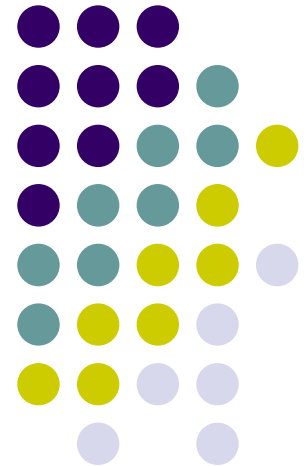


Practical Parallel Computing (実践的並列コンピューティング)

2025 Class No.8
[OpenACC Part] (2)
Discussion on Speed

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





Overview of This Course

- Introduction Part
 - 2 classes
 - OpenMP (OMP) Part
 - 4 classes
 - Report (required)
 - OpenACC (ACC) Part
 - 2 classes
 - Report (required)
 - CUDA Part
 - 3 classes
 - Report (elective)
 - MPI Part
 - 3 classes
 - Report (elective)
- ← We are here (2/2)

Note:

Modification to ppcomp-ex github



[ppcomp-ex/acc/bstort](#) has been missing, and I added it on github

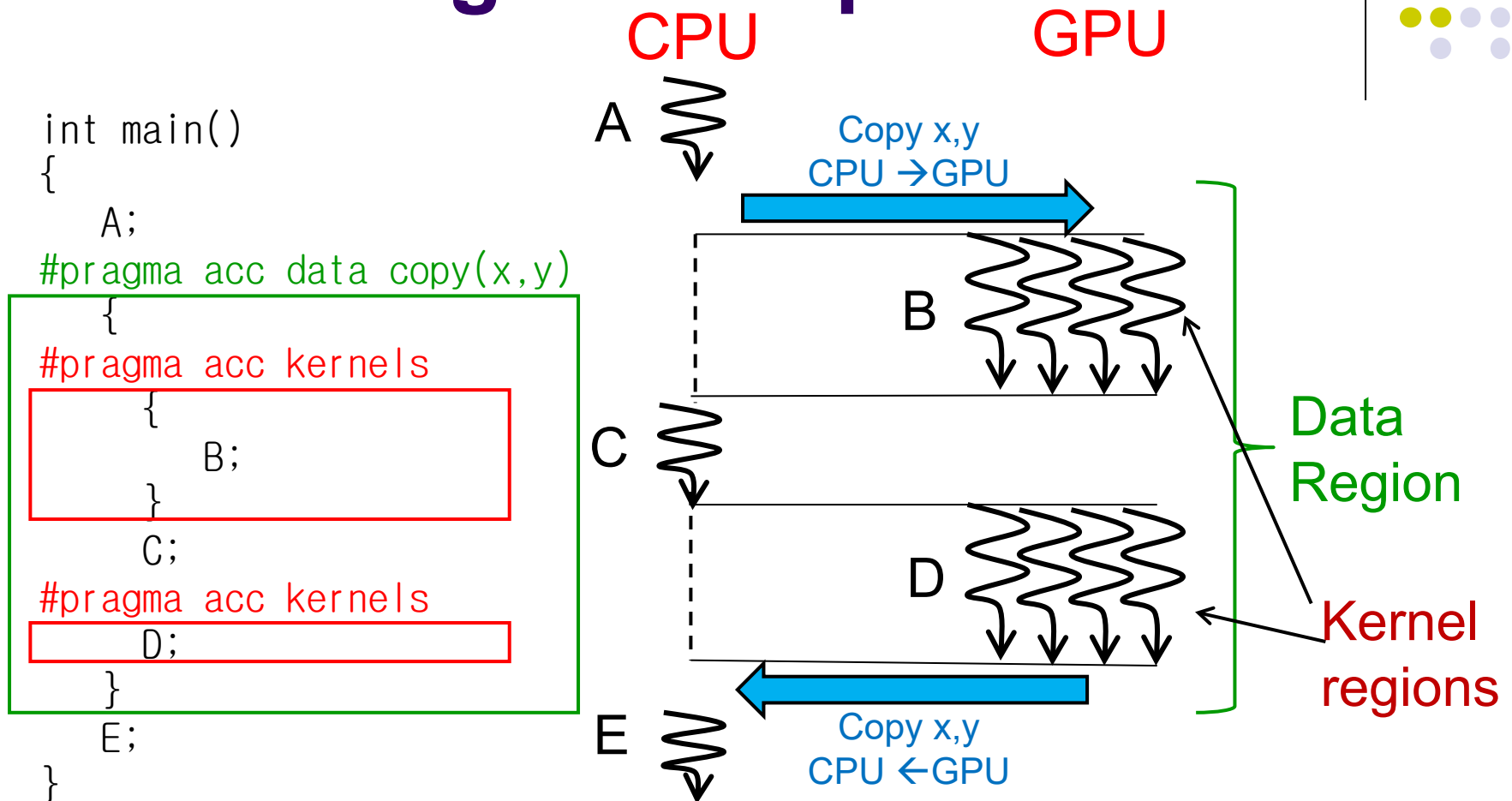
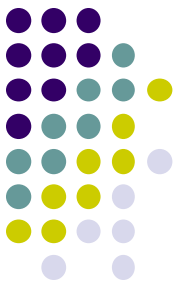
Please update your ppcomp-ex directory

```
cd ppcomp-ex // your ppcomp-ex directory  
git pull  
// → acc/bstort/, cuda/sort, mpi/sortis copied
```

