

# Module II: OPENACC DIRECTIVES



DEEP  
LEARNING  
INSTITUTE

## MODULE OVERVIEW

### OpenACC Directives

- The parallel directive
- The kernels directive
- The loop directive
- Fundamental differences between the kernels and parallel directive
- Expressing parallelism in OpenACC

# OPENACC SYNTAX

## OPENACC SYNTAX

Syntax for using OpenACC directives in code

C/C++

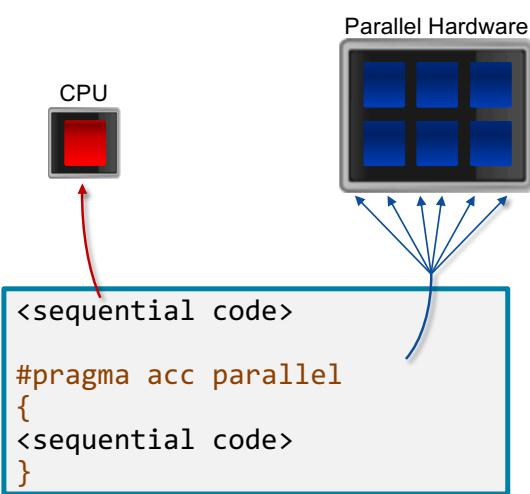
```
#pragma acc directive clauses  
<code>
```

- A **pragma** in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.
- “**acc**” informs the compiler that what will come is an OpenACC directive
- **Directives** are commands in OpenACC for altering our code.
- **Clauses** are specifiers or additions to directives.

# OPENACC PARALLEL DIRECTIVE

## OPENACC PARALLEL DIRECTIVE

### Explicit programming



- The `parallel` directive instructs the compiler to create parallel *gangs* on the accelerator
- Gangs are independent groups of worker threads on the accelerator
- The code contained within a `parallel` directive is executed redundantly by all parallel gangs

