Computational Science Alliance
The University of Tokyo

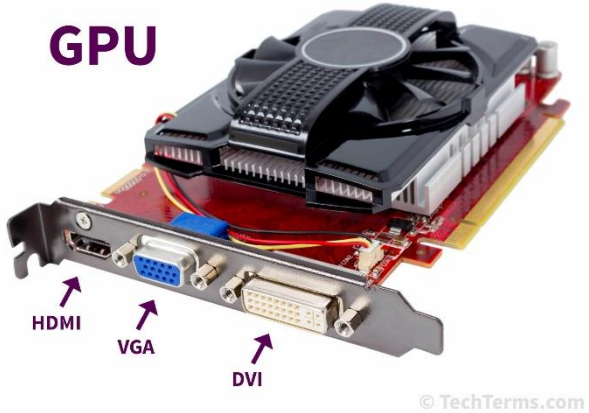
Summer School 2018, Computational Science Alliance, UTokyo
Hands on Lecture on Aug. 22-24, 2018

GPU Computing by OpenACC・CUDA
- 1st day -

Masaharu MATSUMOTO
Project Lecturer, Department of Computer Science

matsumoto@is.s.u-tokyo.ac.jp

What is GPU？



- GPU : Graphics Processing Unit
  - ✓ Graphic board, Video card

- Hardware specialized to image processing
  - ✓ Fast and High resolution drawing, 3D drawing processing, perspective transformation, shade/lighting, screen output, and so on…
  - ✓ Installed in PC, video game, smart phone…

- GPU computing (GPGPU): General-Purpose computing on Graphics Processing Unit

Why GPU Computing？

- High Performance!

	P100	BDW	KNL
Operating frequency（GHz）	1.480	2.10	1.40
The number of cores（effective threads）	3,584	18 (18)	68 (272)
Peak performance（GFLOPS）	**5,304**	604.8	3,046.4
Main memory（GB）	16	128	16
Memory band width（GB/sec., Stream Triad）	534	65.5	490
Notes	GPU on Reedbush-H	CPU on Reedbush-U/H	on Oakforest-PACS (Intel Xeon Phi)

GPU computing is difficult ?

- CPU: Some large cores are installed.
  - ✓ CPU on Reedbush-H: 2.10 GHz, 18 cores
  - ✓ Large core… Branch prediction, pipeline processing, Out-of-Order
    - It can do (almost) anything
  - ✓ Good for serial processing
- GPU: Many small cores are installed.
  - ✓ GPU on Reedbush-H: 1.48 GHz, 3,584 cores
  - ✓ Small core… the above functions are **weak or none**!
  - ✓ **Parallelization is required.**

Difficulty of GPU

1. To use many cores efficiently
2. Coding for parallel programing

NVIDIA Tesla P100

- 56 SMs
- 3584 CUDA Cores
- 16 GB HBM2

From P100 whitepaper

Streaming Multiprocessor (SM) in NVIDIA Tesla P100

GPU TECHNOLOGY CONFERENCE

GP100 SM

	GP100
CUDA Cores	64
Register File	256 KB
Shared Memory	64 KB
Active Threads	2048
Active Blocks	32

GPU for Computation

Easy

- GPU-Accelerated Libraries
  - ✓ Just call a library
  - ✓ Low flexibility of coding
  - ✓ https://developer.nvidia.com/gpu-accelerated-libraries
- OpenACC
  - ✓ Insert a directive in a conventional C/C++/Fortran code
  - ✓ Detailed tuning is difficult

Today's theme

- CUDA C/CUDA Fortran
  - ✓ High flexibility of coding
  - ✓ Program codes have to be rewritten

Difficult

Overview: OpenACC

- OpenACC is a directive-based programming standard for parallel computing in heterogeneous CPU/GPU systems.
- As in OpenMP, the programmer can add directives to the areas that should be accelerated in C, C++ and Fortran codes.
- OpenACC is easier to program than CUDA and have high portability. In an environment that does not correspond to OpenACC, the directives are ignored automatically.

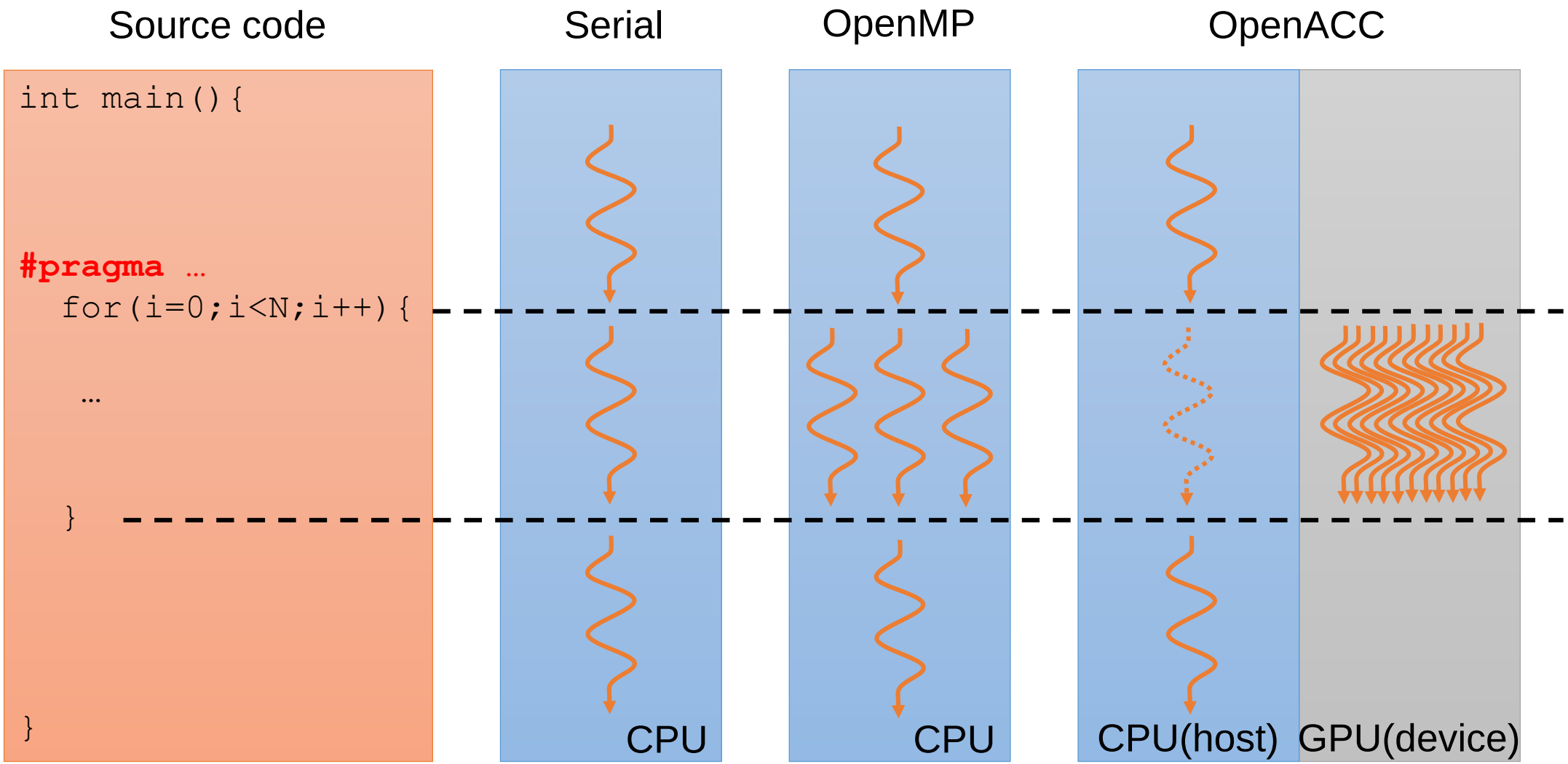
C/C++

```
#pragma acc kernels
for(i=0;i<N;i++){
 …
}
```

```
!$acc kernels
do i=1,N
 …
enddo
!$acc end kernels
```