Thanks for cooperation



Review: Data Region and Kernel Region in OpenACC



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



Review: 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];
}
```

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

- **#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

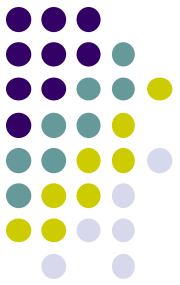
Notes on Assignment [A1][A2]



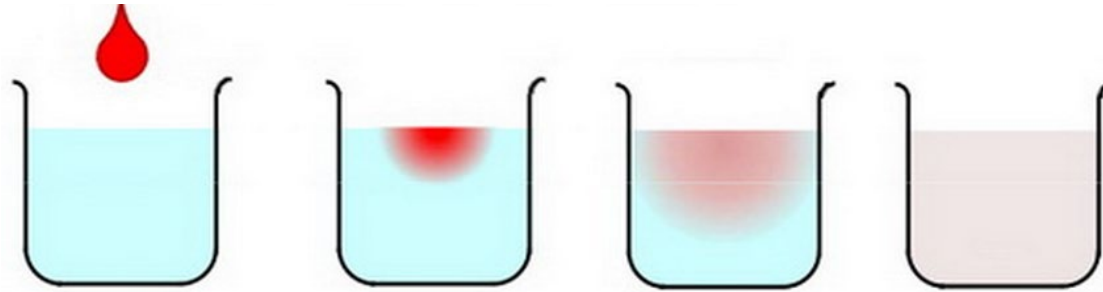
- You will need compiler options different from the `diffusion` directory for OpenACC
- You can use files in `acc/diffusion` or `acc/bSORT` directory as basis
 - `.c` file is NOT parallelized
 - “Makefile” in these directories supports compiler options for OpenACC
 - Don’t forget “`module load nvhpc`” before “make”
 - The effect of module is valid until you ‘exit’ from the shell

“diffusion” Sample Program

Target of [A1], details are in ppcomp25-4



An example of diffusion phenomena:



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

Base version: ppcomp-ex/base/diffusion/
You can use ppcomp-ex/acc/diffusion/

```
cd ppcomp-ex/acc/diffusion
module load nvhpc
make
./diffusion 20    // number of time steps
```

Consideration using OpenACC



- Where do we put **#pragma acc loop independent** ?
 - Which loops are parallelized?
- Where do we put **#pragma acc kernels** ?
 - It defines kernel region, executed on the GPU
 - Kernel region has to include "... acc loop"
- Where do we put **#pragma acc data** ?
 - It defines data region
 - Data touched by GPU must be on device memory
 - Too frequent data copy may decrease program speed



How Can We Parallelizing Diffusion?

- 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++) {  
            :  
        }  
    }  
}
```

Specify kernel region by
#pragma acc kernels
Parallelize x, y loops by
#pragma acc loop ...