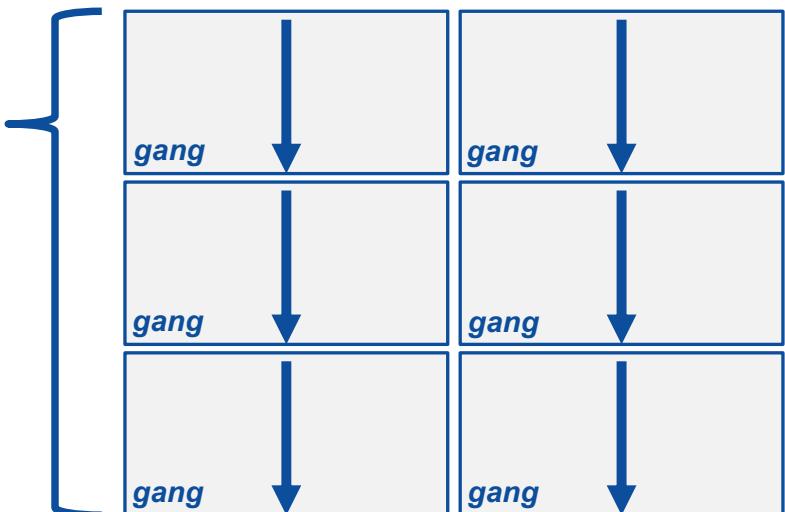
# OPENACC PARALLEL DIRECTIVE

## Expressing parallelism

```
#pragma acc parallel  
{
```

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

```
}
```



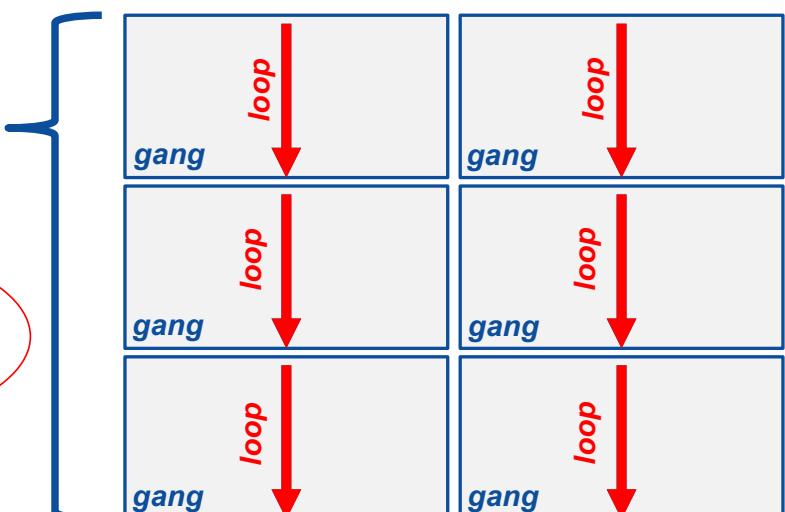
# OPENACC PARALLEL DIRECTIVE

## Expressing parallelism

```
#pragma acc parallel  
{
```

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

This loop will be executed redundantly on each gang



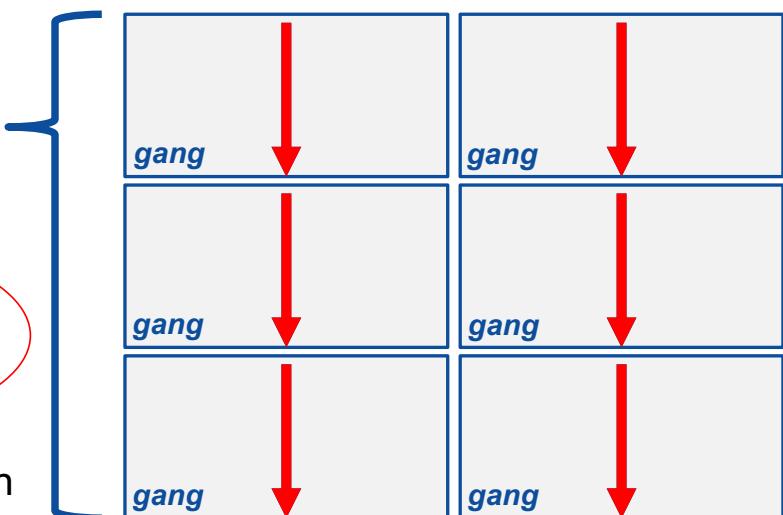
# OPENACC PARALLEL DIRECTIVE

## Expressing parallelism

```
#pragma acc parallel {
```

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

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



OpenACC More Science. Less Programming



NVIDIA

DEEP LEARNING INSTITUTE

# OPENACC PARALLEL DIRECTIVE

## Parallelizing a single loop

C/C++

```
#pragma acc parallel  
{  
    #pragma acc loop  
    for(int i = 0; j < N; i++)  
        a[i] = 0;  
}
```

- Use a **parallel** directive to mark a region of code where you want parallel execution to occur
- This parallel region is marked by curly braces in C/C++ or a start and end directive in Fortran
- The **loop** directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs

OpenACC More Science. Less Programming



NVIDIA

DEEP LEARNING INSTITUTE

# OPENACC PARALLEL DIRECTIVE

## Parallelizing a single loop

C/C++

```
#pragma acc parallel loop
for(int i = 0; j < N; i++)
    a[i] = 0;
```