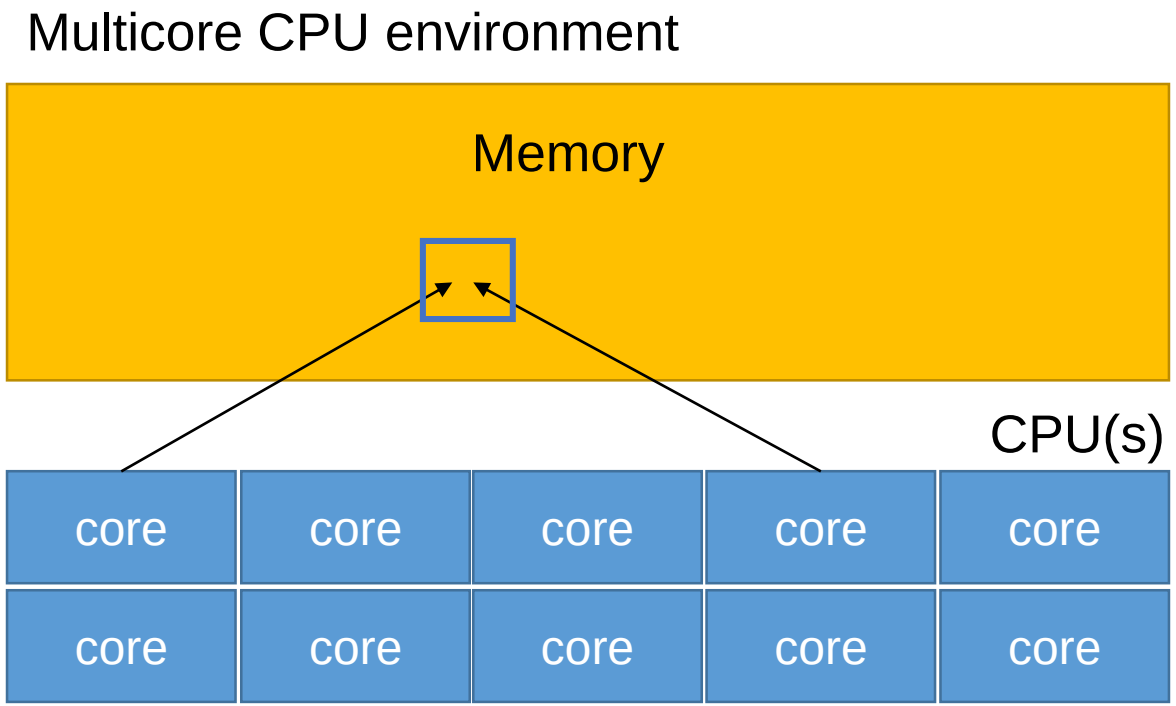
Fortran

Execution image (OpenMP and OpenACC)



Comparison of OpenACC and OpenMP（1/2）

The architecture supposed in OpenMP

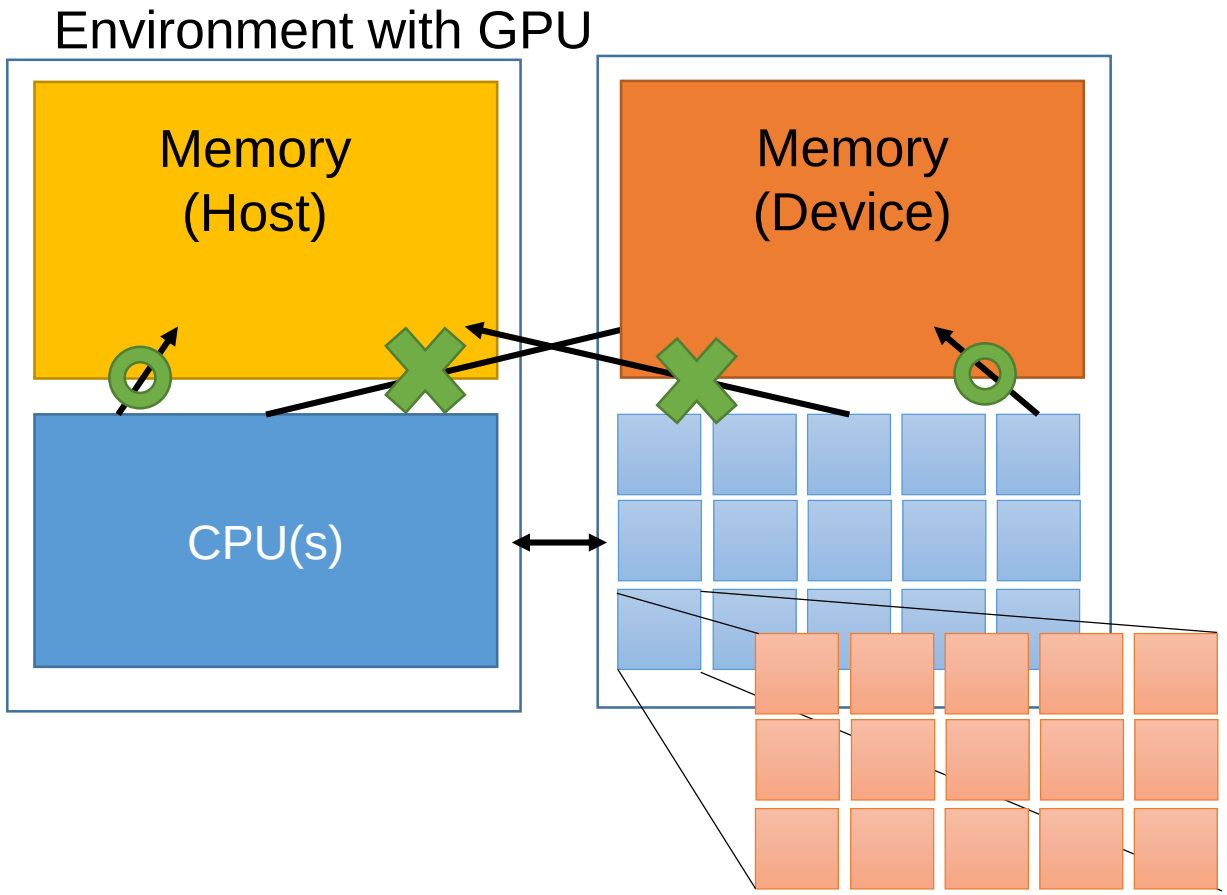


- The number of cores is less than 100（except for Intel Xeon Phi）
- Shared Memory

The biggest difference is the complexity of target architecture.

