

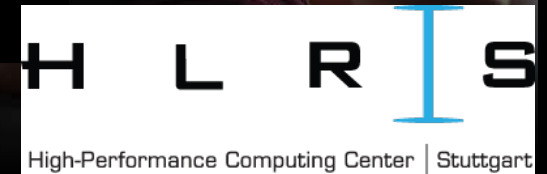


DEEP  
LEARNING  
INSTITUTE



# Deep Learning and GPU programming using OpenACC

14 – 17 July 2020



# MODULE THREE: OPENACC DIRECTIVES

Dr. Volker Weinberg | LRZ | 16.07.2020

# 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>
```

Fortran

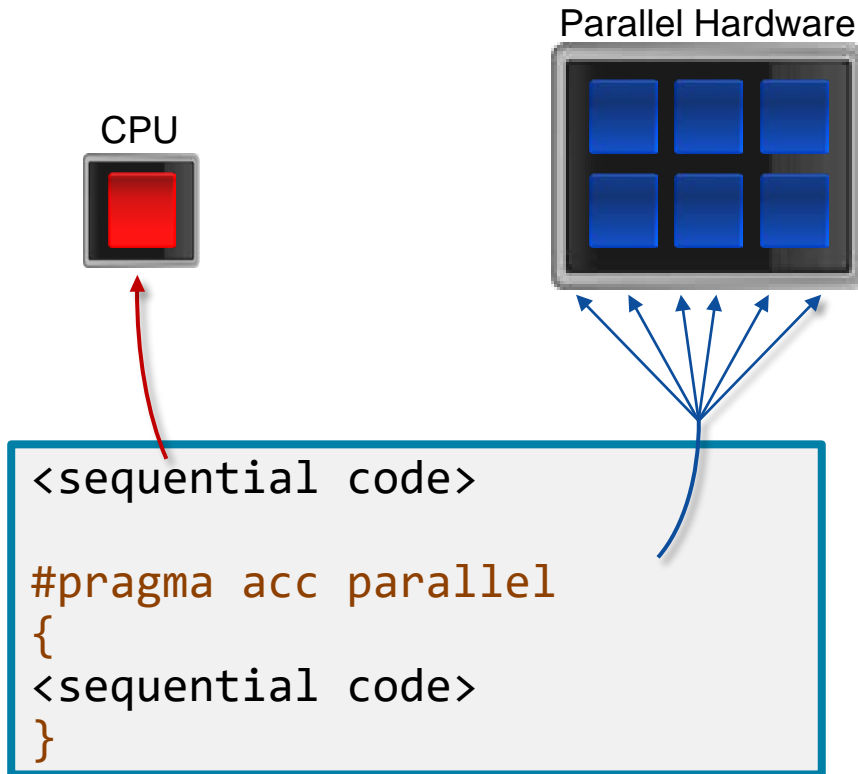
```
!$acc directive clauses  
<code>
```

- 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.
- A ***directive*** in Fortran is a specially formatted comment that likewise instructs the compiler in its compilation of the code and can be freely ignored.
- “***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

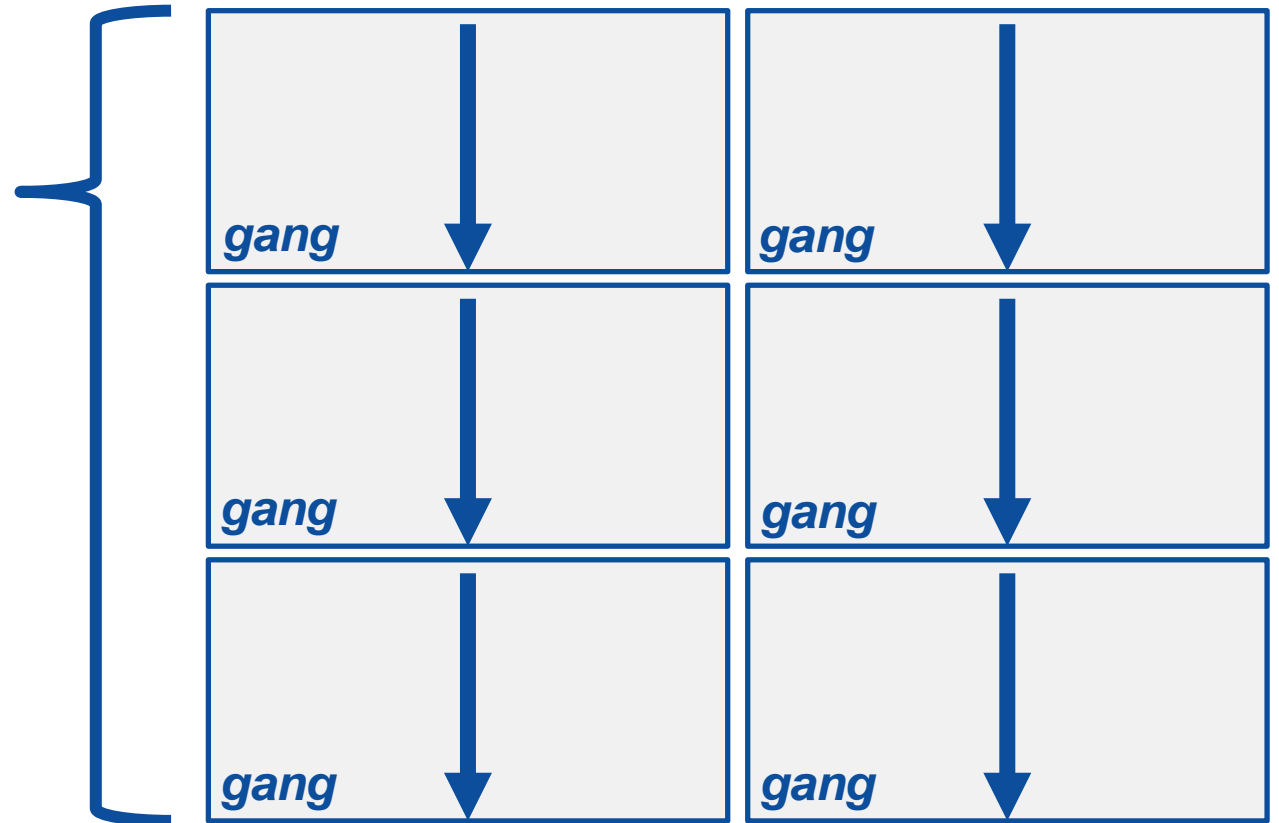
# OPENACC PARALLEL DIRECTIVE

## Expressing parallelism