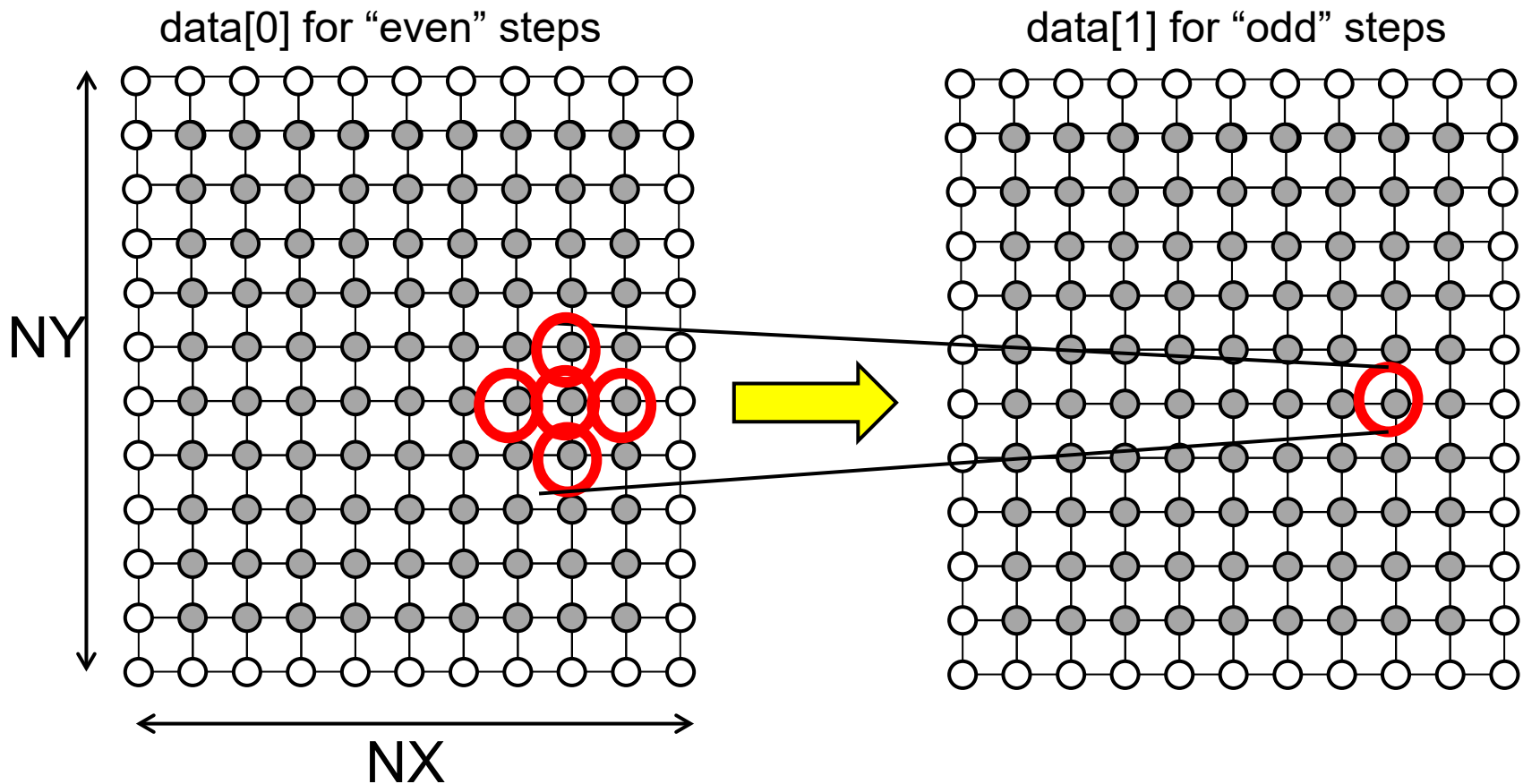
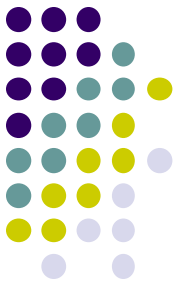
Specify data region by
#pragma acc data ...

