2018
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

CUDA and OpenACC for GPUs

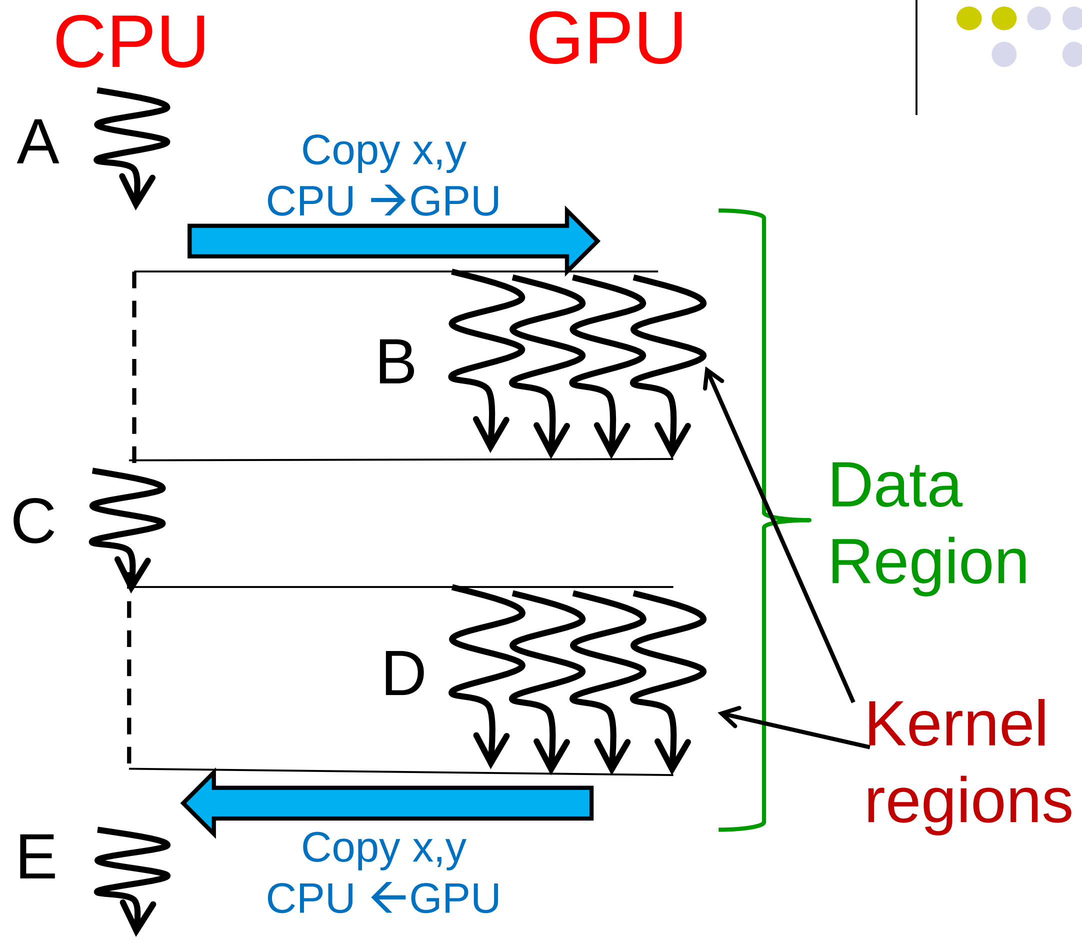
- OpenACC
  - C/Fortran + directives (#pragma acc …), Easier programming
  - PGI compiler works
    - module load pgi
    - pgcc –acc … XXX.c
  - Basically for data parallel programs with for-loops

