops are 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

To See Messages from Compiler

- We often want to see “what compiler did”
  - Is the loop really parallelized?

```
% pgcc -O2 -acc -Minfo mm.c
     :
  24, Generating copyin(A[:m*k])
      Generating copy(C[:m*n])
       Generating copyin(B[:k*n])
     :
  29, Loop is parallelizable
  31, Loop is parallelizable
      Generating Tesla code
      27, #pragma acc loop seq
      29, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
      31, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

Line numbers in source code

Another Description Way for Data Copy

- With “data” directive, copy timing is restricted

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

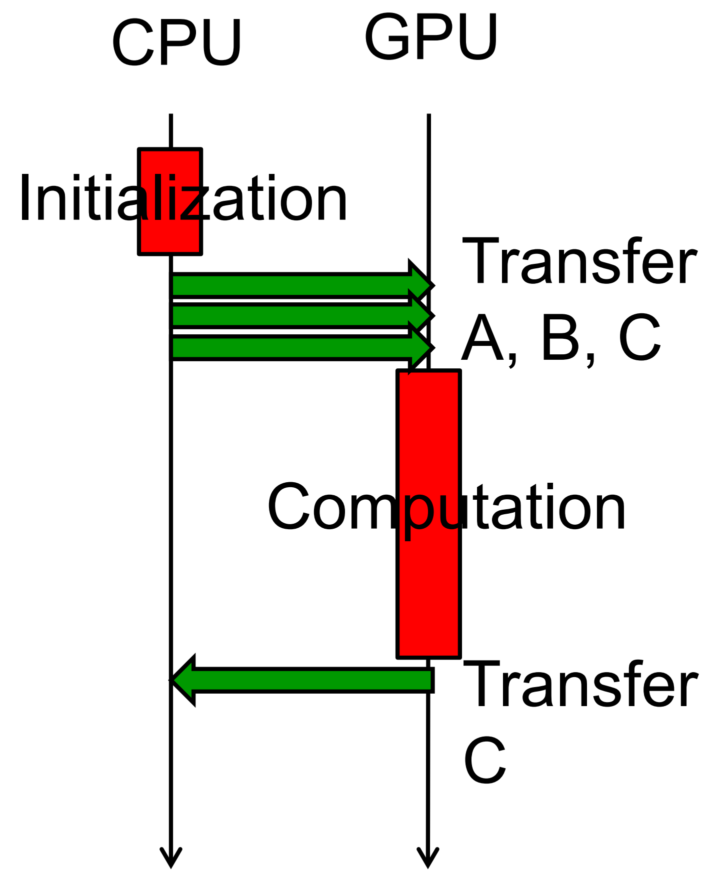
Data Transfer in mm-acc sample related to [G2]

- Data transfer between CPU and GPU is not free ☹
  - T = M / B + L

    Data size — Bandwidth between CPU/GPU (~16GB/s)
- Data transfer in mm-acc
  - A, B, C: CPU → GPU

    Amount of data transfer: O(mk+kn+mn)
  - Computation: O(mnk)
  - C: GPU → CPU

    Amount of data transfer: O(mn)



~endo-t-ac/ppcomp/19/mm-meas-acc sample outputs
time for copyin, computation, copyout

data Clause for Multi-Dimensional arrays

`float` A[2000][1000]; → 2-dim array

.... data copyin(A[0:2000][0:1000])

→ OK, all elements of A are copied

.... data copyin(A[<u>500:600</u>][0:1000])

→ OK, rows[500,1100) are copied

.... data copyin(A[0:2000][<u>300:400</u>])

→ NG in current OpenACC

※ Currently, OpenACC does not support non-consecutive transfer

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! ☺

Reduction in loop Directive

- “OpenMP-like” reduction is ok

```
#pragma acc data …
#pragma acc kernels …

#pragma acc loop independent reduction(+:sum)
for (i = 0; i < n; i++) {
   A[i] = … + B[i] + …;
   …
   sum += … ;
}
```

operator

Variable name

※ “operator” is one of +, *, max, min, &, |

Now explanation of OpenACC is finished; we 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

  → Less freedom in 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];
   }

   // CPU can access to A[i],B[i]
```

Executed on GPU in parallel

A CUDA Program Look Like

Sample:
~endo-t-ac/ppcomp/19/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

