OpenACC
Tutorial

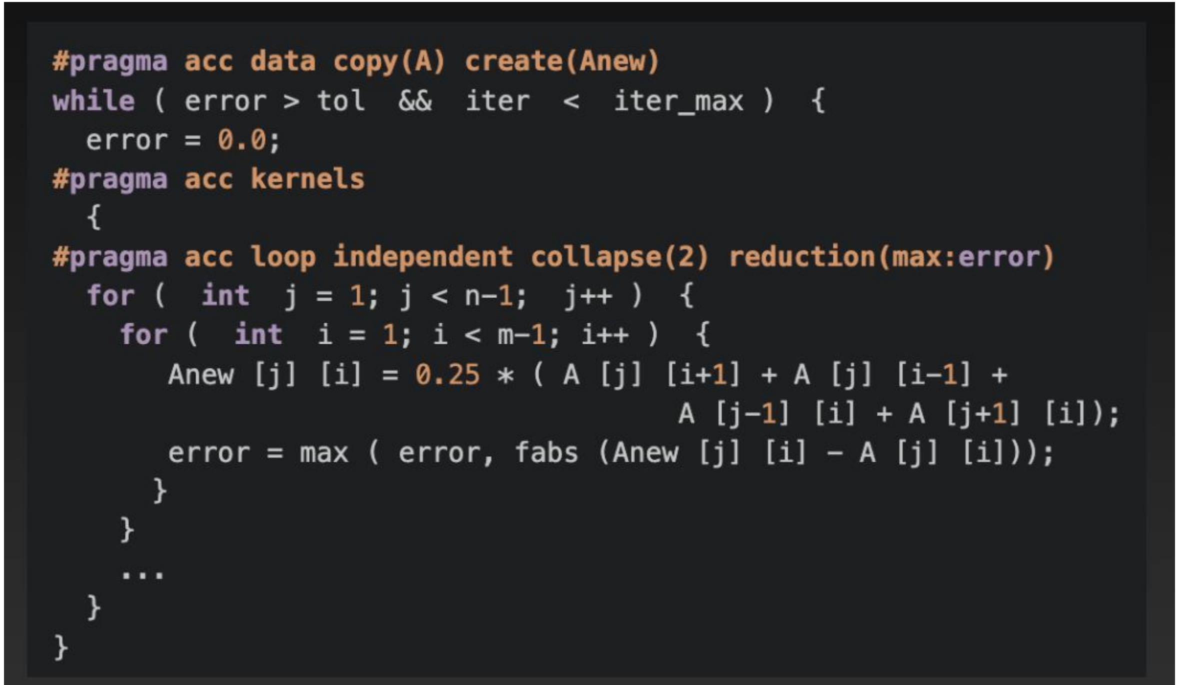
NARLabs 財團法人國家實驗研究院
國家高速網路與計算中心
National Center for High-performance Computing

國立清華大學叢集電腦競賽團隊
Student Cluster Competition Team of NTHU

What is OpenACC

- **Open Acc**elerators
- Through various **compiler directives** to write GPU code
- Lower the technical barriers to GPU programming

What is OpenACC



```
#pragma acc data copy(A) create(Anew)
while ( error > tol  &&  iter  <  iter_max )  {
  error = 0.0;
#pragma acc kernels
  {
#pragma acc loop independent collapse(2) reduction(max:error)
  for (  int  j = 1; j < n-1;  j++ )  {
    for (  int  i = 1; i < m-1; i++ )  {
       Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                                      A [j-1] [i] + A [j+1] [i]);
       error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
      }
    }
    ...
  }
}
```

https://www.openacc.org/

CUDA

- CudaMalloc(...): Declare memory on the GPU
- CudaMemcpy(...): Move data
- functionname<<<thread, blocks>>>(...): Write your own Cuda Kernel Function

=> High entry barrier

OpenACC

- No need to declare the memory on the device
- #pragma acc data copy(...): You can move data with a simple clause
- You can directly use parallel region to port to the GPU.

=> Easy to use

OpenACC Directive

- #pragma acc <directive> <clause>
  - #pragma is a compiler hint
  - acc tells the compiler that this is the OpenACC pragma
  - directive is what OpenACC tells the compiler to indicate
  - clause is an instruction for OpenACC to supplement or optimize the directive.