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

[Data transfer from GPU to CPU]

“diffusion” has Multi-Dimension Array

```
float data[2][NY][NX]; // 2 for double buffering
```



data Clause for Multi-Dimensional arrays



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

`#pragma acc data copy(A)`

→ OK, all elements of A are copied

`#pragma acc data copy(A[0:2000][0:1000])`

→ OK, all elements of A are copied

`#pragma acc data copy(A[500:600][0:1000])`

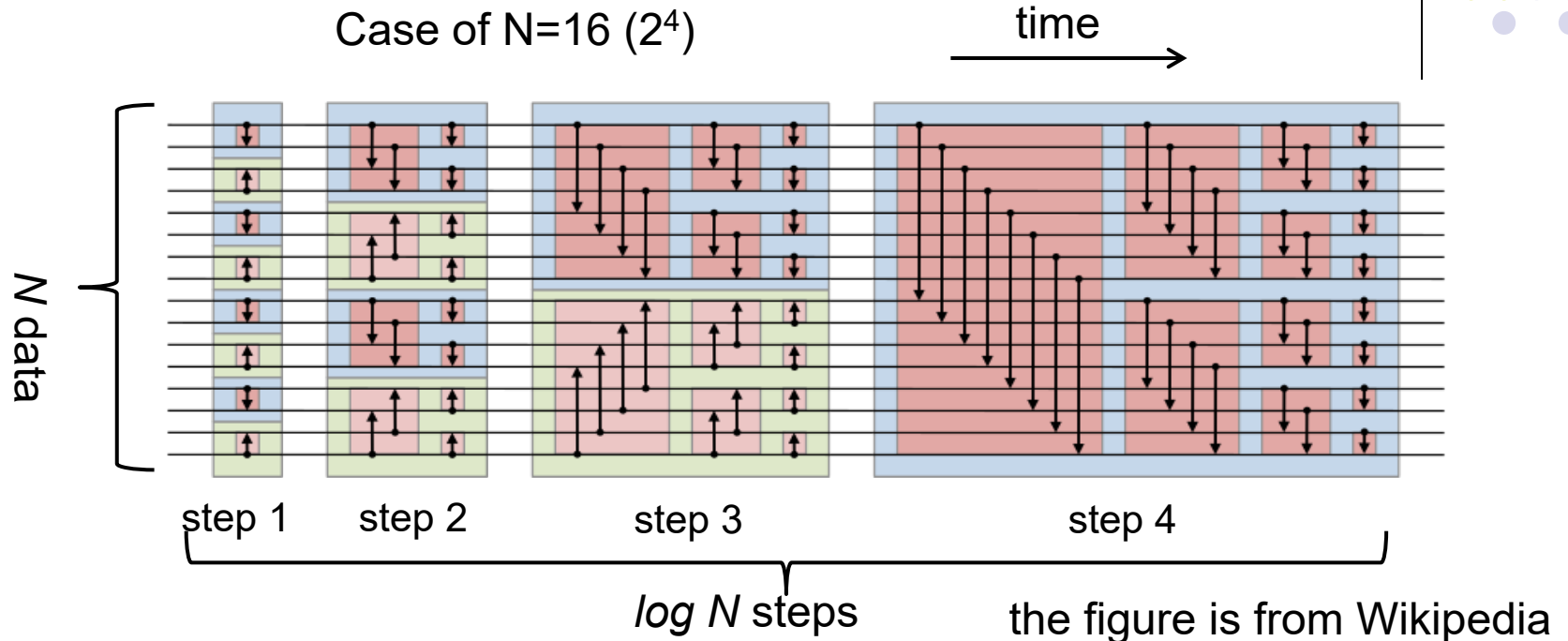
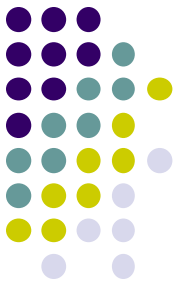
→ OK, rows[500,1100) are copied