Comparison of OpenACC and OpenMP（2/2）

The architecture supposed in OpenACC

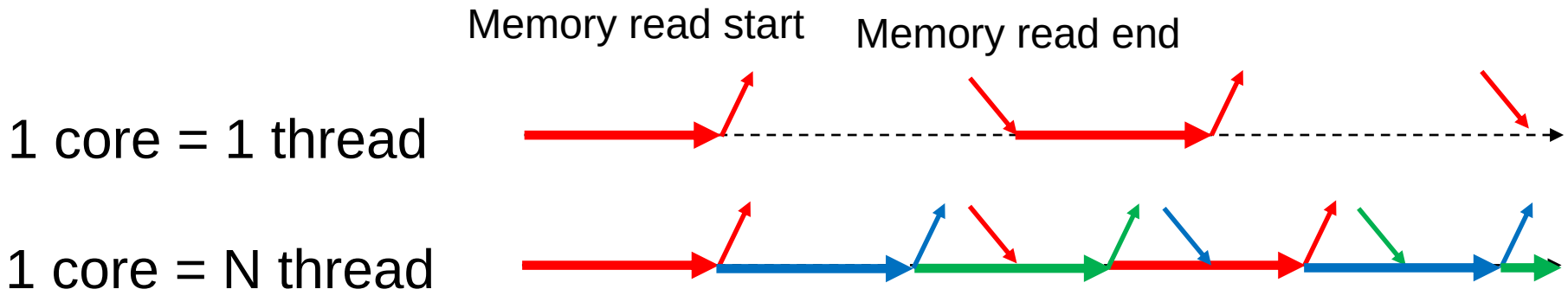


- The number of cores is more than 1000 and there is hierarchical structure.
- Individual memories between host and device
  - ✓ Data transfer between host and device is slow

The biggest difference is the complexity of target architecture.

The Number of Threads and Cores

- Recommended number of threads
  - ✓ CPU: the number of threads = the number of cores（several 10 threads）
  - ✓ GPU: the number of threads >> the number of cores（several ten thousand~several million threads）
    - Optimal value depends on the balance between other resources.
- Low memory latency by the fast context switch
  - ✓ CPU : Evacuation of register/stack is done by OS (late)
  - ✓ GPU : Overhead is almost zero by hardware support
    - Other thread is executed at free time (stall) due to memory access



Main Directives for OpenACC

- Compute Constructs
  - ✓ `kernels, parallel`
- Data Environment
  - ✓ `data, enter data, exit data, update`
- Loop Constructs
  - ✓ `loop`
- Others

Compute Constructs: kernels or parallel

- The `kernels` construct gives the compiler maximum leeway to parallelize and optimize the code how it sees fit for the target accelerator, but also relies most heavily on the compiler's ability to automatically parallelize the code.

- In contrast, the `parallel` construct tells the compiler that everything in the scope of the following region is a single parallel operation that will run in each thread. By itself a parallel region is of limited use, but when paired with the `loop` directive, the compiler will generate a parallel version of the loop for the accelerator. These two directives can, and most often are, combined into a single `parallel loop` directive.

Compute Constructs: kernels or parallel

kernels

```
program main

!$acc kernels
    do i = 1, N
        ! loop body
    end do
!$acc end kernels


end program
```

parallel

```
program main

!$acc parallel num_gangs(N)
!$acc loop gang
    do i = 1, N
        ! loop body
    end do
!$acc end parallel

end program
```

Compute Constructs: kernels or parallel

kernels

Host | Device

```
program main

!$acc kernels
    do i = 1, N
        ! loop body
    end do
!$acc end kernels

end program
```

(relatively automatic) set the number of threads suitable for the target device

parallel

```
program main

!$acc parallel num_gangs(N)
!$acc loop gang
    do i = 1, N
        ! loop body
    end do
!$acc end parallel

end program
```

(relatively manual) set the number of threads to “N”

Clauses for kernels/parallel

kernels

- async
- wait
- device_type
- if
- default(none)
- copy…

parallel

- async
- wait
- device_type
- if
- default(none)
- copy…
- num_gangs
- num_workers
- vector_length
- reduction
- private
- firstprivate

Clauses for kernels/parallel

parallel

Asynchronous execution

- async
- wait

Parameter settings in each device

- device_type
- if
- default(none)

Data handling

- copy…

In the parallel construct, there are clauses for specifying the number of threads, variable types, … and so on.

- num_gangs
- num_workers
- vector_length
- reduction
- private
- firstprivate

Execution Image: kernels

Fortran

```
subroutine copy(dis, src)
  real(4),dimension(:)::dis,src

!$acc kernels copy(src,dis)
  do i=1,N
     dis(i)=src(i)
  enddo
!$acc end kernels

end subroutine copy
```

C

```
void copy(float *dis, float *src){
  int i;

#pragma acc kernels copy(src[0:N] ¥
     dis[0:N])
  for(i=0;i<N;i++){
     dis[i]=src[i];
  }

}
```