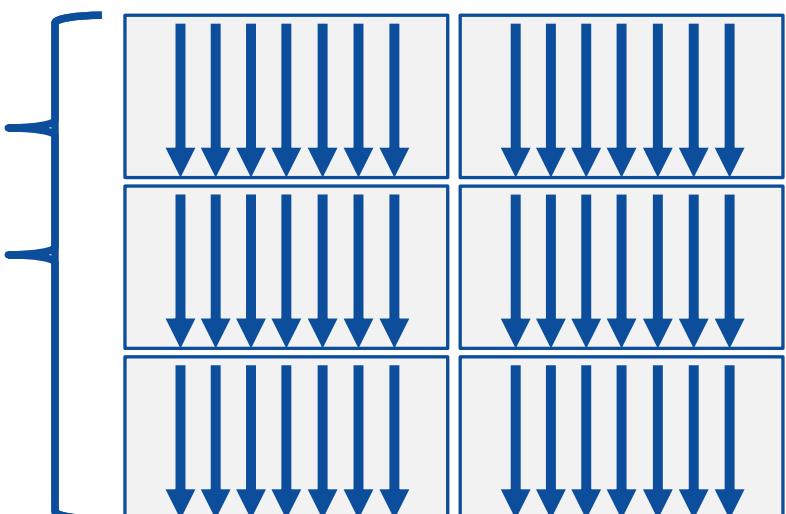
- This pattern is so common that you can do all of this in a single line of code
- In this example, the parallel loop directive applies to the next loop
- This directive both marks the region for parallel execution and distributes the iterations of the loop.
- When applied to a loop with a data dependency, parallel loop may produce incorrect results

# OPENACC PARALLEL DIRECTIVE

## Expressing parallelism

```
#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.



# OPENACC PARALLEL DIRECTIVE

## Parallelizing many loops

```
#pragma acc parallel loop
for(int i = 0; i < N; i++)
    a[i] = 0;

#pragma acc parallel loop
for(int j = 0; j < M; j++)
    b[j] = 0;
```

- To parallelize multiple loops, each loop should be accompanied by a parallel directive
- Each parallel loop can have different loop boundaries and loop optimizations
- Each parallel loop can be parallelized in a different way
- This is the recommended way to parallelize multiple loops. Attempting to parallelize multiple loops within the same parallel region may give performance issues or unexpected results



# OPENACC LOOP DIRECTIVE



# OPENACC LOOP DIRECTIVE

## Expressing parallelism

- Mark a single for loop for parallelization
- Allows the programmer to give additional information and/or optimizations about the loop
- Provides many different ways to describe the type of parallelism to apply to the loop
- Must be contained within an OpenACC compute region (either a kernels or a parallel region) to parallelize loops

C/C++

```
#pragma acc loop
for(int i = 0; i < N; i++)
    // Do something
```

# OPENACC LOOP DIRECTIVE

## Inside of a parallel compute region

```
#pragma acc parallel
{
    for(int i = 0; i < N; i++)
        a[i] = 0;

    #pragma acc loop
    for(int j = 0; j < N; j++)
        a[i]++;
}
```

- In this example, the first loop is not marked with the loop directive
- This means that the loop will be “redundantly parallelized”
- Redundant parallelization, in this case, means that the loop will be run in its entirety, multiple times, by the parallel hardware
- The second loop is marked with the loop directive, meaning that the loop iterations will be properly split across the parallel hardware

# OPENACC LOOP DIRECTIVE

Inside of a kernels compute region

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

    #pragma acc loop
    for(int j = 0; j < M; j++)
        b[i] = 0;
}
```

- With the kernels directive, the loop directive is implied
- The programmer can still explicitly define loops with the loop directive, however this could affect the optimizations the compiler makes
- The loop directive is not needed, but does allow the programmer to optimize the loops themselves



# OPENACC LOOP DIRECTIVE

Parallelizing loop nests

C/C++

```
+ #pragma acc parallel loop
# pragma acc loop
for(int i = 0; i < N; i++){
    for(int j = 0; j < M; j++){
        a[i][j] = 0;
    }
}
```

- You are able to include multiple loop directives to parallelize multi-dimensional loop nests
- On some parallel hardware, this will allow you to express more levels of parallelism, and increase performance further
- Other parallel hardware has difficulties expressing enough parallelism for multi-dimensional loops
- In this case, inner loop directives may be ignored



