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

Part 2: GPU

No 2: OpenACC and CUDA

May 9, 2024

Toshio Endo

School of Computing & GSIC

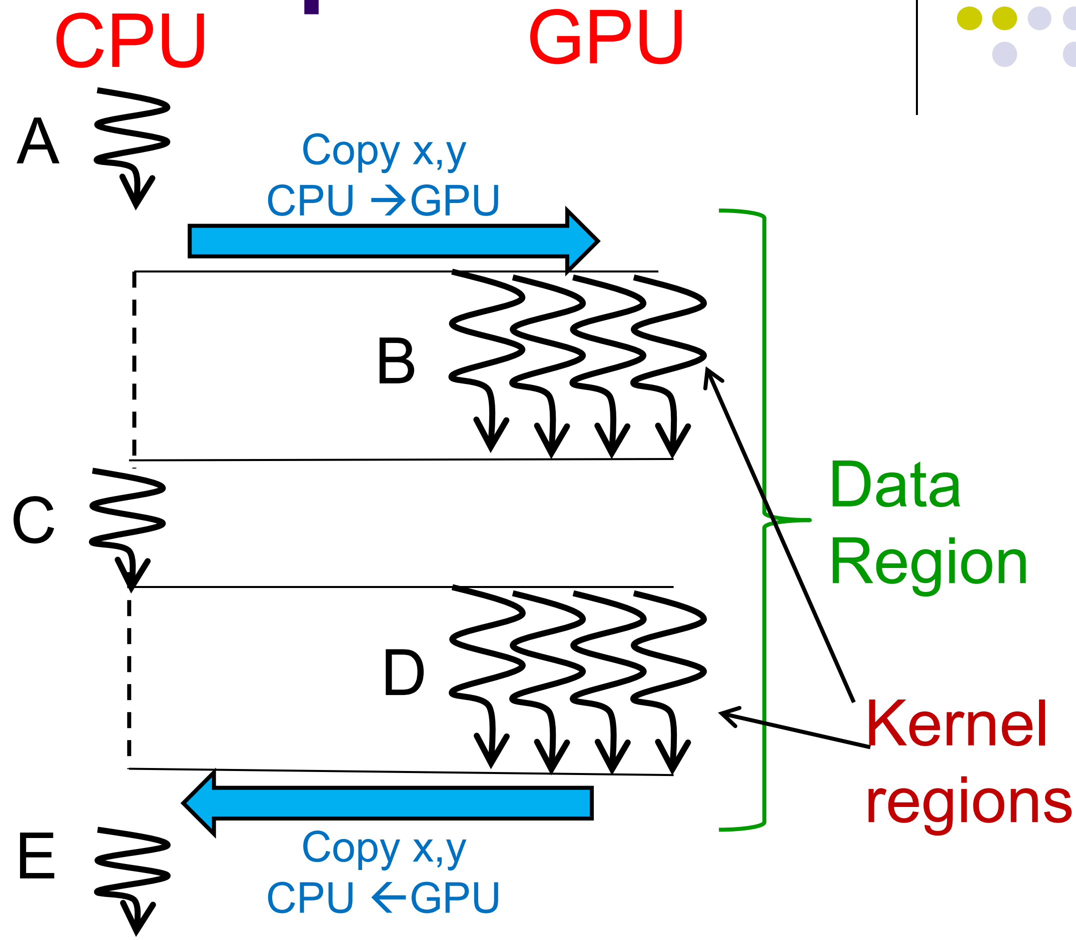
endo@is.titech.ac.jp

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

Review: 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

Review: Loop Directive

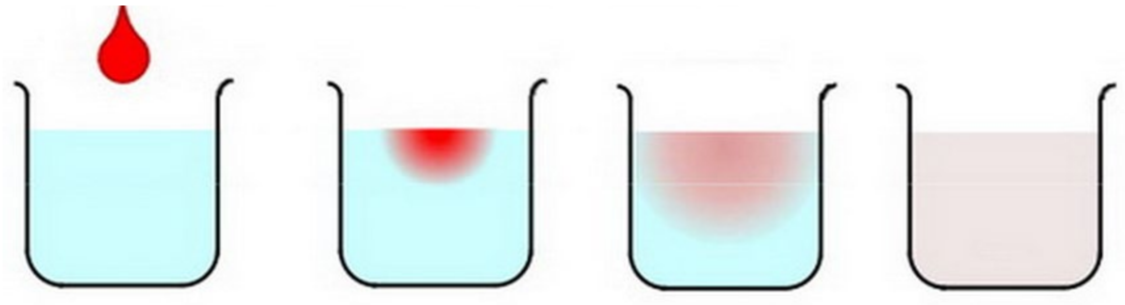
```
   int a[100], b[100], c[100];
   int i;
#pragma acc data copy(a,b,c)
#pragma acc kernels
#pragma acc loop independent
  for (i = 0; i < 100; i++) {
        a[i] = b[i]+c[i];
  }
```