```
#pragma acc parallel  
{
```

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

```
}
```





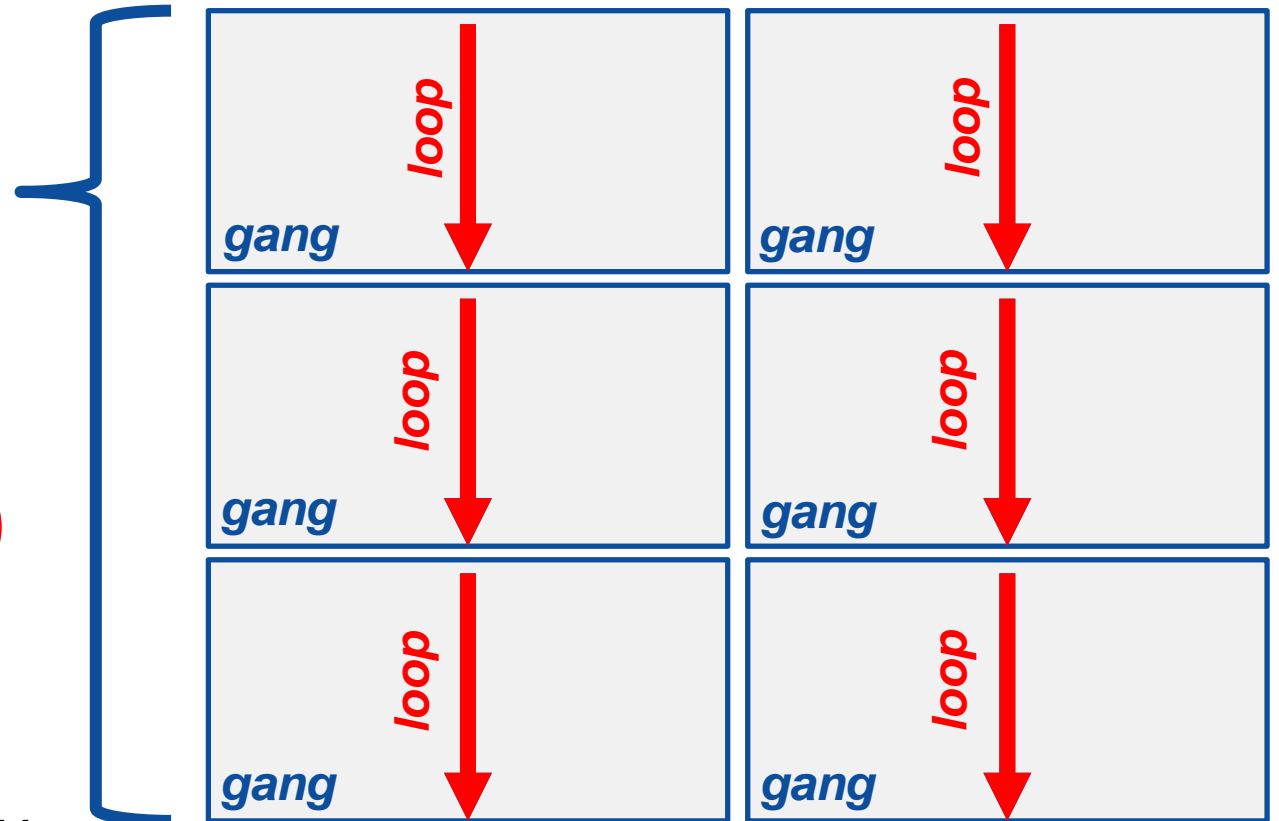
# OPENACC PARALLEL DIRECTIVE

## Expressing parallelism

```
#pragma acc parallel  