  → 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

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;
}
```

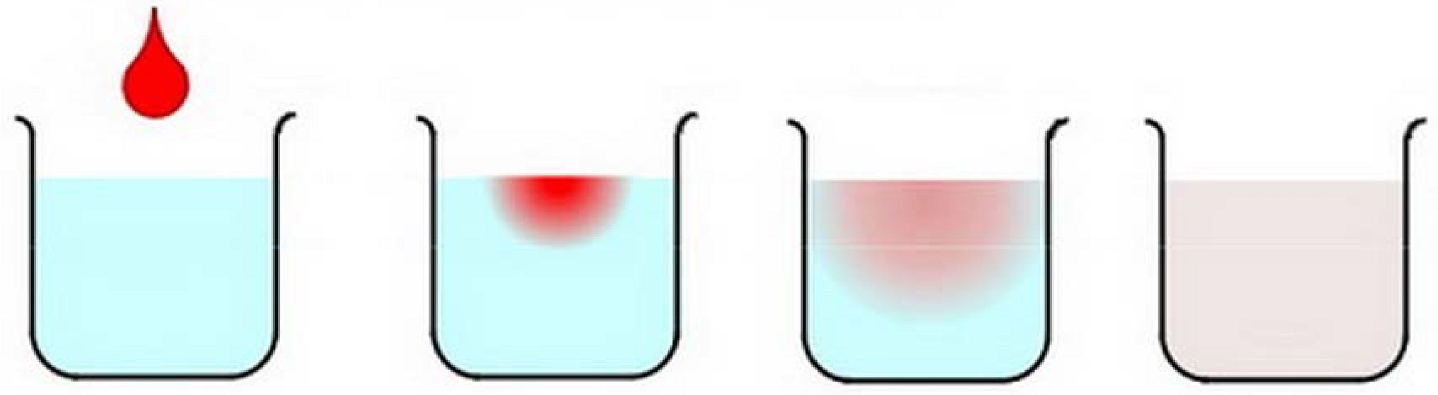


- Data region may contain 1 or more kernel regions
- Data movement occurs at beginning and end of data region

“diffusion” Sample Program (1)
(Revisited, related to [G1])

An example of diffusion phenomena:

- Pour a drop of ink into a water glass

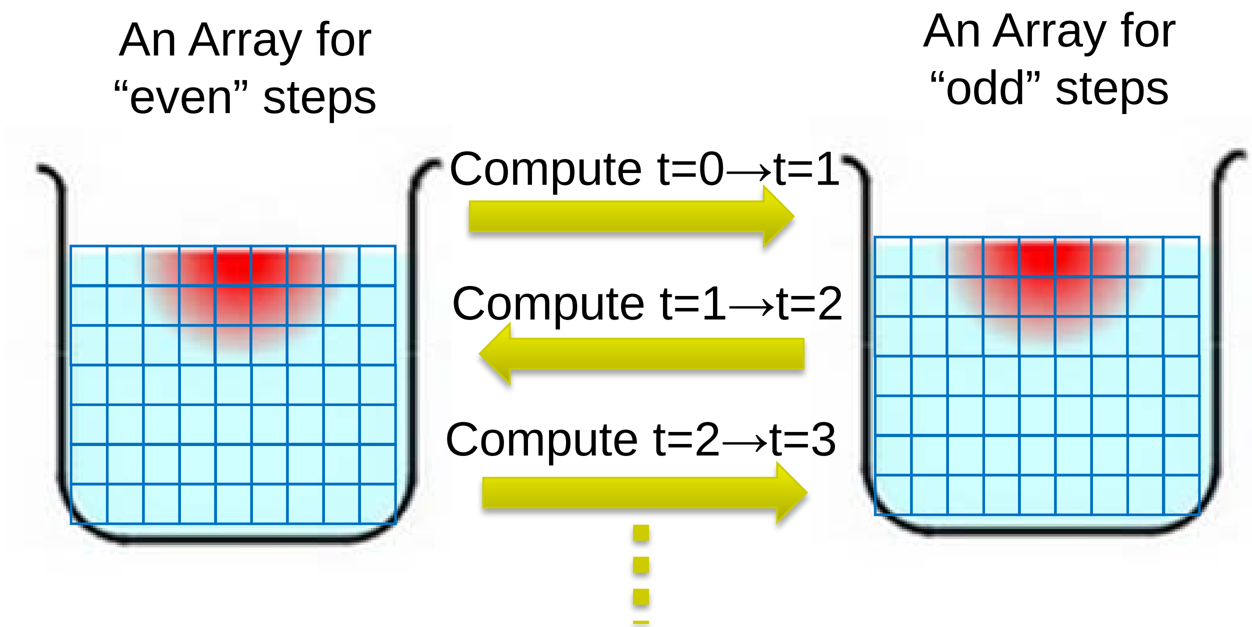


The ink spreads gradually, and finally the density becomes uniform (Figure by Prof. T. Aoki)

- Density of ink in each point vary according to time → Simulated by computers

Double Buffering Technique (Revisited)

- It is sufficient to have “current” array and “previous” array. “Double buffers” are used for many times



※ Sample program uses a global variables
float data[2][NY][NX];

Parallelizing Diffusion with OpenACC

- x, y loops are 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]
```