- #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

"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/bs/tga-ppcomp/24/diffusion/
You can use /gs/bs/tga-ppcomp/24/diffusion-acc/

- Execution：./diffusion [nt]
  - nt: 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

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

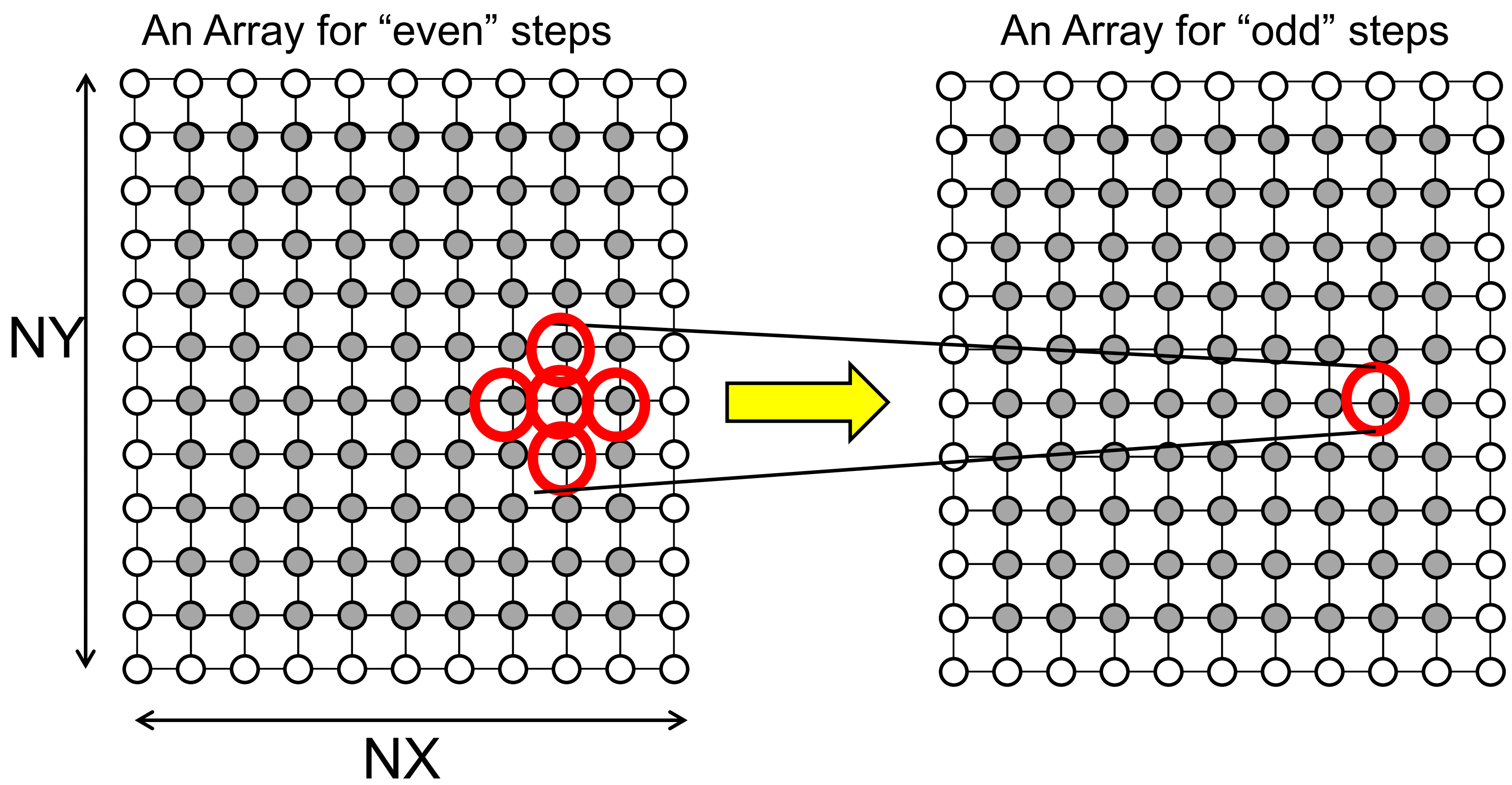
```
[Data transfer from CPU to GPU]
for (t = 0; t < nt; t++) {