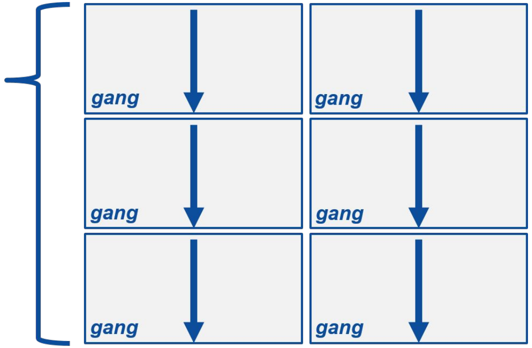
OpenACC Directive

- #pragma acc parallel
  - parallel tells the compiler that this code should be **redundantly parallelized**

```
#pragma acc parallel
{
```

When encountering the ***parallel*** directive, the compiler will generate *1 or more parallel* ***gangs***, which execute redundantly.

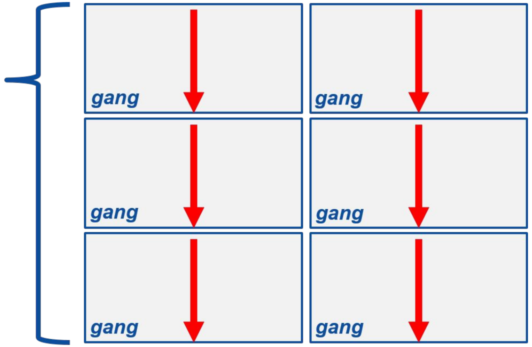
```
}
```



```
#pragma acc parallel
{

    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

}
```

This loop will be ***redundantly parallelized*** across the ***gangs***

This means that each ***gang*** will execute the entire loop

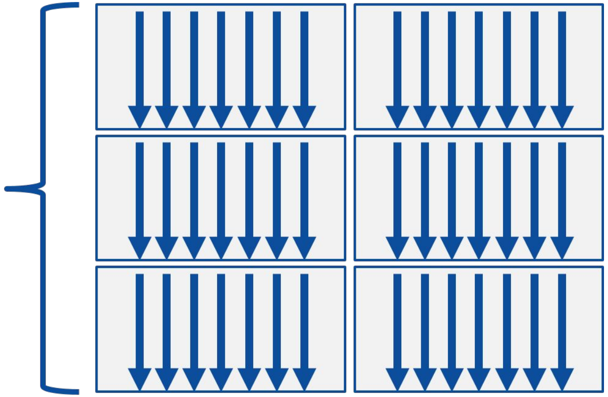


OpenACC Directive

- #pragma acc parallel loop
  - loop tells the compiler that this loop needs to be parallelized
  - It also tells the compiler that this loop can be safely parallelized.

```
#pragma acc parallel
{

    #pragma acc loop
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

}
```

The *loop* directive informs the compiler which loops to parallelize.



The iterations of the loop will be broken up evenly among the parallel *gangs*.

The *gangs* will then execute in parallel with one another.

OpenACC Directive

- #pragma acc parallel loop reduction(<operation>:<target>)
    - reduction tells the compiler that a target is to be reduced
    - reduce: perform global operations on the selected target

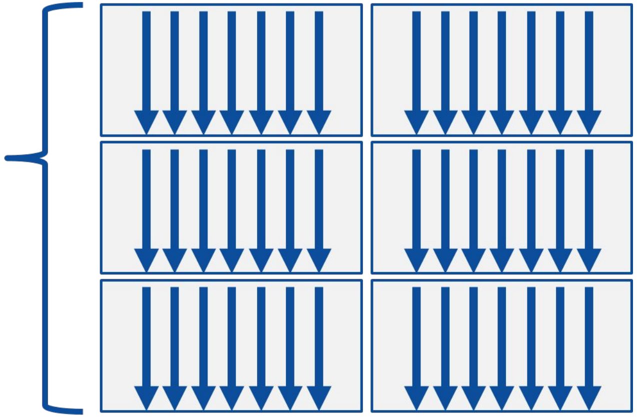
```
int sum = 0;

#pragma acc parallel loop reduction(+:sum)

for(int i = 0, i < N, i++) sum += i;
```

OpenACC Directive

- #pragma acc kernels
  - All actions are decided by the compiler
  - You can also include the sequential code

```
#pragma acc kernels
{
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

    for(int i = 0; i < M; i++)
    {
        // Do Something Else
    }
}
```

With the ***kernels*** directive, the ***loop*** directive is implied.