{
```

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

This loop will be  
executed redundantly  
on each gang



# OPENACC PARALLEL DIRECTIVE

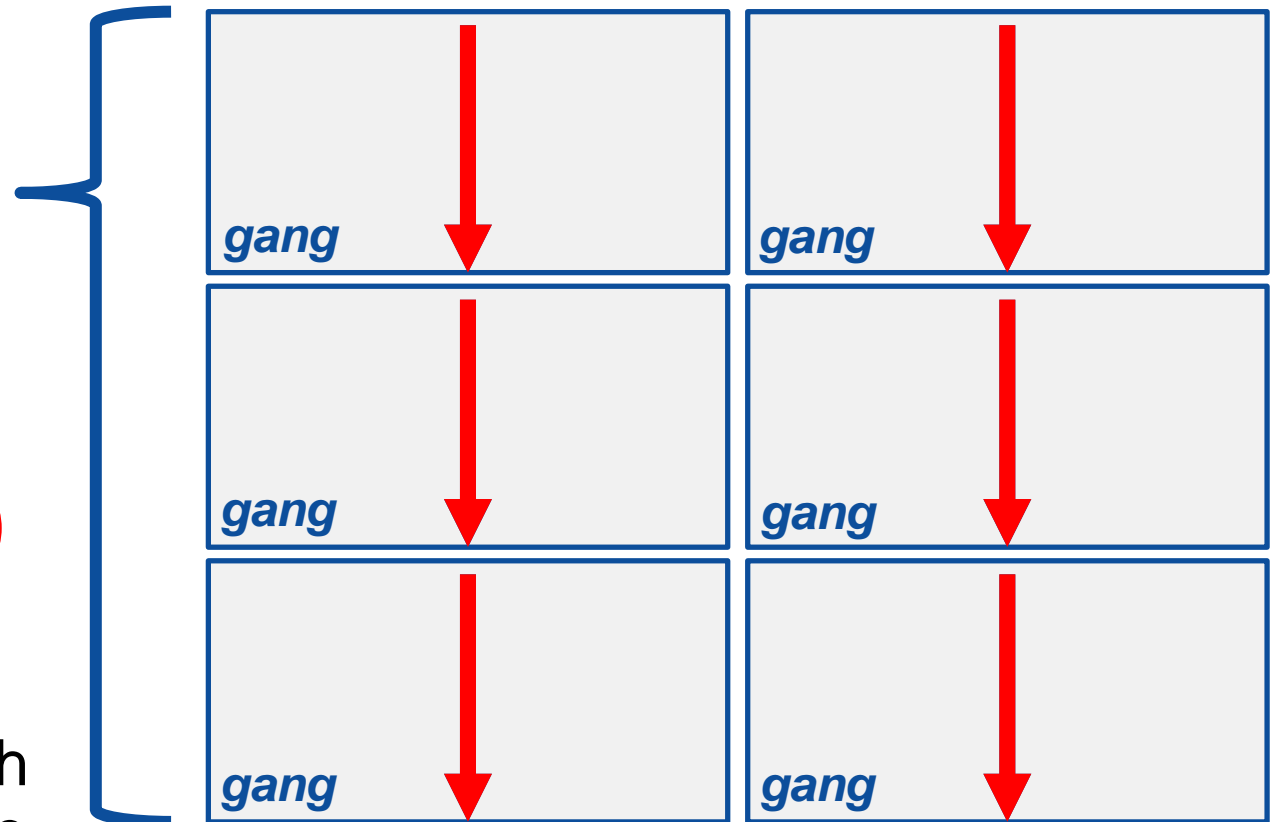
## Expressing parallelism