  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 Structure in “diffusion”

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])

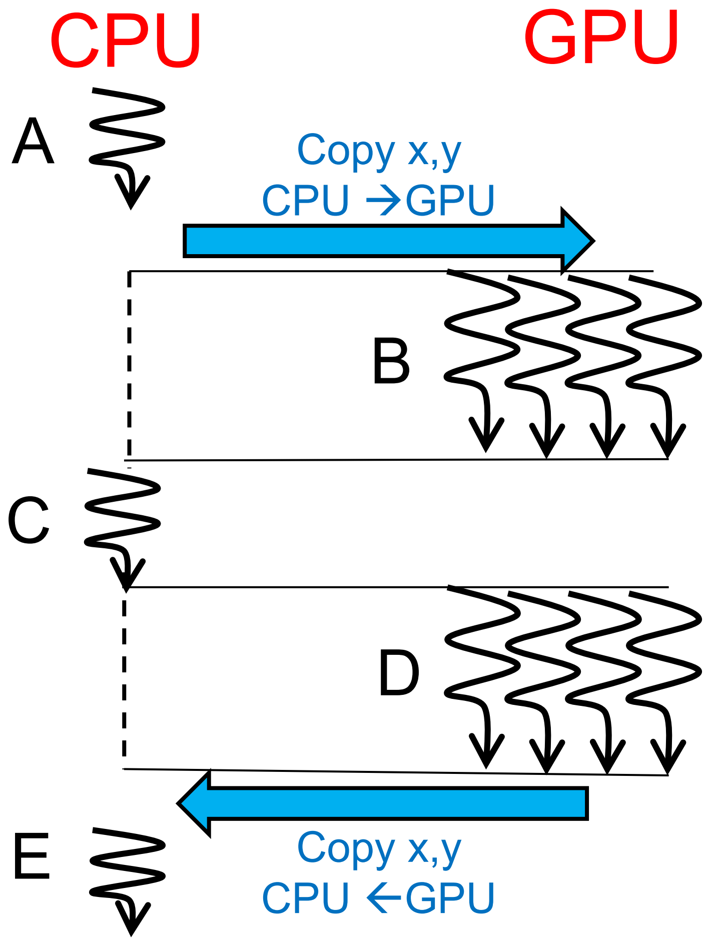
→ Recently OK

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
  - /gs/bs/tga-ppcomp/24/diffusion-acc/
  - "Makefile" in this directory supports compiler options for OpenACC
  - Don't forget "module load gcc nvhpc" before "make"

Data Transfer Costs in GPU Programming
Related to [G2]

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