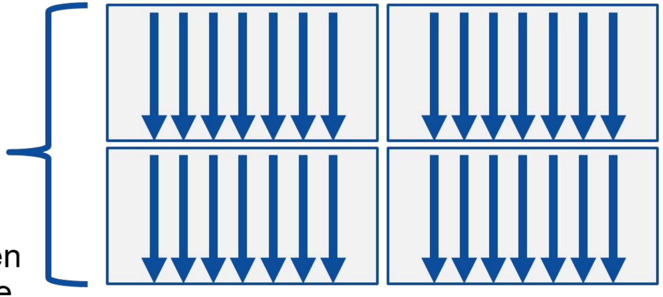


OpenACC Directive

- #pragma acc kernels loop independent
    - Tell the compiler that this loop can be safely parallelized, and force parallelization of it

```
#pragma acc kernels
{

    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

    for(int i = 0; i < M; i++)
    {
        // Do Something Else
    }

}
```

Each loop can have a different number of gangs, and those gangs can be organized/optimized completely differently.



This process can happen multiple times within the ***kernels*** region.

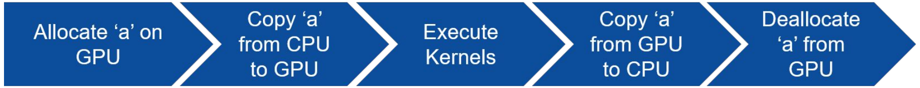
Data Management

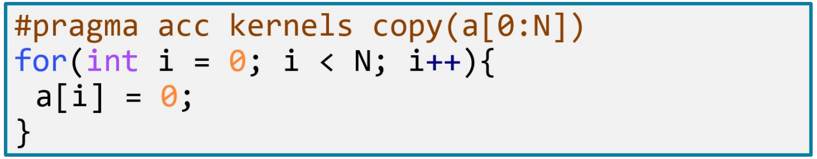
You can copy only part of the data
#pragma acc parallel loop copy(A[1:N-2])

- #pragma acc data copy(...)
  - Copy the data into the GPU and copy the data back to the CPU after the parallel region ends
- #pragma acc data copyin(...)
  - Copy the data into the GPU and delete the data on the GPU after the parallel region ends.
- #pragma acc data copyout(...)
  - Copy the data back to the CPU and delete the data on the GPU after the parallel region ends.
- #pragma acc data create(...)
  - Declare a space on the GPU without performing any copying operations
  - When there are variables for temporary storage, using this clause eliminates the need to copy in and out.

Data Management

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

#pragma acc parallel

{

        #pragma acc loop

        for(int i = 0; i < N; i++) A[i] = 0;

}
```





```
#pragma acc kernels copy(a[0:N])
for(int i = 0; i < N; i++){
 a[i] = 0;
}
```

Loop Optimization

- #pragma acc parallel loop collapse(...)
  - Can be used in tightly nested loops
  - collapse can flatten loops and turn multiple loops into one large parallel loop

```
#pragma acc parallel loop collapse( 2 )
for(int j = 0; j < M; j++) {
    for(int k = 0; k < Q; k++) {
        < loop code >
    }
}
```

TIP1:
When the outer loop is too small, flattening the loop can increase GPU usage.

Loop Optimization

- #pragma acc parallel loop tile(x, y)
  - Calculate loop break for multiple tiles (blocks)

```
#pragma acc parallel loop tile( 32, 32 )

for(int j = 0; j < 128; j++) {

    for(int k = 0; k < 128; k++) {

        < loop code >

    }

}
```

TIP1:
Try to make the tile size a multiple of 32. The threads in a worker and vector of Nvidia GPU are executed in units of 32.

TIP2:
Do not use tiles larger than 32*32, because in NVIDIA GPU, the maximum number of threads in a gang is 1024 (32*32)