`#pragma acc data copy(A[0:2000][300:400])`

→ Recently OK

“bsort” Sample Program

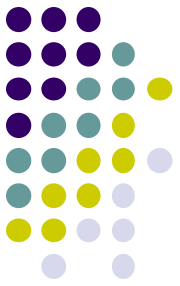
Target of [A2], details are in ppcomp25-4



Base version: [ppcomp-ex/base/bsort/](https://github.com/ppcomp-ex/base/bsort/)

You can use [ppcomp-ex/acc/bsort/](https://github.com/ppcomp-ex/acc/bsort/) (see notes in p.3)

```
cd ppcomp-ex/acc/bsort
module load nvhpc // if not yet
make
./bsort 1000000 // number of elements to be sorted
```



How Can We Parallelizing bsort?

- k loops can be parallelized
- i, j loops cannot be parallelized

```
for (i = 1; (1<<i) <= N2; i++) { / step loop
    for (j = i-1; j >= 0; j--) { // sub-step loop
        :
        for (k = 0; k < N2; k++) {
            // compare 2 data and swap
        }
    }
}
```

Specify kernel region by
`#pragma acc kernels`
Parallelize k loops by
`#pragma acc loop ...`

Next, where should the **data region** be?

- Please consider reducing data copy cost

Data Transfer Costs in GPU Programming

Related to [A3], also [A1] [A2]



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

from ppcomp25-7 slides

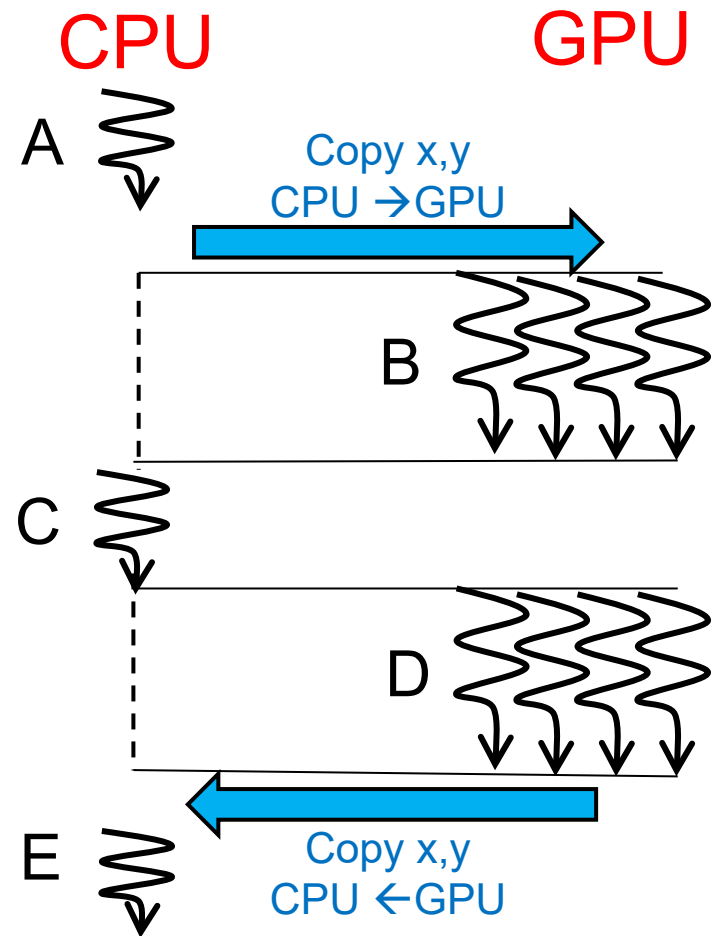
[A3] Evaluate speed of “acc/mm” sample in detail

ppcomp-ex/acc/mm/

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

Optional:

- To use different loop orders
- To change/improve the program
 - Cache blocking?