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

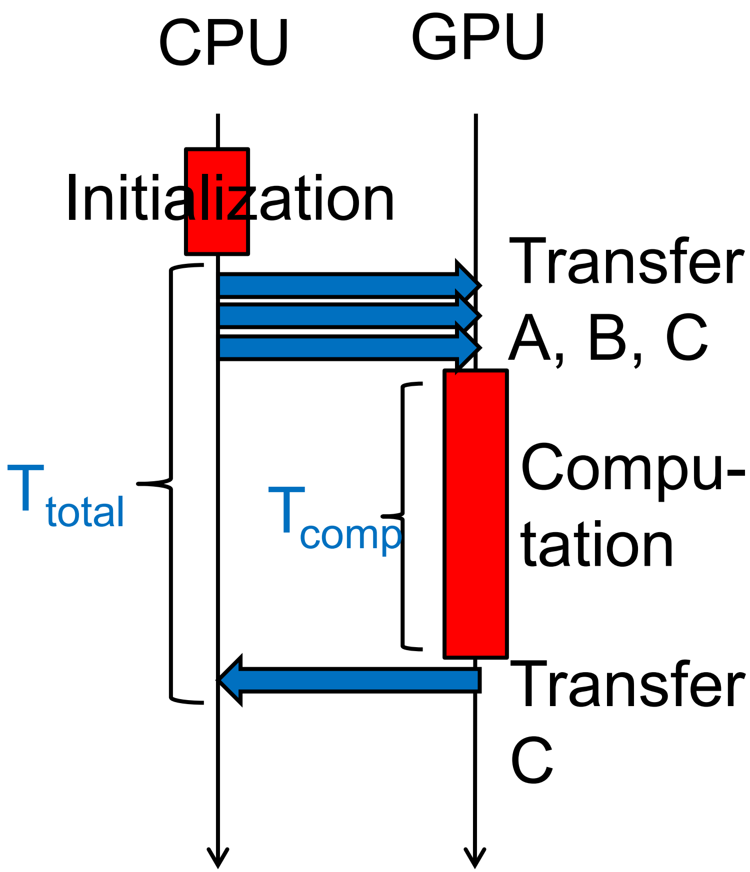
→ S counts slow-down by transfer

To see the effects, let's try another sample

/gs/bs/tga-ppcomp/24/mm-meas-acc

which outputs time for

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



In [G2], please evaluate effects of transfer costs

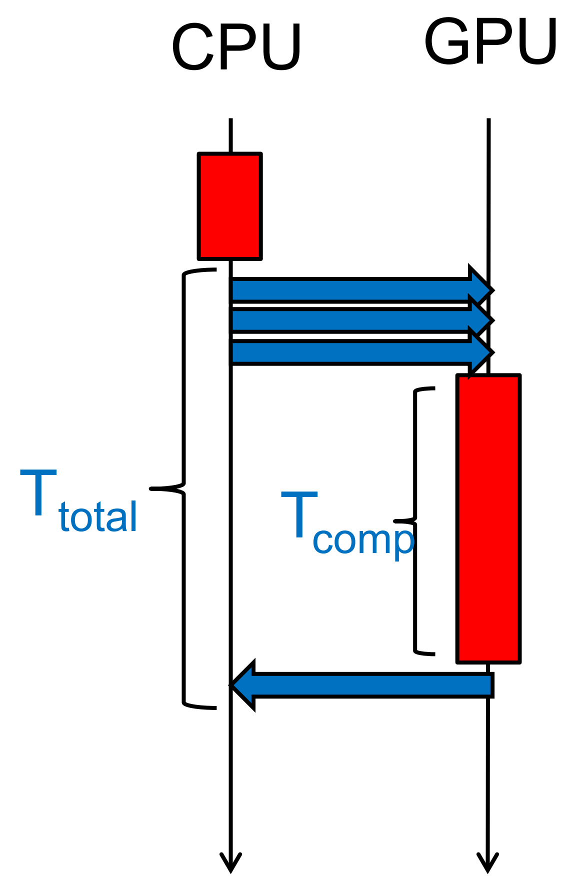
Measurement of Transfer Time

- 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
```