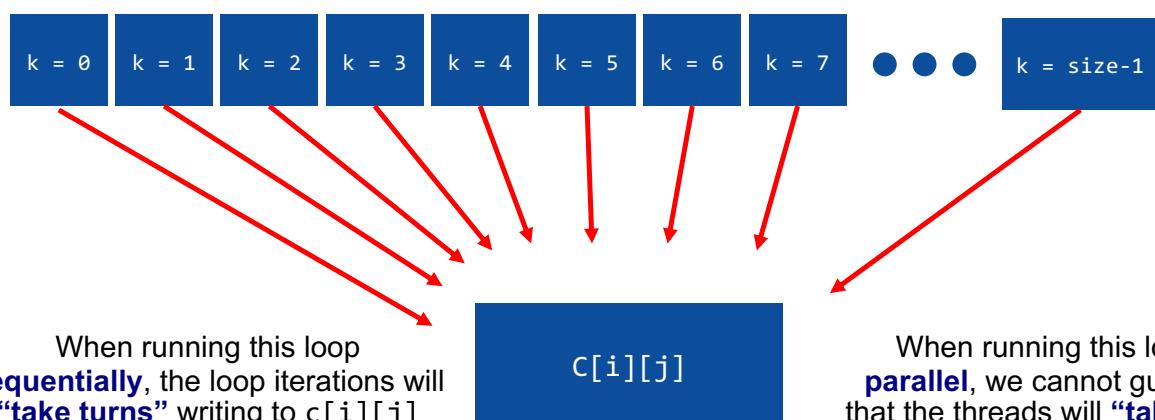
# REDUCTION CLAUSE

```
for( i = 0; i < size; i++ )  
    for( j = 0; j < size; j++ )  
        for( k = 0; k < size; k++ )  
            c[i][j] += a[i][k] * b[k][j];
```

- The inner-most loop is not parallelizable
- If we attempted to parallelize it without any changes, multiple threads could attempt to write to `c[i][j]`
- When multiple threads try to write to the same place in memory simultaneously, we should expect to receive erroneous results
- To fix this, we should use the **reduction clause**

## WITHOUT A REDUCTION

```
#pragma acc parallel loop  
for( k = 0; k < size; k++ )  
    c[i][j] += a[i][k] * b[k][j];
```



# REDUCTION CLAUSE

- The **reduction** clause is used when taking many values and “reducing” it to a single value such as in a summation
- Each thread will have their own private copy of the reduction variable and perform a partial reduction on the loop iterations that they compute
- After the loop, the reduction clause will perform a final reduction to produce a **single global result**

```
for( i = 0; i < size; i++ )
    for( j = 0; j < size; j++ )
        for( k = 0; k < size; k++ )
            c[i][j] += a[i][k] *
b[k][j];
```

```
for( i = 0; i < size; i++ )
    for( j = 0; j < size; j++ )
        double tmp = 0.0F;
#pragma parallel acc loop reduction(+:tmp)
for( k = 0; k < size; k++ )
    tmp += a[i][k] * b[k][j];
c[i][j] = tmp;
```

# REDUCTION CLAUSE

- The compiler is often very good at detecting when a reduction is needed so the clause may be optional
- May be more applicable to the parallel directive (depending on the compiler)

```
for( i = 0; i < size; i++ )
    for( j = 0; j < size; j++ )
        double tmp = 0.0F;
#pragma parallel acc loop reduction(+:tmp)
for( k = 0; k < size; k++ )
    tmp += a[i][k] * b[k][j];
c[i][j] = tmp;
```

# REDUCTION CLAUSE OPERATORS

Operator	Description	Example
+	Addition/Summation	reduction(+:sum)
*	Multiplication/Product	reduction(*:product)
max	Maximum value	reduction(max:maximum)
min	Minimum value	reduction(min:minimum)
&	Bitwise and	reduction(&:val)
	Bitwise or	reduction(  :val)
&&	Logical and	reduction(&&:val)
	Logical or	reduction(  :val)