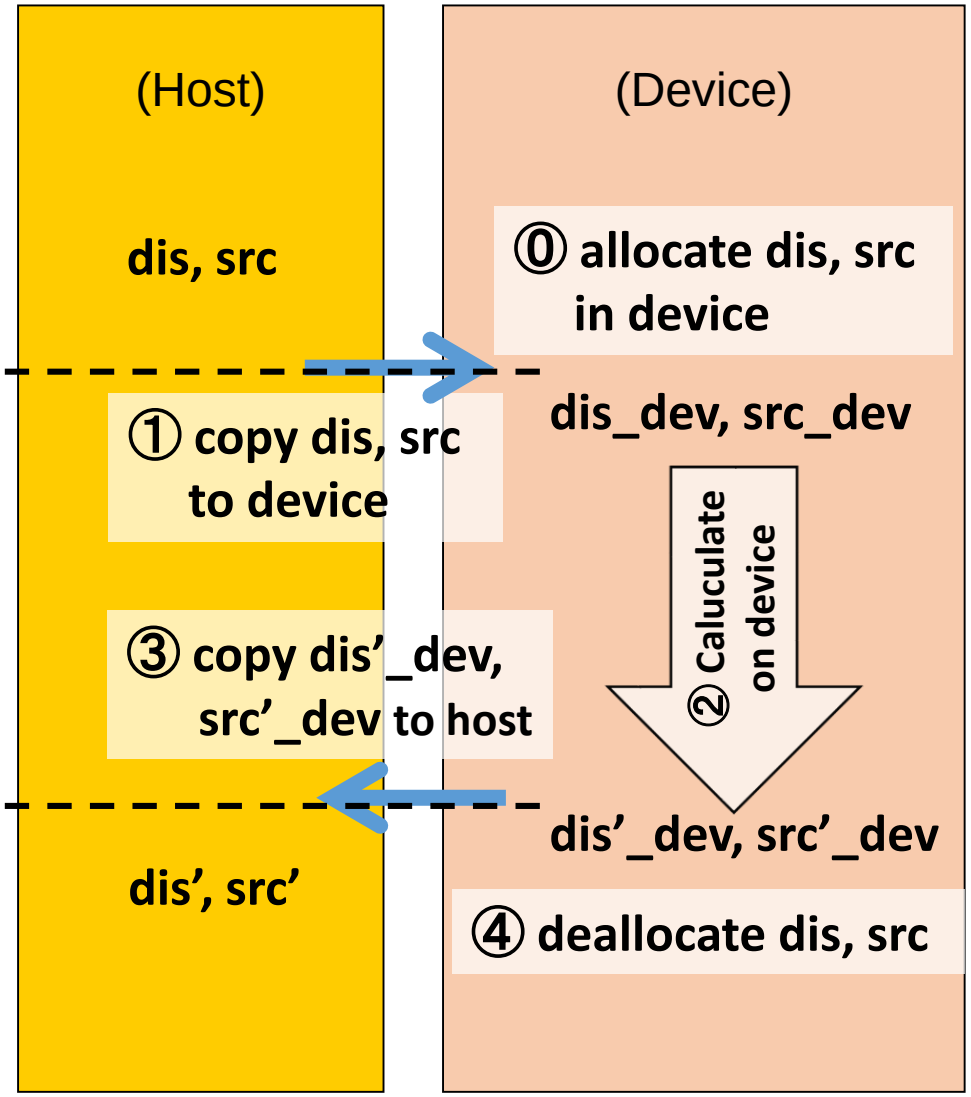
Execution Image: kernels

Fortran

```
subroutine copy(dis, src)
  real(4),dimension(:)::dis,src

!$acc kernels copy(src,dis)
  do i=1,N
     dis(i)=src(i)
  enddo
!$acc end kernels

end subroutine copy
```



Main Directives for OpenACC

- Compute Constructs
  - ✓ `kernels, parallel`
- Data Environment
  - ✓ `data, enter data, exit data, update`
- Loop Constructs
  - ✓ `loop`
- Others

Data on Device

- The OpenACC compilier tries to copy the data required for computation at the `kernels/parallel` construct automatically.
  - ✓ There are cases where data copy doesn't work. The programmer should write data constructs or data clauses.
  - ✓ Automatic data copy can be controlled by `default(none)` clause.

- In the default settings, scaler variables are treated as "firstprivate". Arrays are assigned in the device and are treated as "shared".

Data Clause (copy) in kernels Directive

Fortran

```
subroutine copy(dis, src)
  real(4),dimension(:)::dis,src
  do j=1,M
!$acc kernels copy(src,dis)
     do i=1,N
        dis(i)=dis(i)+src(i)
     enddo
!$acc end kernels
  end do
end subroutine copy
```

C

```
void copy(float *dis,float *src){
  int i,j;
  for(j=0;j<M;j++){
#pragma acc kernels copy(src[0:N] ¥
     dis[0:N])
     for(i = 0;i < N;i++){
        dis[i] = dis[i] + src[i];
     }
  }
}
```

Kernels construct is in a for/do loop…

Data transfer is done every `j` step. It's inefficient.

Data Directives

Fortran

```
subroutine copy(dis, src)
  real(4),dimension(:)::dis,src

!$acc data copy(src,dis)
  do j=1,M
!$acc kernels present(src,dis)
     do i=1,N
        dis(i)=dis(i)+src(i)
     enddo
!$acc end kern
  end do
!$acc end data
end subroutine copy
```

present: already exist on device

C

```
void copy(float *dis,float *src){
  int i,j;
#pragma acc data copy(src[0:N] ¥
   dis[0:N]){
  for(j=0;j<M;j++){
#pragma acc kernels present(src,dis)
     for(i = 0;i < N;i++){
        dis[i] = dis[i] + src[i];
     }

  }
}
}
```

Data transfer can be done at one time by using `data` construct.

Structured data region

enter data, exit data Directives

```
int main(){
  double *q;
  int step;
  for(step=0;step<N;step++){
     if(step==0) init(q);
     solverA(q);
     solverB(q);
      ….
     if(step==N-1) fin(q);
  }
}
```

```
void init(double *q){
  q=(double *)malloc(sizeof(double)*M);
  q= … ; // initialize
#pragma acc enter data copyin(q[0:M])
}
```

```
void fin(double *q){
#pragma acc exit data copyout(q[0:M])
  print(q); // output
  free(q);
}
```

Unstructured data region

Clauses for data, enter/exit data

data

- if
- copy
- copyin
- copyout
- create
- present
- present_or_...
- deviceptr

enter data

- if
- async
- wait
- copyin
- create
- present_or_...

exit data

- if
- async
- wait
- copyout
- delete

Clauses for data, enter/exit data

- `copy`

  allocate, memcpy (H→D), memcpy (D→H), deallocate
- `copyin`

  allocate, memcpy (H→D), deallocate (No data output)
- `copyout`

  allocate, memcpy (D→H), deallocate (No data input)
- `create`

  allocate, deallocate (No data copy)
- `present`

  Do nothing. Tell that data is already exist on device.
- `present_or_copy/copyin/copyout/create` (abbreviation: `pcopy`)

  If data is not on device, `copy/copyin/copyout/create` is done.

After OpenACC2.5, the behavior of `copy,copyin,copyout` is same as that of `pcopy,pcopyin,pcopyout`

Partial Data Transfer

- Capability for partial data transfers on all data constructs and clauses
- Example for the data transfer of 2D array A

Fortran

```
!$acc data copy(A(lower1:upper1,lower2:upper2))
   …
!$acc end data
```

In Fortran, the range by listing the starting and ending index are specified.