T_{total}

T_{comp}



See mm-meas-acc/mm.c

Also note that gettimeofday() must be called 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, <u>we can ignore it</u>)
- In a H100 GPU,
  - Theoretical bandwidth B is 64GB/s (64×10^9 Bytes per second)
    - Actual transfer speed is slower than this valye

Discussion on Computation and Transfer Costs

In mm-acc,

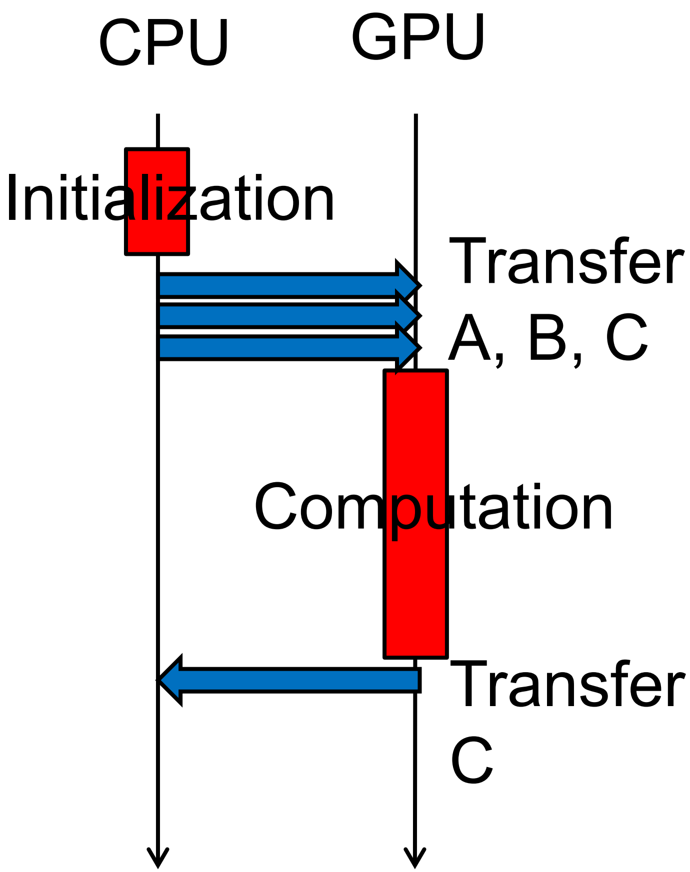
- Computation amount: 2mnk
- Data transfer amount:
  - A, B, C: CPU → GPU: 8(mk+kn+mn)
  - C: GPU → CPU: 8(mn)

sizeof(double) = 8

Observations:

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

  When m, n, k are doubled:
  - Computation time is 8x
  - Transfer time is 4x



Note:
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 …;
}
```

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! ☺

Now explanation of OpenACC is finished
We will go to CUDA

OpenACC and CUDA for GPUs

- OpenACC
  - C/Fortran + directives (#pragma acc …), Easier programming
  - NVIDIA HPC SDK compiler works
    - module load nvhpc
    - nvc -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];
  }
```

Executed on GPU in parallel

A CUDA Program Look Like

Sample:
/gs/bs/tga-ppcomp/24/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 (rXn11)]

module load cuda   *[Do once after login]*

cd ~/ppc24

cp -r /gs/bs/tga-ppcomp/24/add-cuda .

cd add-cuda

make

[An executable file "add" is created]