Parallelized on GPU

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

Unstructured Data Copy

- With “data” directive, copy timing is restricted

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

```
:
#pragma acc data copy(x,y)
   {
      :
   }
:
```

```
:
#pragma enter data copyin(x,y)

   :

#pragma exit data copyout(x,y)
:
```

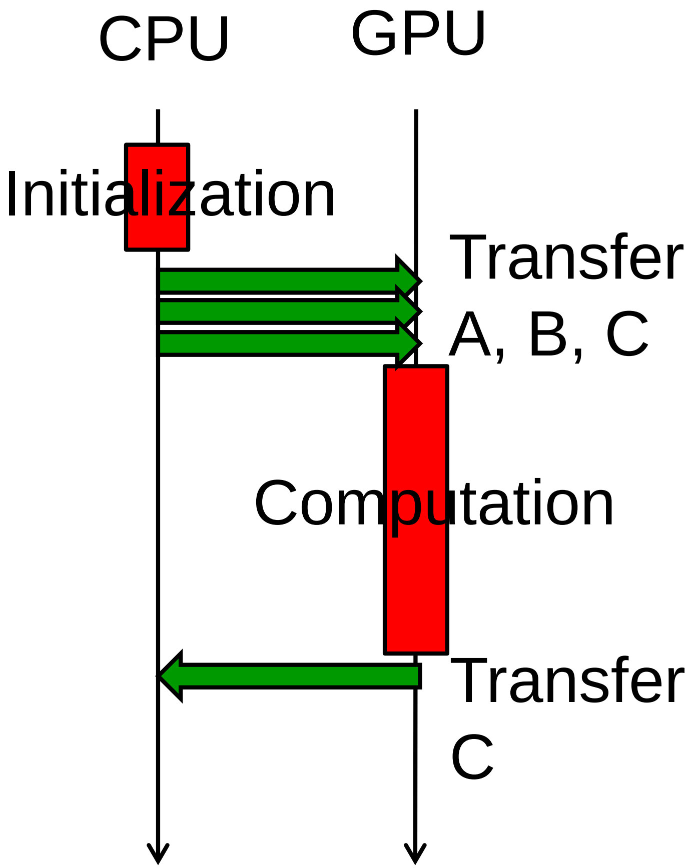
- ~endo-t-ac/ppcomp/18/mm-meas-acc sample uses them for time measurement

Data Transfer in mm-acc sample

- Host memory on CPU and device memory on GPU are different → data transfer is needed
- Current design
  - After initialization of A, B, C, we transfer them from CPU to GPU

  → Amount of data transfer: O(mk+kn+mn)
  - Computation: O(mnk)
  - After computation, we transfer C to CPU

  → Amount of data transfer: O(mn)



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[500:600][0:1000])

→ OK, rows[500,1100) are copied

…. data copyin(A[0:2000][300:400])

→ NG in current OpenACC

※ Currently, OpenACC does not support non-consecutive transfer

Supporting Larger Data
(Related to [G2])

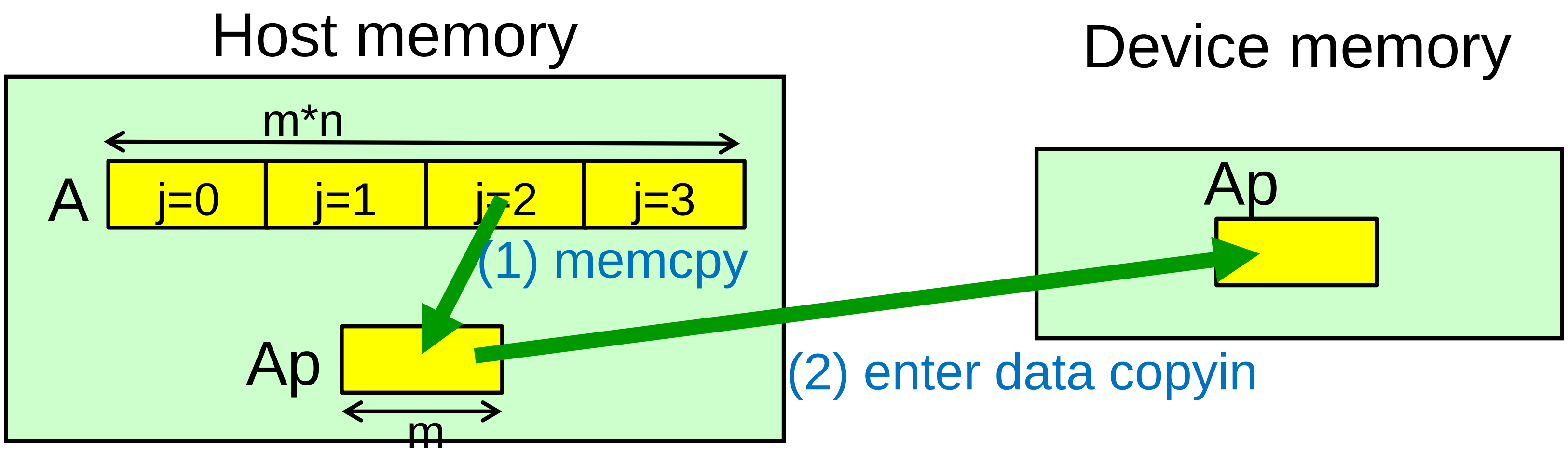
- Device (GPU) memory is smaller. How can we use larger data?