Speed of GPU Programs: case of acc/mm



In mm-acc, speed in Gflops is computed by

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

T_{total} includes both computation time and transfer

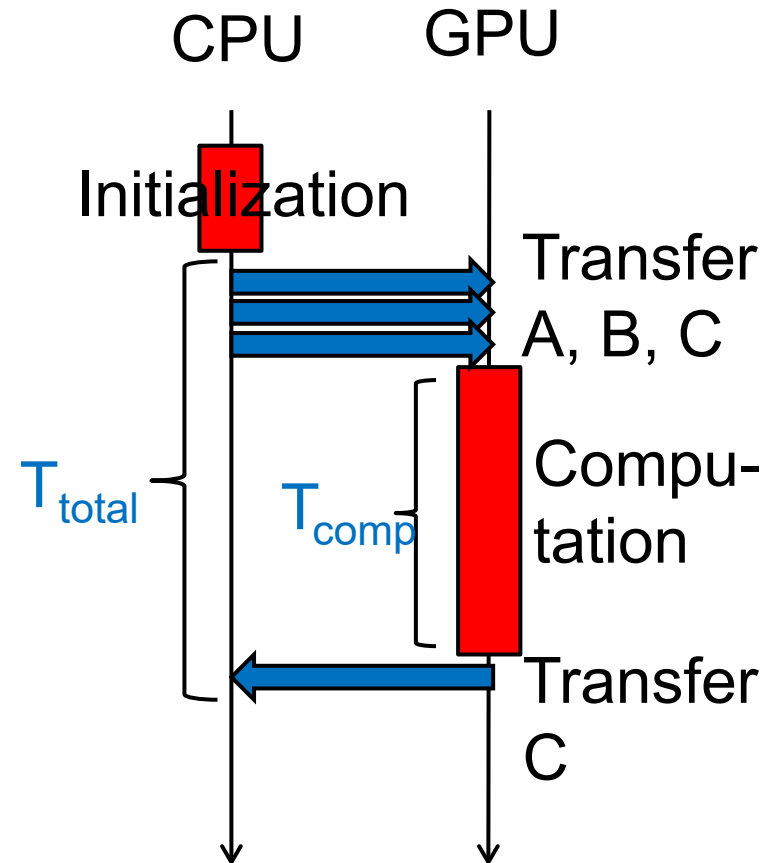
→ S counts slow-down by transfer

To see the effects, let's try another sample

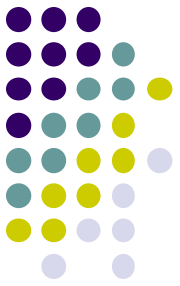
ppcomp-ex/acc/mm-meas/

which outputs time for

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



In [A3], please evaluate effects of transfer costs



Measurement of Transfer Time

- Data transfer occurs at the beginning and the end of “data region”