./add

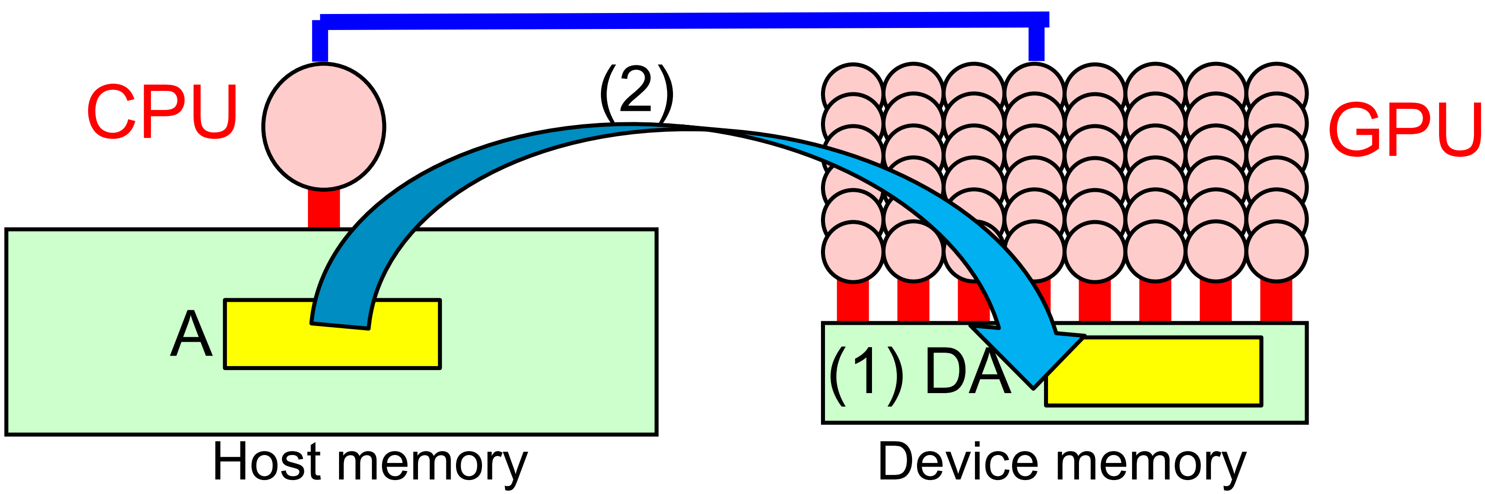
Preparing Data on Device Memory

(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

[CPU side]

```
func<<<20, 5>>>(…);
```

# of thread blocks
# of threads per block

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

call

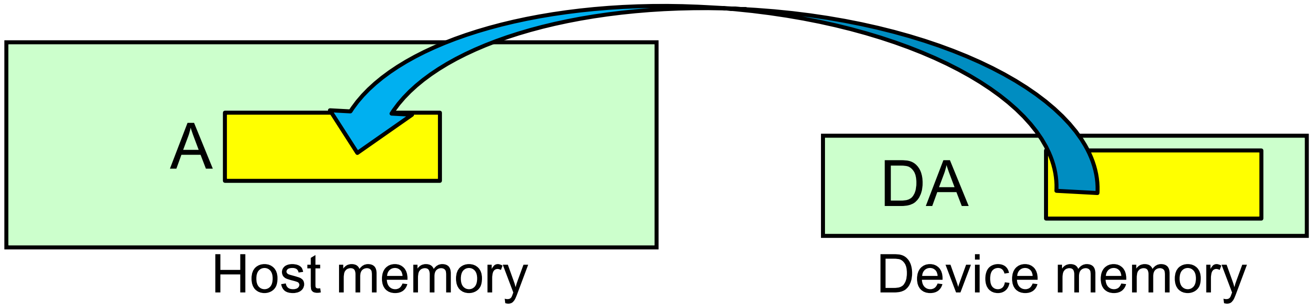
return

[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

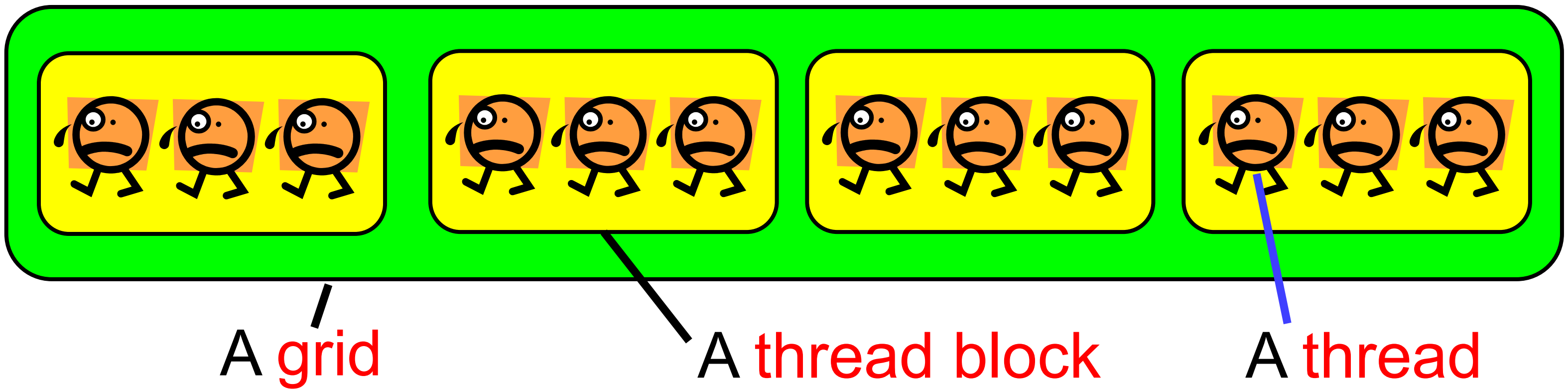


- Copy data using cudaMemcpy
  - cf) cudaMemcpy(A, DA, *size*, cudaMemcpyDefault);
  - 4th argument is one of
    - cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice, cudaMemcpyHostToHost
    - cudaMemcpyDefault ← Detect memory type automatically ☺

- When a memory area is unnecessary, free it
  - cf) cudaFree(DA);