# REDUCTION CLAUSE Restrictions

- The reduction variable may not be an array element

```
a[0] = 0;
#pragma parallel acc loop \
    reduction(+:a[0])
for( i = 0; i < 100; i++ )
    a[0] += i;
```

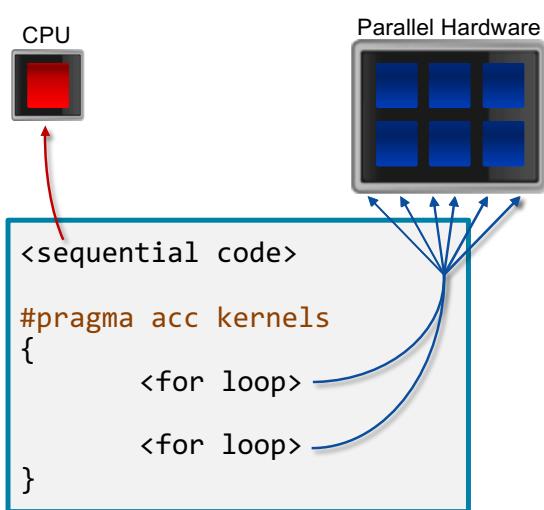
- The reduction variable may not be a C struct member, a C++ class or struct member, or a Fortran derived type member

```
v.val = 0;
#pragma acc parallel loop \
    reduction(+:v.val)
for( i = 0; i < v.n; i++ )
    v.val += i;
```

# OPENACC KERNELS DIRECTIVE

## OPENACC KERNELS DIRECTIVE

Compiler directed parallelization



- The kernels directive instructs the compiler to search for parallel loops in the code
- The compiler will analyze the loops and parallelize those it finds safe and profitable to do so
- The kernels directive can be applied to regions containing multiple loop nests

# OPENACC KERNELS DIRECTIVE

## Parallelizing a single loop

C/C++

```
#pragma acc kernels
for(int i = 0; j < N; i++)
    a[i] = 0;
```

- In this example, the kernels directive applies to the next for loop
- The compiler will take the loop, and attempt to parallelize it on the parallel hardware
- The compiler will also attempt to optimize the loop
- If the compiler decides that the loop is not parallelizable, it will not parallelize the loop