C

```
#pragma acc data copy(A[start1:length1][start2:length2])
   …
#pragma acc end data
```

In C, the start index and after the colon the number of elements are specified.

update Directives

- `update` directive can be used if data is already on the device.
- Function of Memcpy (H ⇄ D) only

```
!$acc data copy(A(:,:))
do step=1,N
  …
!$acc update host(A(1:2,:))
  call comm_boundary(A)
!$acc update device(A(1:2,:))
  …
enddo
!$acc end data
```

update

- if
- async
- wait
- device_type
- self same as # host
- host # H ← D
- device # H → D

Main Directives for OpenACC

- Compute Constructs
  - ✓ `kernels, parallel`
- Data Environment
  - ✓ `data, enter data, exit data, update`
- Loop Constructs
  - ✓ `loop`
- Others

Clauses for loop Directive

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

Clauses for loop Directive

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

Merge the triple loops

```
!$acc kernels
!$acc loop collapse(3) gang vector
do k=1,10
   do j=1,10
      do i=1,10

        ….

      enddo
   enddo
enddo
!$acc end kernels
```

Effective for loops having too short loop length for parallelization

Clauses for loop Directive

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

```
!$acc kernels
!$acc loop gang(N)
do k=1,N
!$acc loop worker(1)
   do j=1,N
!$acc loop vector(128)
      do i=1,N

         ….

!$acc kernels
!$acc loop gang vector(128)
do i=1,N

   ….
```

`worker` should be inside of `gang`

`vector` should be inside of `worker`

Multiple clauses can be added.

It is difficult to specify the number of gang, worker and vector.
Therefore it depends on the compiler at first.

Hierarchical Parallelized Model and Loop Directives

- Threads are controlled hierarchically in OpenACC
  - ✓ 3 levels of parallelism (gang, worker, vector)
  - ✓ Thread ID is unkown in OpenACC
- `loop` directives
  - ✓ Specify the treatment of for/do loop in `parallel/kernels`
    - Parameters are set automatically to a certain degree.
  - ✓ Granularity (gang, worker, vector)
  - ✓ Specify the presence or absence of the loop carried dependence

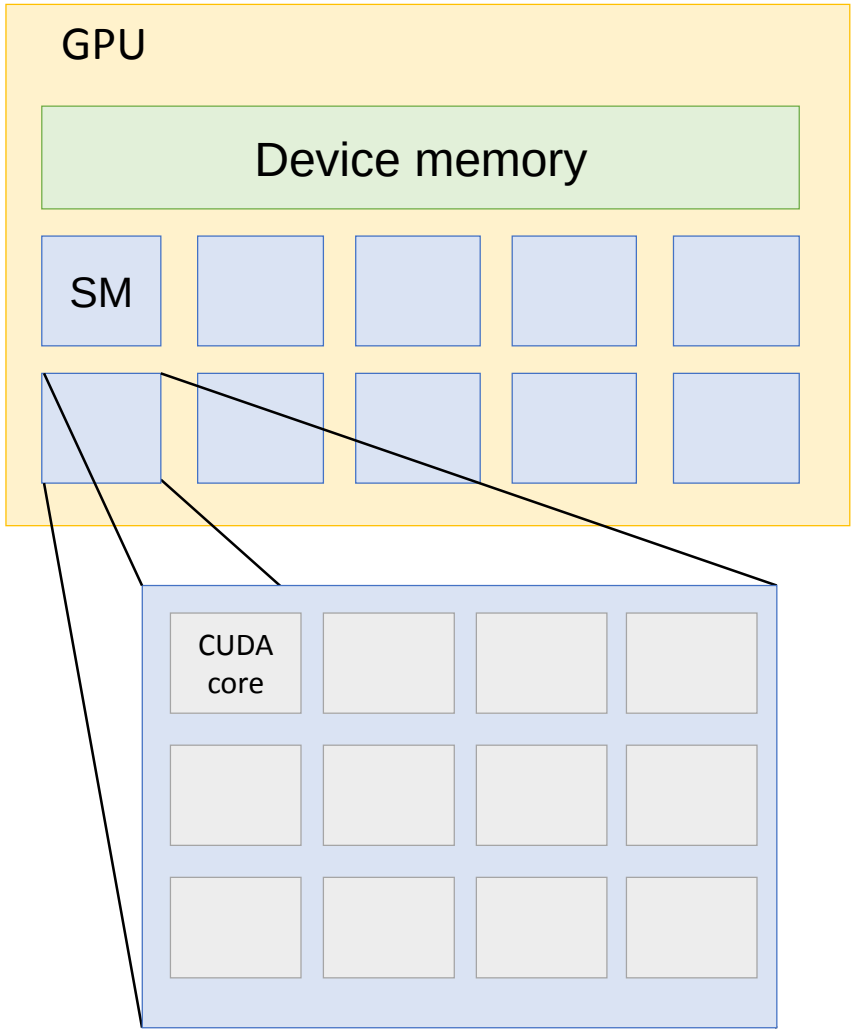
Example of Matrix-Matrix Multiplication on GPU

```
!$acc kernels
!$acc loop gang
do j=1,n
!$acc loop vector
   do i=1,n
      cc=0
!$acc loop seq
      do k=1,n
         cc=cc+a(i,k)*b(k,j)
      enddo
      c(i,j)=cc
   enddo
end do
!$acc end kernels
```

Hierarchical Parallelization Model and Architecture

- OpenMP is 1 level
  - ✓ Multicore CPU is also 1 level
  - ✓ Recently, there is 2nd level (SIMD)
- CUDA is 2 level (block and thread)
  - ✓ NVIDA GPU is also 2 level
    - Multiple CUDA cores are installed in 1SM
    - Each core shares resources of SM
- OpenACC is 3 level
  - ✓ Corresponding to various device
  - ✓ In NVIDIA GPU, 2 level (gang and vector) only are often specified.

- Configulation of NVIDIA GPU



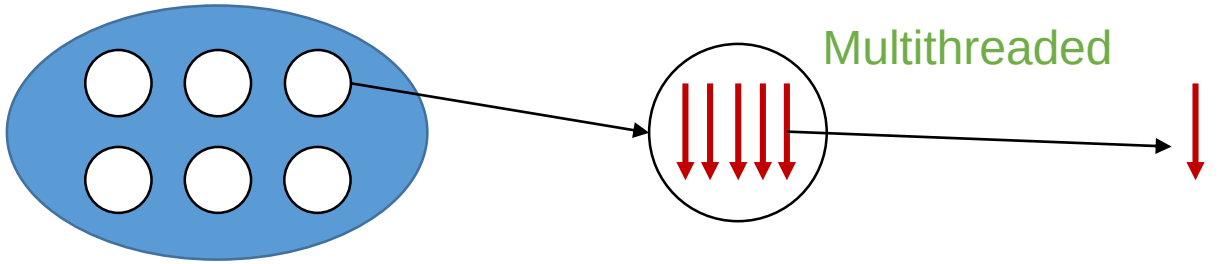
3 Level Hierarchical Parallelization Model in OpenACC

gang … Gang is an unit realizing parallel execution at the level of large task (parallelization of coarse granularity) and has no synchronization mechanism on hardware and is executed independently.

worker … Worker is an unit having synchronization mechanism and is executed at the parallelization of fine granularity.

vector … Vector is an unit executing in worker for SIMD or vector processing.