Threads in CUDA

When calling a GPU kernel function, specify 2 numbers (at least) for number of threads



cf) func <<< 4, 3 >>> (); → 12 threads

Number of thread blocks = gridDim

Number of threads per block = blockDim

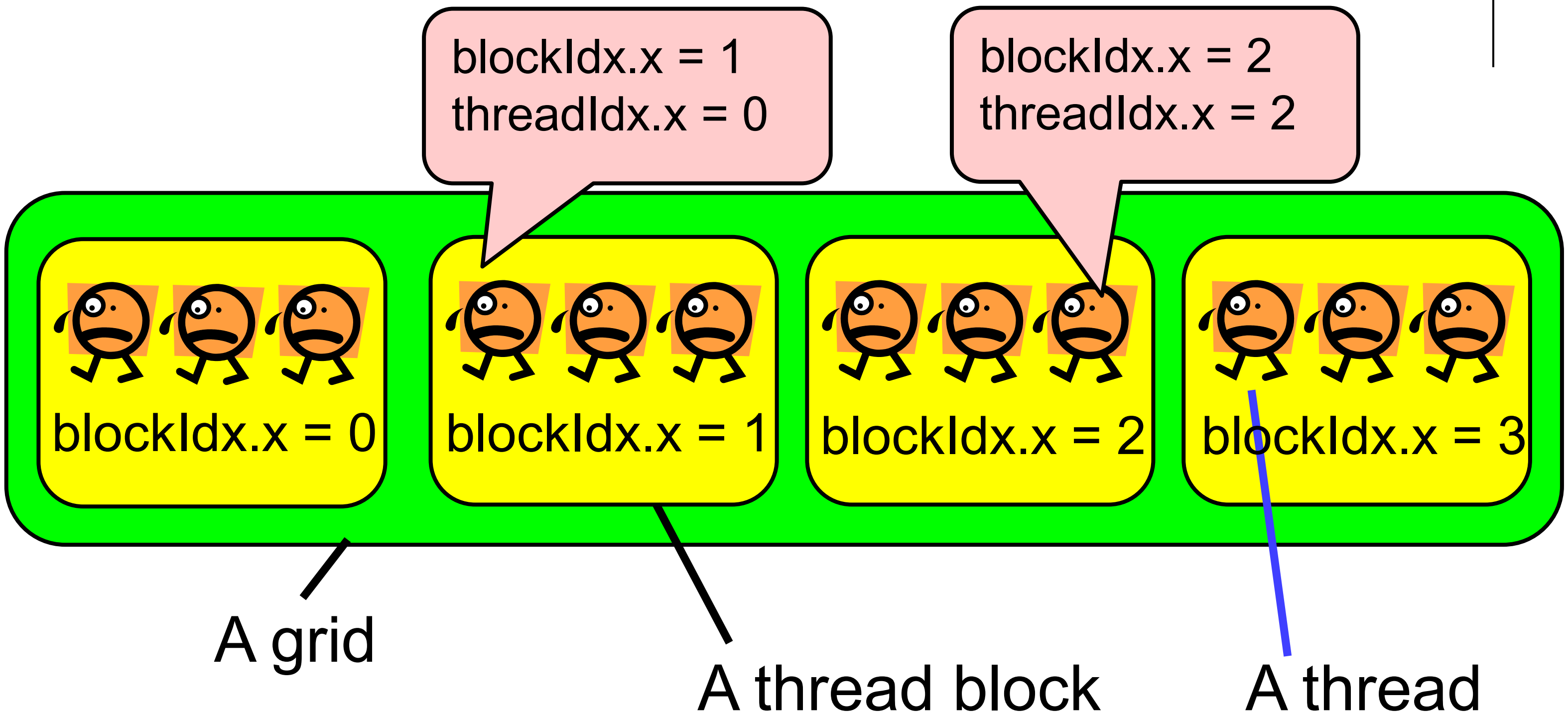
The reason is related to GPU hardware

Thread block ⇔ SMX, Thread ⇔ CUDA core

To See Who am I

- By reading the following special variables, each thread can see its thread ID in GPU kernel function
- My ID
  - blockIdx.x: Index of the block the thread belong to (≧0)
  - threadIdx.x: Index of the thread (inside the block) (≧0)
- Number of thread/blocks
  - gridDim.x: How many blocks are running
  - blockDim.x: How many threads (per block) are running