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

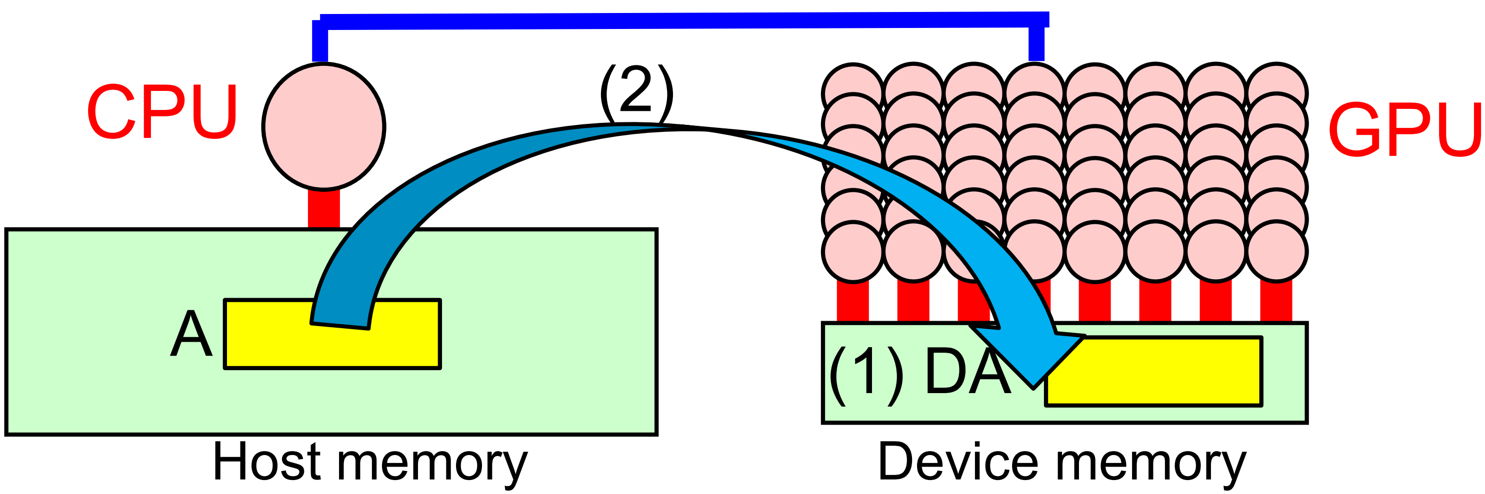
Preparing Data on Device Memory

(1) Allocate a region on device memory

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

(2) Copy data from host to device

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



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

Comparing OpenACC and CUDA

OpenACC

Both allocation and copy are done by … 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

[CPU side]

```
func<<<20, 5>>>(…);
```

# of thread blocks
# of threads per block

In this case, 20x5=100 threads run on GPU

call

return

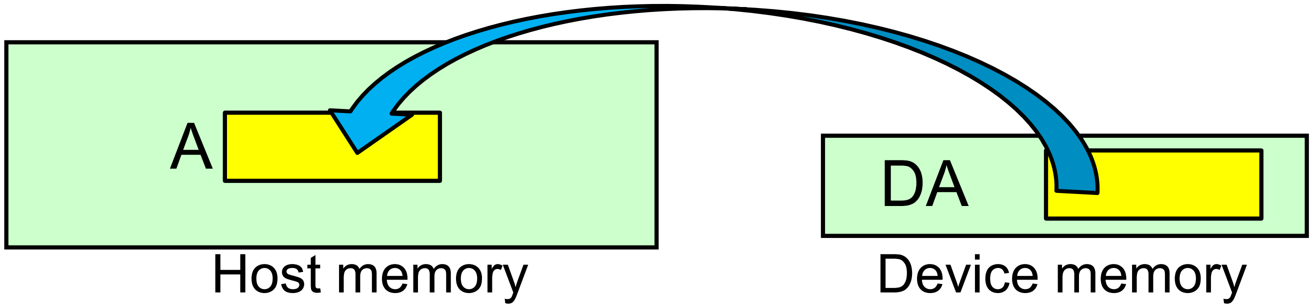
[GPU side]

```
__global__ void func(…)
{
      :
  return;
}
```

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*, cudaMemcpyDeviceToHost);
  - 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 <u>one of </u>[G1]—[G3], and submit a report

Due date: June 17 (Monday)

[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 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:

- May 27: Cancelled (休講)
- May 30: GPU Programming (3)
    - Parallelization with CUDA
- June 3: TSUBAME3.0 tour
- June 6: GPU Programming (4)

Information

Lecture

- Slides are uploaded in OCW
  - www.ocw.titech.ac.jp → search “2019 practical parallel computing”
- Assignments information/submission site are in OCW-i
  - Login portal.titech.ac.jp → OCW/OCW-i
- Inquiry
  - ppcomp@el.gsic.titech.ac.jp
- Sample programs
  - Login TSUBAME, and see ~endo-t-ac/ppcomp/19/ directory

TSUBAME

- Official web including Users guide
  - www.t3.gsic.titech.ac.jp
- Your account information
  - Login portal.titech.ac.jp → TSUBAME portal