Execution Unit



NVIDIA GPU case

gang → **S**treaming **M**ultiprocessors

worker → Multithreading → **Warps** within **SM**

vector → Vector length→ Threads in a warp = 32

In order to obtain the good combination of 3 level and high performance, the number of gang/worker/vector should be changed by trial and error.

Clauses for loop Directive

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

```
do j=1,N
   do i=1,N
      idxI(i)=i; idxJ(j)=j
   enddo
enddo
!$acc kernels &
!$acc& copyin(A, idxI, idxJ) copyout(B)
!$acc loop independent gang
do j=1,N
!$acc loop independent vector(128)
   do i=1,N
      B(idxI(i),idxJ(j))=alpha*A(i,j)
   enddo
enddo
!$acc end kernels
```

Indirect reference →

An OpenACC compiler is conservative. If it seems there are dependencies of variables, the compiler does not parallelize. The programmers should check compile message whether the parallelization is done or not.

Clauses for loop Directive

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

```
!$acc kernels &
!$acc loop reduction(+:val)
do i=1,N
   val = val + 1
enddo
!$acc end kernels
```

```
!$acc loop reduction (+:val)
```

operator

Target variable
(scaler only)

C and C++		Fortran	
operator	initialization value	operator	initialization value
+	0	+	0
*	1	*	1
max	least	max	least
min	largest	min	largest
&	~0	iand	all bits on
|	0	ior	0
^	0	ieor	0
&&	1	.and.	.true.
||	0	.or.	.false.
		.eqv.	.true.
		.neqv.	.false.

Execution of Sample Programs for OpenACC on Reedbush-H (with GPU node)

Execution of OpenACC Program (1/2)

1. Move to /lustre directory.
   `$ cd /lustre/gi16/XXXXXX/`
2. Copy OpenACC.tar on /lustre/gi16/c26050 to your own directory.
   `$ cp /lustre/gi16/c26050/OpenACC.tar ./`
3. Extract files from OpenACC.tar
   `$ tar xvf OpenACC.tar`
4. Move to OpenACC/ directory
   `$ cd OpenACC`
   for C users : `$ cd C`
   for Fortran users : `$ cd F`
5. Move to Calc/ directory
   `$ cd Calc`

Execution of OpenACC Program (2/2)

6. Load CUDA environment and PGI compiler
   `$ module load cuda pgi`
7. Compile the source file (execute a file for compile prepared already)
   `$ ./compile`
8. Submit the job by qsub command and job script
   `$ qsub run.sh`
9. Confirm the status of submitted job
   `$ rbstat`
10. After the execution, the following files are generated.
    `run.sh.exxxxxx`
    `run.sh.oxxxxxx` (`xxxxxx` is Job ID)
11. See the standard output file
    `cat run.sh.oxxxxxx`

Check the standard output file

You can see the following result. The results in C and Fortran are slightly different.

```
====== OpenACC parallel/kernels construct test program ======
kernels                 : sum(B) = 1000000.00
kernels restrict        : sum(B) = 1000000.00
kernels independent     : sum(B) = 1000000.00
parallel                : sum(B) = 1000000.00
parallel restrict       : sum(B) = 1000000.00
parallel independent    : sum(B) = 1000000.00
kernels default none    : sum(B) = 1000000.00
parallel default none   : sum(B) = 1000000.00
kernels indirect ref    : sum(B) = 1000000.00
kernels local array     : sum(B) = 1000000.00
parallel local array    : sum(B) = 1000000.00
```

Checking the compile message

- In OpenACC programming, the confirmation of compile messages is very important.
  - ✓ Parallelization by OpenACC is conservative, so a program code that can be parallelized may not be parallelized.
  - ✓ There are many loops for parallelization.
  - ✓ In order to know the granularity (gang, worker, vector) and the number of threads in each parallelized loop.

- How to output the compile message
  - ✓ Add `-Minfo=accel` to compile option.

Check the Compile Message (Fortran)

Source code →

```
07 subroutine acc_kernels()
08   double precision::A(N,N),B(N,N)
09   double precision::alpha=1.0d0
10   integer::i,j
11   A(:,:)=1.0d0
12   B(:,:)=0.0d0
13   !$acc kernels
14   do j=1,N
15      do i=1,N
16         B(i,j)=alpha*A(i,j)
17      enddo
18   enddo
19   !$acc end kernels
20 end subroutine acc_kernels
```

Subroutine name

↓ compile message

```
pgfortran -O3 -acc -Minfo=accel -ta=tesla,cc60 -Mcuda acc_compute.f90
acc_kernels:
     13, Generating implicit copyin(a(:,:))
         Generating implicit copyout(b(:,:))
     14, Loop is parallelizable
     15, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         14, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         15, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

Array A is transferred as copyin, and array B is transferred as copyout.

The double loop at line 15 and 16 is treated as the block construction of (32x4) threads.

Check the Compile Message (C)

Source code →

```
41 void acc_kernels(double *A, double *B){
42  double alpha=1.0;
43  int i,j;
      /* Init of A and B */
50 #pragma acc kernels
51   for(j=0;j<N;j++){
52     for(i=0;i<N;i++){
53       B[i+j*N]=alpha*A[i+j*N];
54     }
55   }
56 }
```

↓ Compile message

```
pgcc -O3 -acc -Minfo=accel -ta=tesla,cc60 -Mcuda acc_compute.c
acc_kernels:
     50, Generating implicit copy(B[:1000000])
         Generating implicit copyin(A[:1000000])
     51, Loop carried dependence of B-> prevents parallelization
         Loop carried backward dependence of B-> prevents vectorization
         Complex loop carried dependence of B->,A-> prevents parallelization
         Accelerator scalar kernel generated
         Accelerator kernel generated
         Generating Tesla code
         51, #pragma acc loop seq
         52, #pragma acc loop seq
     52, Complex loop carried dependence of B->,A-> prevents parallelization
```

Array A is transferred as copyin, and array B is transferred as copy.

Parallelization is prevented because pointer A and B may point to same region.

Check the Compile Message (C)

Source code →

```
59 void acc_kernels_restrict(double
*restrict A, double *restrict B){
60  double alpha=1.0;
61  int i,j;
      /* Init of A and B */
68 #pragma acc kernels
69   for(j=0;j<N;j++){
70     for(i=0;i<N;i++){
71       B[i+j*N]=alpha*A[i+j*N];
72     }
73   }
74 }
```

↓ Compile message

```
acc_kernels_restrict:
     68, Generating implicit copy(B[:1000000])
         Generating implicit copyin(A[:1000000])
     69, Loop carried dependence of B-> prevents parallelization
         Loop carried backward dependence of B-> prevents vectorization
     70, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         69, #pragma acc loop seq
         70, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Parallelization is still prevented only on line.69.