Thread Block ID, Thread ID



For every thread, gridDim.x = 4, blockDim.x = 3

Note: In order to see the entire sequential ID, we should compute
blockIdx.x * blockDim.x + threadIdx.x

The Case of add-cuda Sample

- /gs/bs/tga-ppcomp/24/add-cuda
- We want to do

```
for (i = 0; i < 100; i++) { DA[i] += DB[i]; }
```

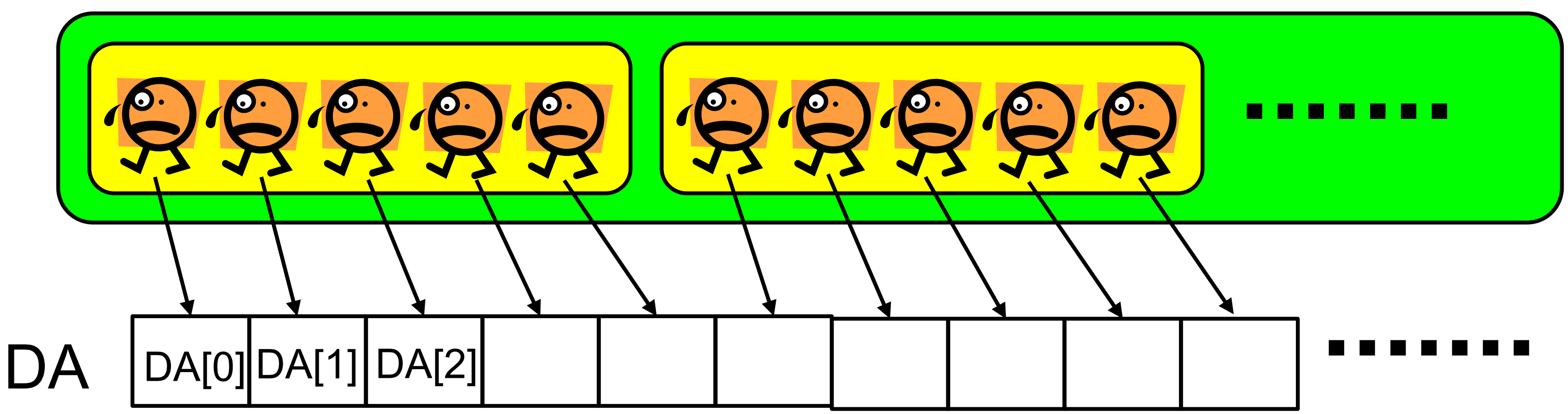
[CPU side]

```
add<<<20, 5>>>(…);
```

20x5=100 threads
will execute add function

[GPU side]

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



Comparing OpenMP/OpenACC/CUDA

	OpenMP	OpenACC	CUDA
Processors	CPU	CPU+GPU	
File extension	.c, .cc		.cu
To start parallel (GPU) region	#pragma omp parallel	#pragma acc kernels	func<<<…, …>>>()
To specify # of threads	export OMP_NUM _THREADS=…	(num_gangs, vector_length etc)	
Desirable # of threads	# of CPU cores or less	# of GPU cores or “more”	
To get thread ID	omp_thread_num()	-	blockIdx, threadIdx
Parallel for loop	#pragma omp for	#pragma acc loop	-
Task parallel	#pragma omp task	-	-
To allocate device memory	-	#pragma acc data	cudaMalloc()
To copy to/from device memory	-	#pragma acc data #pragma acc update	cudaMemcpy()
Functions on GPU	-	#pragma acc routine	__global__,__device__

※ “# of XXX” = “The number of XXX”

Assignments in GPU Part (Abstract)

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

Due date: May 30 (Thursday)

[G1] Parallelize "diffusion" sample program by OpenACC or CUDA

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

※ In OpenACC case, mm-meas-acc sample is useful

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

For more detail, please see ppcomp-2-1 slides

Next Class:

- GPU Programming (3)
  - mm sample on CUDA

- Schedule
  - Mon, May 13: GPU (3)
  - Thu, May 16: No classes (cancelled/休講)
  - Mon, May 20: GPU (4)
- Also please note due date of OpenMP assignment is Today!