# OPENACC KERNELS DIRECTIVE

## Parallelizing many loops

C/C++

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

    for(int j = 0; j < M; j++)
        b[j] = 0;
}
```

- In this example, we mark a region of code with the kernels directive
- The kernels region is defined by the **curly braces**
- The compiler will attempt to parallelize all loops within the kernels region
- Each loop can be parallelized/optimized in a different way

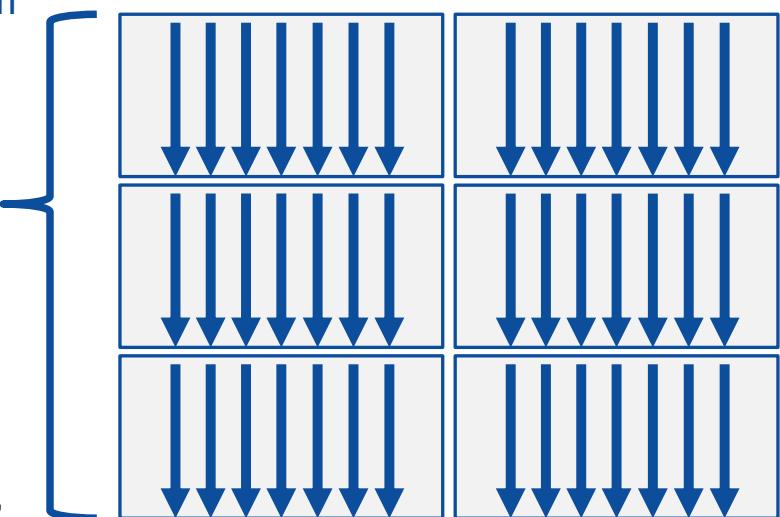
# EXPRESSING PARALLELISM

Compiler generated parallelism

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

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

OpenACC NVIDIA DEEP LEARNING INSTITUTE



# EXPRESSING PARALLELISM

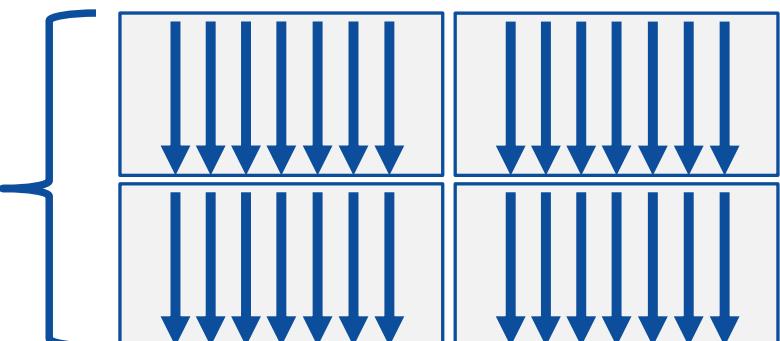
Compiler generated parallelism

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

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

OpenACC NVIDIA DEEP LEARNING INSTITUTE

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



# KERNELS VS PARALLEL

## Kernels

- Compiler decides what to parallelize with direction from user
- Compiler guarantees correctness
- Can cover multiple loop nests

## Parallel

- Programmer decides what to parallelize and communicates that to the compiler
- Programmer guarantees correctness
- Must decorate each loop nest

When fully optimized, both will give similar performance.



# THANK YOU

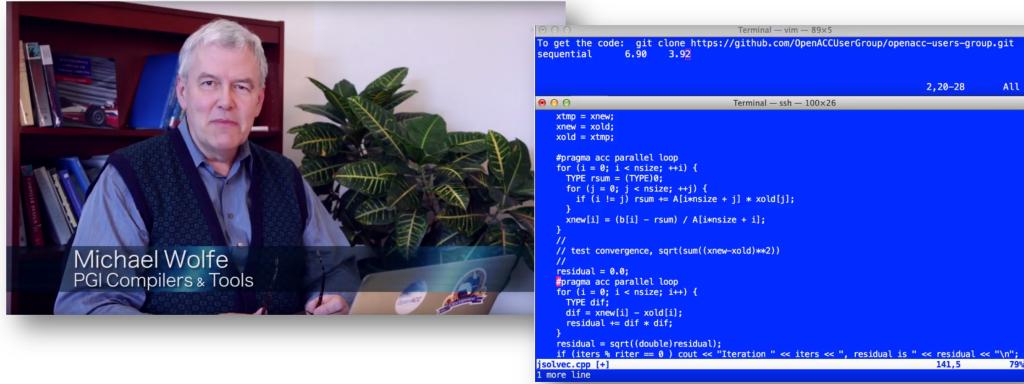


# ADDITIONAL RESOURCES

## YouTube OpenACC Introduction Series by Michael Wolfe

[Introduction to Parallel Programming with OpenACC – Part 1](#)

[Introduction to Parallel Programming with OpenACC – Part 2](#)



**OpenACC**  
More Science. Less Programming.



DEEP  
LEARNING  
INSTITUTE

[Follow along by downloading the code here!](#)

# OPENACC RESOURCES

Guides • Talks • Tutorials • Videos • Books • Spec • Code Samples • Teaching Materials • Events • Success Stories • Courses • Slack • Stack Overflow



**slack**  
<https://www.openacc.org/community#slack>

## Resources

<https://www.openacc.org/resources>

## Compilers and Tools

<https://www.openacc.org/tools>

## Success Stories

<https://www.openacc.org/success-stories>

## Events

<https://www.openacc.org/events>

**OpenACC**  
More Science. Less Programming.



DEEP  
LEARNING  
INSTITUTE