Check the Compile Message (C)

Source code →

```
77 void acc_kernels_independent(double
*restrict A, double *restrict B){
78   double alpha=1.0;
79   int i,j;
       /* Init of A and B */
86 #pragma acc kernels
87 #pragma acc loop independent
88   for(j=0;j<N;j++){
89 #pragma acc loop independent
90     for(i=0;i<N;i++){
91       B[i+j*N]=alpha*A[i+j*N];
92     }
93   }
94 }
```

↓ Compile message

```
acc_kernels_independent:
     86, Generating implicit copy(B[:1000000])
         Generating implicit copyin(A[:1000000])
     88, Loop is parallelizable
     90, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         88, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         90, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

Parallelization is done completely.

Chech the OpenACC program by PGI_ACC_TIME

- If you use PGI environment, you can check the execution of OpenACC program by using the environment variable “`PGI_ACC_TIME`”.
- How to use (in general Linux environment or interactive job)

```
$ export PGI_ACC_TIME=1
$ （execution of a program）
```

- In Reedbush, above sentence is written in a job script file.

Chech the OpenACC Program by PGI_ACC_TIME

```
$ qsub run.sh
```

After the execution, you can get two files as follows.

`run.sh.eXXXXX` (standard error)

`run.sh.oXXXXX` (standard output)

```
$ cat run.sh.eXXXXX
```

```
41 void acc_kernels(double *A, double *B){
42  double alpha=1.0;
43  int i,j;
      /* Init of A and B */
50 #pragma acc kernels
51   for(j=0;j<N;j++){
52     for(i=0;i<N;i++){
53       B[i+j*N]=alpha*A[i+j*N];
54     }
55   }
56 }
```

```
Accelerator Kernel Timing data
/lustre/pz0108/z30108/OpenACC_samples/C/acc_compute.c
  acc_kernels  NVIDIA  devicenum=0
    time(us): 149,101
    50: compute region reached 1 time
        51: kernel launched 1 time
            grid: [1]  block: [1]
             device time(us): total=140,552 max=140,552 min=140,552 avg=140,552
            elapsed time(us): total=140,611 max=140,611 min=140,611 avg=140,611
    50: data region reached 2 times
        50: data copyin transfers: 2
             device time(us): total=3,742 max=3,052 min=690 avg=1,871
        56: data copyout transfers: 1
             device time(us): total=4,807 max=4,807 min=4,807 avg=4,807
```

←The number of threads

↓kernel execution time

↓ the number and time of data-transfer

Practice 1 : compile and execute programs

- Compare the source file "`acc_compute.f90`" or "`acc_compute.c`" and the compile message.
- After the execution, check the results in the standard error in each function/subroutine.

- Focus point
  - ✓ Difference of `parallel` and `kernels`
  - ✓ Compile message when there is an indirect reference
    - Comparison of "`acc_kernels_BAD_indirect_reference`" and "`acc_kernels_indirect_reference`"

Other Programs for OpenACC

Practice 2: time comparison of data transfer

- Move to `Data/` directory
- Compare the source file "`acc_data.f90`" or "`acc_data.c`" and the compile message.
- Execute the program and check the output of `PGI_ACC_TIME`
  - ✓ After the execution of batch job, two output files `run.sh.oXXXXXX`(standard output), `run.sh.eXXXXXX` (standard error) are generated.
  - ✓ Chech the output of `PGI_ACC_TIME`
    ```
    $ cat run.sh.eXXXXXX
    ```
  - ✓ Please focus on the execution time of `acc_data_copy` and `acc_data_copyinout`

Practice 3: Matrix-Matrix Multiplication

- Move to `Matmul/` directory
- In `matmul.f90` or `matmul.c`,
  - ✓ Add OpenACC directives to `acc_matmul` routine.
  - ✓ Attention: check the compile message
    - An example of error(C):

      `Accelerator restriction: size of the GPU copy of C,B is unknown`

Example of Output

```
====== OpenACC matmul program ======
1024 * 1024 matrix
check result...OK
elapsed time[sec] :    1.44754
FLOPS[GFlops]     :    1.48210
```

Practice 4: 2D Heat Equation

- Move to `Diff2D/acc/` directory
- In `diff2D_acc.f90` or `diff2D_acc.c`,
  - ✓ Accelerate the source code by OpenACC directives although some directives are already inserted.
  - ✓ OpenMP version is also prepared as the comparison (`Diff2D/omp/`)

An example of output