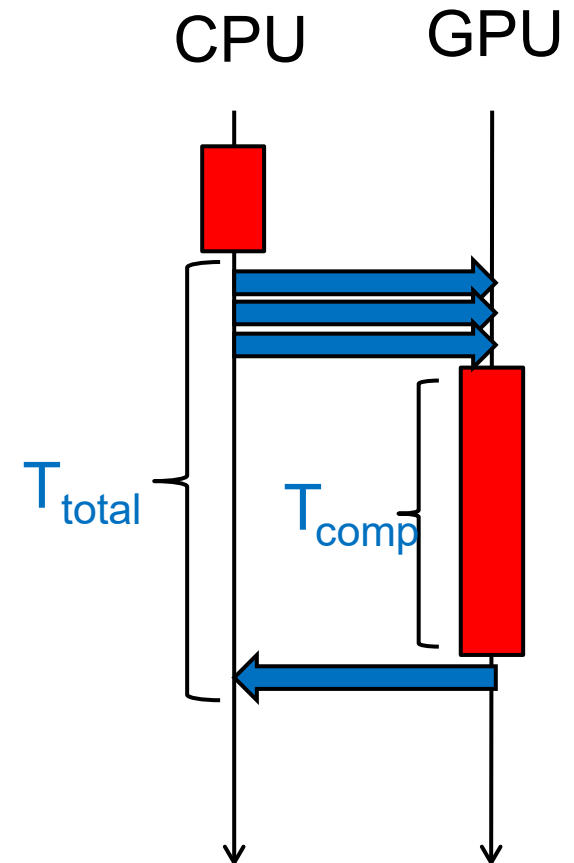
```
// A,B,C are on CPU
```

```
#pragma acc data copyin(A,B) copy(C)  
{ // copyin (CPU->GPU) here
```

```
#pragma acc kernels
```

```
{  
:  
}
```

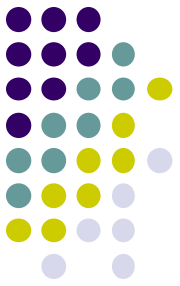
```
} //copyout (GPU->CPU) here
```



See [acc/mm-meas/mm.c](#)

Also note that `gettimeofday()` must be called on CPU

Discussion on Data Transfer Costs



- Time for data transfer $T_{\text{trans}} \doteq M / B + L$
 - M : Data size in bytes
 - B : “Bandwidth” (speed)
 - L : “Latency” (if M is sufficiently large, we can ignore it)
- In a H100 GPU,
 - Theoretical bandwidth B is 64GB/s (64×10^9 Bytes per second)
 - Actual transfer speed is slower than this value

Discussion on Computation and Transfer Costs



In mm-acc,

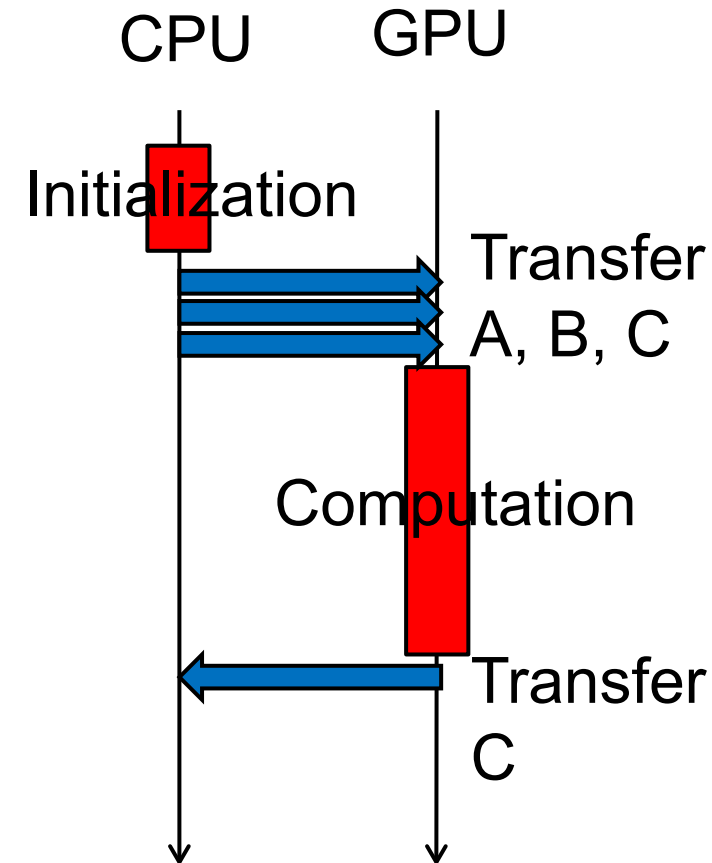
- Computation amount: $2mnk$
 - Data transfer amount:
 - A, B, C: CPU \rightarrow GPU: $8(mk+kn+mn)$
 - C: GPU \rightarrow CPU: $8(mn)$
- sizeof(double) = 8

Observations:

- We can compute actual transfer speed from $B \doteq M / T_{\text{trans}}$
 - L is ignored here
- Balance between computation and data transfer changes with different m, n, k

When m, n, k are 2x:

- Computation time is 8x
- Transfer time is 4x



Discussion of data copy cost in diffusion [A1]



- Which is faster?