→ to split data

~endo-t-ac/ppcomp/18/array-acc sample

./array [m] [n], such as ./array 1000000 100

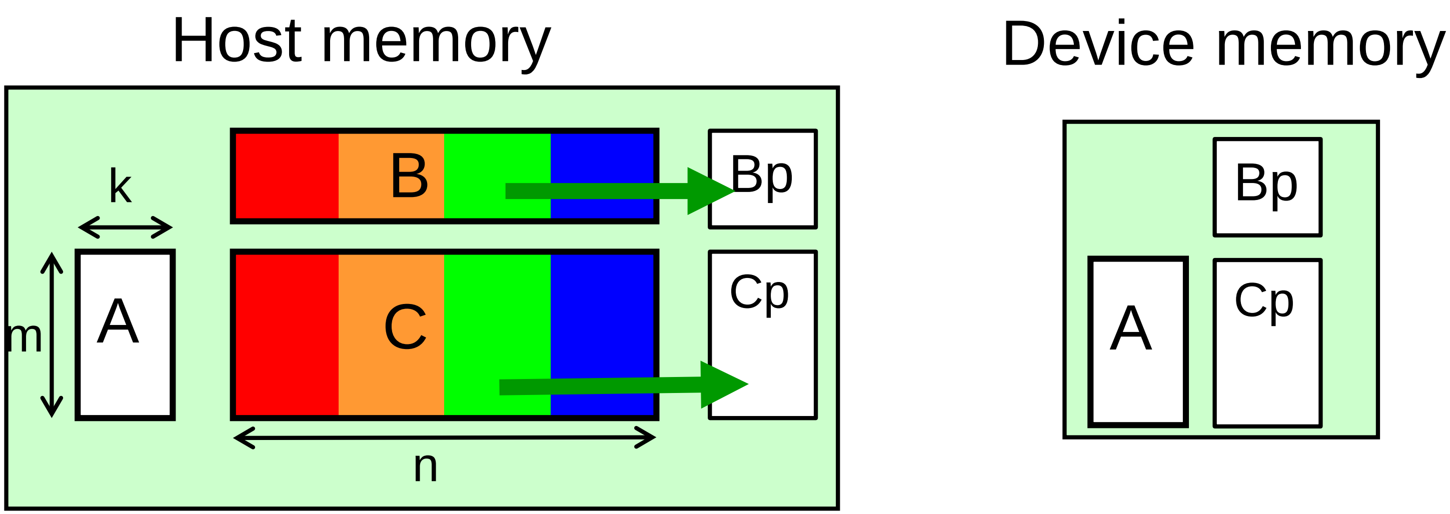
→ Create m*n length array A, and do A[i] *= 2



Note that Ap[i] ⇔ A[i + m*j]

- Direct copy partial A causes runtime errors → Under investigation

Larger Matrix Multiply (Concept)



- In this case, n is large → B, C are large
  - Such as ./mm 2000 60000 2000
  - Do we need to transfer A each step?
- How can we support large A?
  - How do we divide matrices?
  - How do we change data transfer algorithm?