```
====== OpenACC diffusion program ======
1024*1024
Elapsed time[s]: 574.247
```

← Very slow

- Work procedure
  1. Rewrite and compile
  2. Check the compile message
  3. Check the output and result of PGI_ACC_TIME

Discretization of 2D Heat Equation by Finite Difference Method

$$C\rho\frac{\partial T}{\partial t}=\lambda\frac{\partial^2 T}{\partial x^2}+\lambda\frac{\partial^2 T}{\partial y^2}+\lambda\frac{\partial^2 T}{\partial z^2}\quad\rightarrow\quad\frac{\partial T}{\partial t}=\alpha\left(\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}\right)\quad\rightarrow\quad\boxed{\frac{\partial T}{\partial t}=\alpha\nabla^2 T}$$

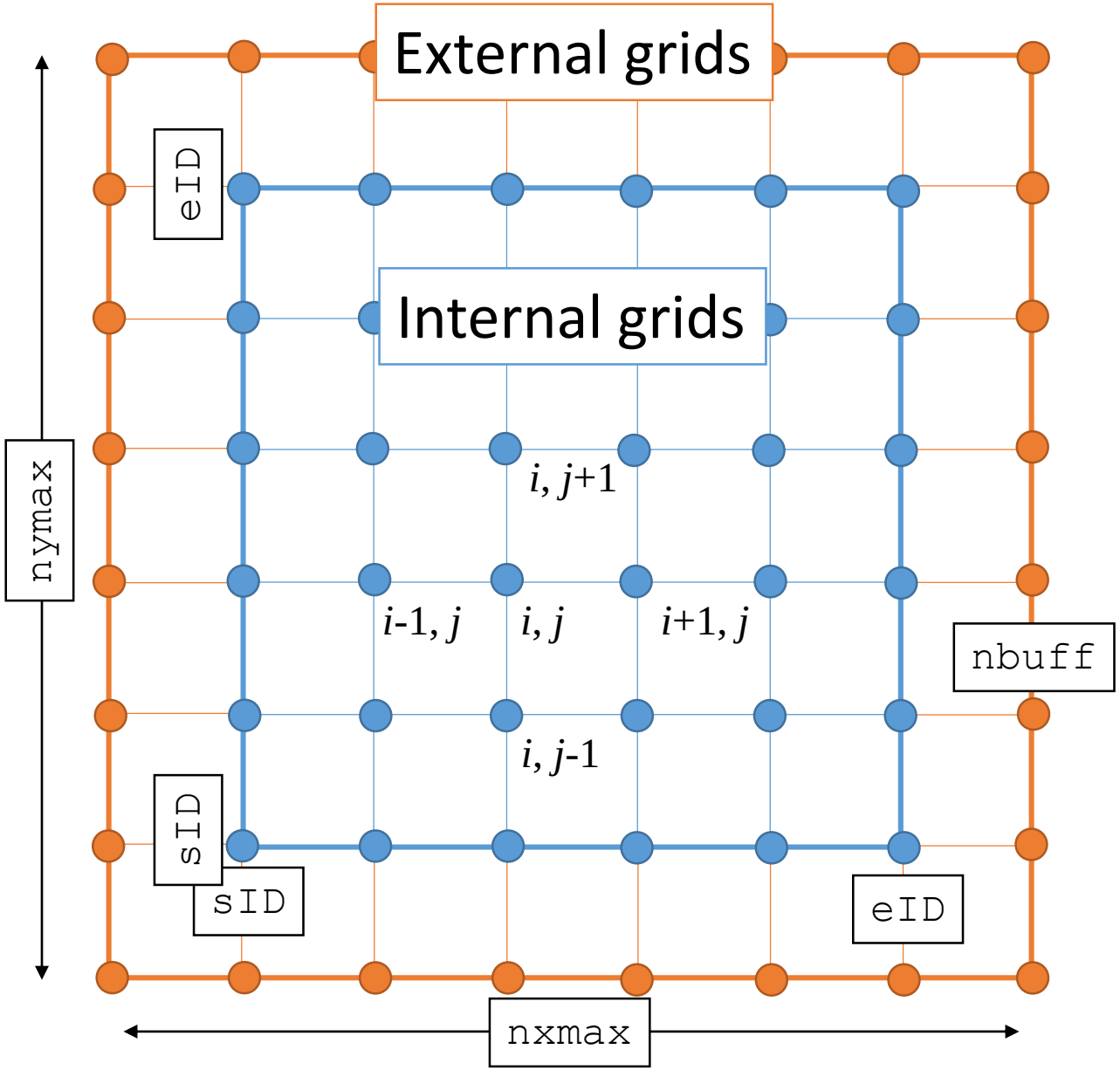
$$\alpha=\frac{\lambda}{C\rho}$$

Time: 1st-order Euler explicit, Space: 2nd-order central difference

$$\frac{\partial T}{\partial t}=\alpha\left(\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}\right)\rightarrow\frac{T_{i,j}^{n+1}-T_{i,j}^{n}}{\Delta t}=\alpha\left(\frac{T_{i+1,j}^{n}-2T_{i,j}^{n}+T_{i-1,j}^{n}}{\Delta x^2}+\frac{T_{i,j+1}^{n}-2T_{i,j}^{n}+T_{i,j-1}^{n}}{\Delta y^2}\right)$$

$$\rightarrow\therefore T_{i,j}^{n+1}=T_{i,j}^{n}+\Delta t\alpha\left(\frac{T_{i+1,j}^{n}-2T_{i,j}^{n}+T_{i-1,j}^{n}}{\Delta x^2}+\frac{T_{i,j+1}^{n}-2T_{i,j}^{n}+T_{i,j-1}^{n}}{\Delta y^2}\right)$$

Grid configuration



External grids: prepared for boundary

`nxmax`: the number of grid points for x
`nymax`: the number of grid points for y
`nbuff`: width of the external grid

`sID`, `eID`: start and end point of internal grids

Fortran:	`sID(1)`,	`eID(1)`	x direction
	`sID(2)`,	`eID(2)`	y direction
C:	`sID[0]`,	`eID[0]`	x direction
	`sID[1]`,	`eID[1]`	y direction

main（diff2D_acc）

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \Delta t \alpha \left(\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2} \right)$$

Calculate the left equation in `kernel`

Comment out for output

Fortran

```
25   do nt=1,ntmax
26      call kernel(nxmax,nymax,sID,eID,dx,dt,alp,T)
27 !     if(mod(nt,5000)==1) then
28 !        nstep=nstep+1
29 !        print *,'iteration and output count:',nt,nstep
30 !        call output(nstep,nxmax,nymax,sID,eID,dx,T)
31 !     endif
32   enddo
```

C

```
38    for(nt=1;nt<=ntmax;nt++){
39      kernel(nxmax,nymax,sID,eID,dx,dt,alp,T);
40 //     if(nt%5000==1){
41 //       nstep=nstep+1;
42 //       printf("iteration and output count: %d %d¥n",nt,nstep);
43 //       output(nstep,nxmax,nymax,sID,eID,dx,T);
44 //     }
45    }
```

kernel(diff2D_acc.f90)

```
! -- calc. next timestep --
!$acc kernels copyin(T,sID,eID) copyout(Tn)
do ny=sID(2),eID(2)
   do nx=sID(1),eID(1)
      Tn(nx,ny)=T(nx,ny)+rx*(T(nx+1,ny  )-2.0d0*T(nx,ny)+T(nx-1,ny  )) &
                         +ry*(T(nx  ,ny+1)-2.0d0*T(nx,ny)+T(nx  ,ny-1))
   enddo
enddo
!$acc end kernels
! -- update --
!$acc kernels copyin(Tn,sID,eID) copyout(T)
do ny=sID(2),eID(2)
   do nx=sID(1),eID(1)
      T(nx,ny)=Tn(nx,ny)
   enddo
enddo
!$acc end kernels
```

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \Delta t \alpha \left(\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2} \right)$$

It takes long time to execute `diff2D_acc` code although OpenACC directives are inserted already. This is because data-copy is frequently done in the `kernels` directive in each time step.

kernel(diff2D_acc.c)

```
70   // calc, next timestep
71 #pragma acc kernels copyin(T[0:nxmax][0:nymax],sID[0:2],eID[0:2])        ¥
72   copyout(Tn[0:nxmax][0:nymax])
73 #pragma acc loop independent
74   for(nx=sID[0];nx<=eID[0];nx++){
75 #pragma acc loop independent
76     for(ny=sID[1];ny<=eID[1];ny++){
77       Tn[nx][ny]=T[nx][ny]+rx*(T[nx+1][ny  ]-2.0*T[nx][ny]+T[nx-1][ny  ])
78                           +ry*(T[nx  ][ny+1]-2.0*T[nx][ny]+T[nx  ][ny-1]);
79     }
80   }
81   // update
82 #pragma acc kernels copyin(Tn[0:nxmax][0:nymax],sID[0:2],eID[0:2])        ¥
83   copyout(T[0:nxmax][0:nymax])
84 #pragma acc loop independent
85   for(nx=sID[0];nx<=eID[0];nx++){
86 #pragma acc loop independent
87     for(ny=sID[1];ny<=eID[1];ny++){
88       T[nx][ny]=Tn[nx][ny];
89     }
90   }
```

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \Delta t \alpha \left(\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2} \right)$$

It takes long time to execute `diff2D_acc` code although OpenACC directives are inserted already. This is because data-copy is frequently done in the `kernels` directive in each time step.