Data Region

Kernel Region

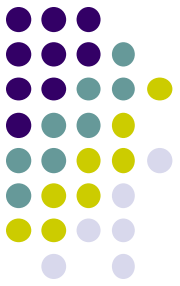
```
for (t = 0; t < nt; t++) {  
    for (y = 1; y < NY-1; y++) {  
        for (x = 1; x < NX-1; x++) {  
            :  
        }  
    }  
}
```

```
for (t = 0; t < nt; t++) {  
    for (y = 1; y < NY-1; y++) {  
        for (x = 1; x < NX-1; x++) {  
            :  
        }  
    }  
}
```

Computation: $O(NX \ NY \ nt)$
Data copy: $O(NX \ NY)$
➔ faster

Computation: $O(NX \ NY \ nt)$
Data copy: $O(NX \ NY \ nt)$

Note on OpenACC: Function Calls from GPU

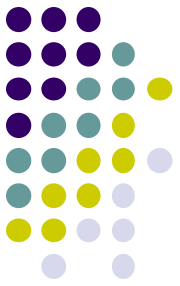


- Calling functions in kernel region is ok, but we need to be careful
 - “**acc routine**” directive is required by compiler to generate GPU code

```
int main()
{
    #pragma acc kernels
    {
        ... func(A[i]) ...
    }
}

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

A diagram illustrating a function call within a GPU kernel. A red box highlights the kernel region in the main function, which contains the call `... func(A[i]) ...`. Two red arrows originate from this call: one points to the opening curly brace of the `func` function definition, and the other points to the ellipsis (`...`) following the function call.



How about Library Functions?

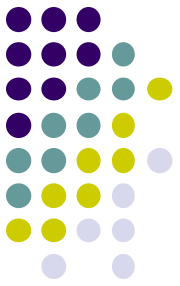
Inside kernel regions (`#pragma acc kernel`),

- Available library functions is very limited 😞
- We cannot use `strlen()`, `memcpy()`, `fopen()`, `fflush()`... 😞
- We cannot use `gettimeofday()` 😞
- Exceptionally, some mathematical functions are ok 😊
 - `fabs`, `sqrt`, `fmax`...
 - `#include <math.h>` is needed
- Recently, `printf()` in kernel regions is ok! 😊

Limitation of OpenACC (Some are Limitations of GPU)



- Task parallel program is not supported
 - like fib, qsort
 - Also CUDA is hard
- Mutual exclusion is not supported (generally)
 - OpenACC supports “acc loop reduction(…)”
 - CUDA supports atomic operations
- Hard to use some hardware features
 - Synchronize in thread block, SM’s shared memory, Tensor core...
 - CUDA can use



Assignments in OpenACC Part (1)

Choose one of [A1]—[A4], and submit a report

Due date: May 12 (Monday)

[A1] Parallelize “diffusion” sample program by OpenACC

[A2] Parallelize “bsort” sample program by OpenACC

[A3] Evaluate speed of “acc/mm” sample in detail

[A4] (Freestyle) Parallelize *any* program by OpenACC

For more details, please see [ppcomp25-7](#) slides



Next Part: CUDA Part

- Class #9
 - Introduction to CUDA, kernel functions
- Class #10
 - Characteristics of grid, thread blocks, threads
- Class #11
 - Performance improvement on GPU
- Schedule
 - Mon, May 5: No classes (national holiday)
 - Thu, May 8: Class #9, CUDA (1)