```
#pragma acc parallel  
{
```

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

```
}
```

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



# OPENACC PARALLEL DIRECTIVE

## Parallelizing a single loop

### C/C++

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

### Fortran

```
!$acc parallel
!$acc loop
do i = 1, N
    a(i) = 0
end do
!$acc end parallel
```

- 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 PARALLEL DIRECTIVE

## Parallelizing a single loop

### C/C++

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

### Fortran

```
!$acc parallel loop
do i = 1, N
    a(i) = 0
end do
```

- 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

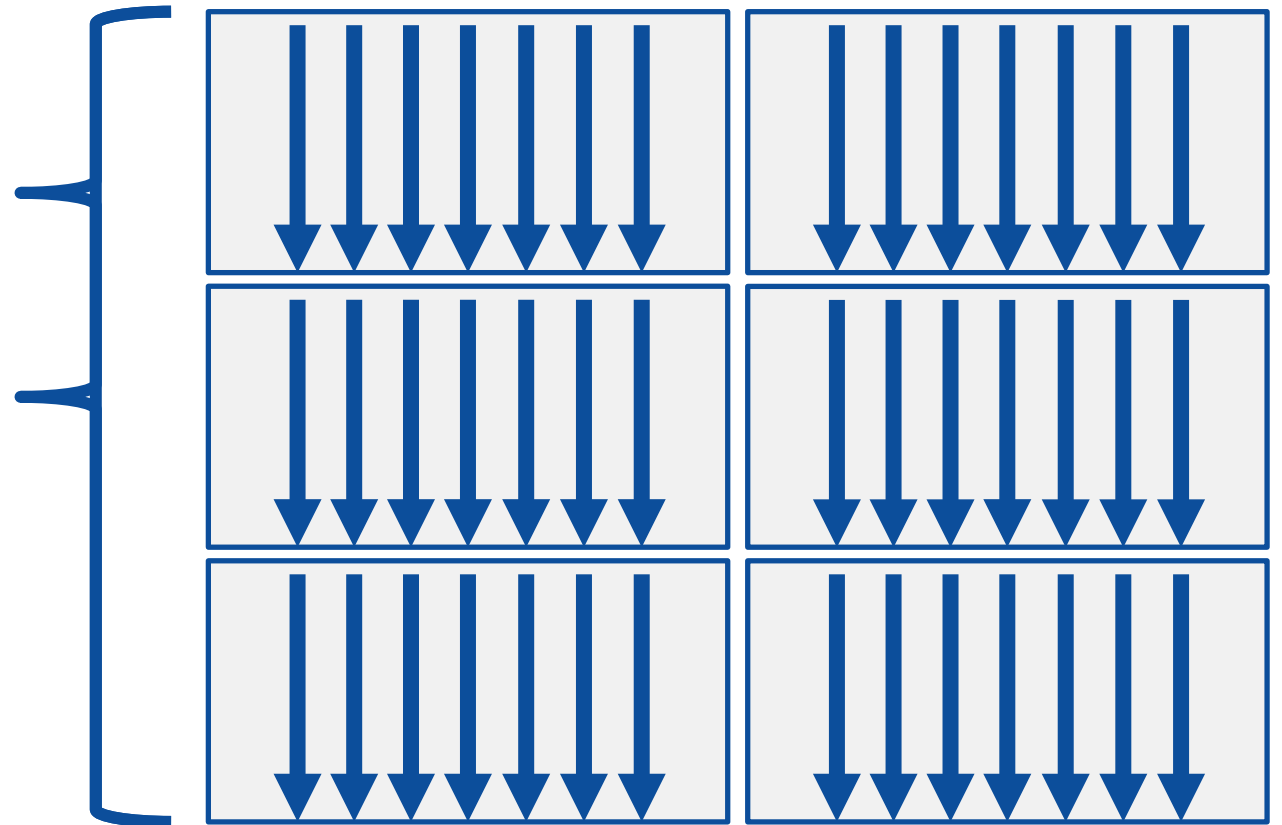
# OPENACC PARALLEL DIRECTIVE

## 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
```