Function Calls from Kernel Region

- Kernel region can call functions, but attention

```
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?

- Calling library functions is very limited ☹
- Exceptionally, some mathematical functions a re ok
  - fabs, sqrt, fmax…
  - #include <math.h> is needed

- We cannot use printf, strlen… ☹
  - If we want to see variables (for debug), we need to copy to CPU

Reduction in loop Directive

- “OpenMP-like” reduction is ok

operator

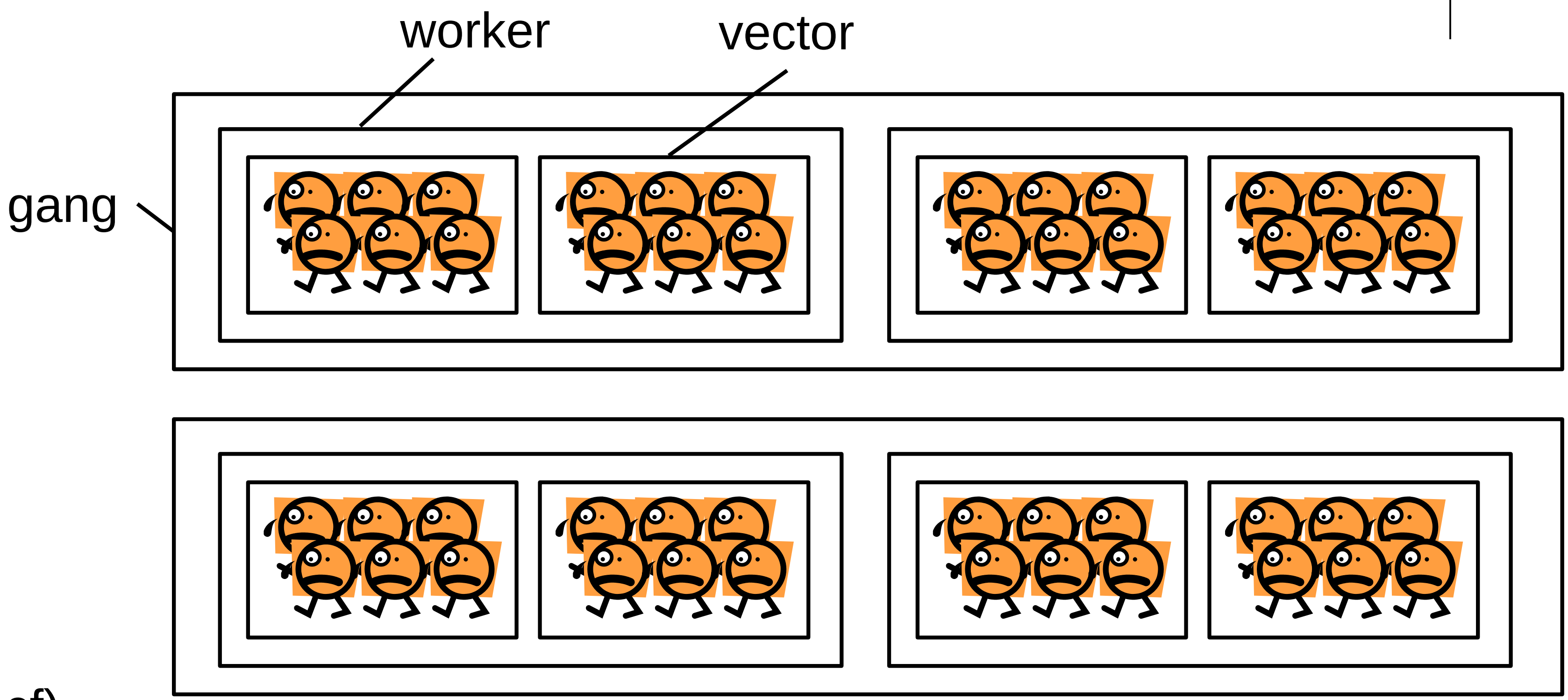
```
#pragma acc data …
#pragma acc kernels …

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

We should avoid race condition

※ “operator” may be +, *, max, min, &, |

Specify Hardware Mapping in loop Directive



cf)

#pragma acc loop independent gang,worker

for (i= 0….)

#pragma acc loop independent vector

for (j=0….)

※ Usually, default mapping is good ☺

Assignments in this Course

- There is homework for each part. Submissions of reports for 2 parts are required
- Also attendances will be considered

Part	Problems		
Part 1 OpenMP	[O1] diffusion [O2] sort [O3] free	*Select 1 problem*	*Select 2 parts*
Part 2 MPI	[M1] diffusion [M2] mm [M3] free	*Select 1 problem*	
Part 3 GPU	[G1] diffusion [G2] mm [G3] free	*Select 1 problem*	

Assignments in GPU Part (Abstract)

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

Due date: June 14 (Thursay)

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

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

[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:

- GPU Programming (3)
  - Introduction to CUDA