Fortran

```
!$acc loop  
do i = 1, N  
    ! 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[j]++;
}
```

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

Fortran

```
!$acc parallel loop
do i = 1, N
    !$acc loop
    do j = 1, M
        a(i,j) = 0
    end do
end do
```

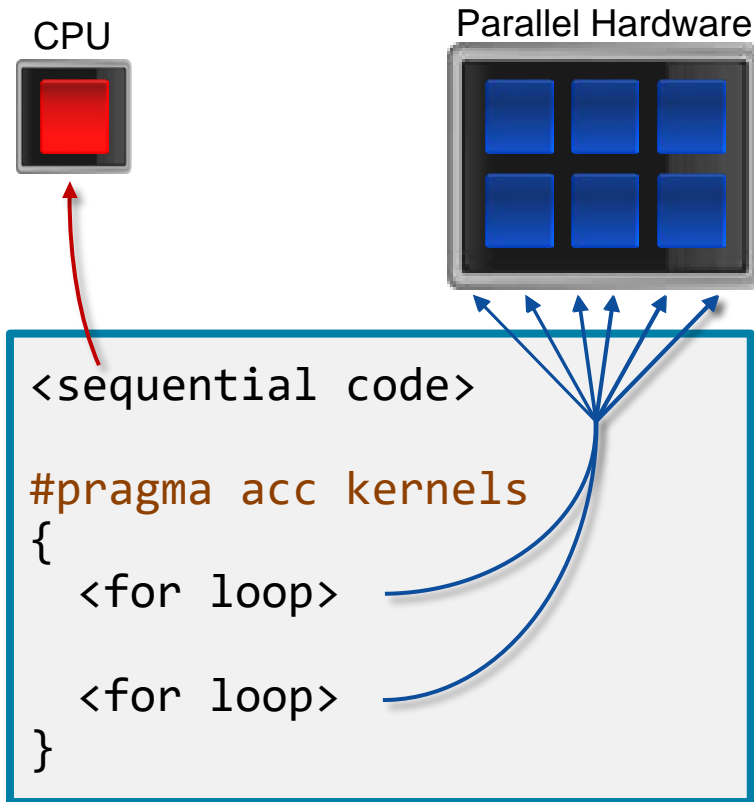
- You are able to include multiple loop directives to parallelize multi-dimensional loop nests
- On some parallel hardware, this will allow you to express more levels of parallelism, and increase performance further
- Other parallel hardware has difficulties expressing enough parallelism for multi-dimensional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

PLEASE START LAB NOW!

# 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;
```

### Fortran

```
!$acc kernels  
do i = 1, N  
    a(i) = 0  
end do  
!$acc end kernels
```

- 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;
}
```

Fortran

```
!$acc kernels
do i = 1, N
  a(i) = 0
end do

do j = 1, M
  b(j) = 0
end do
!$acc end kernels
```

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

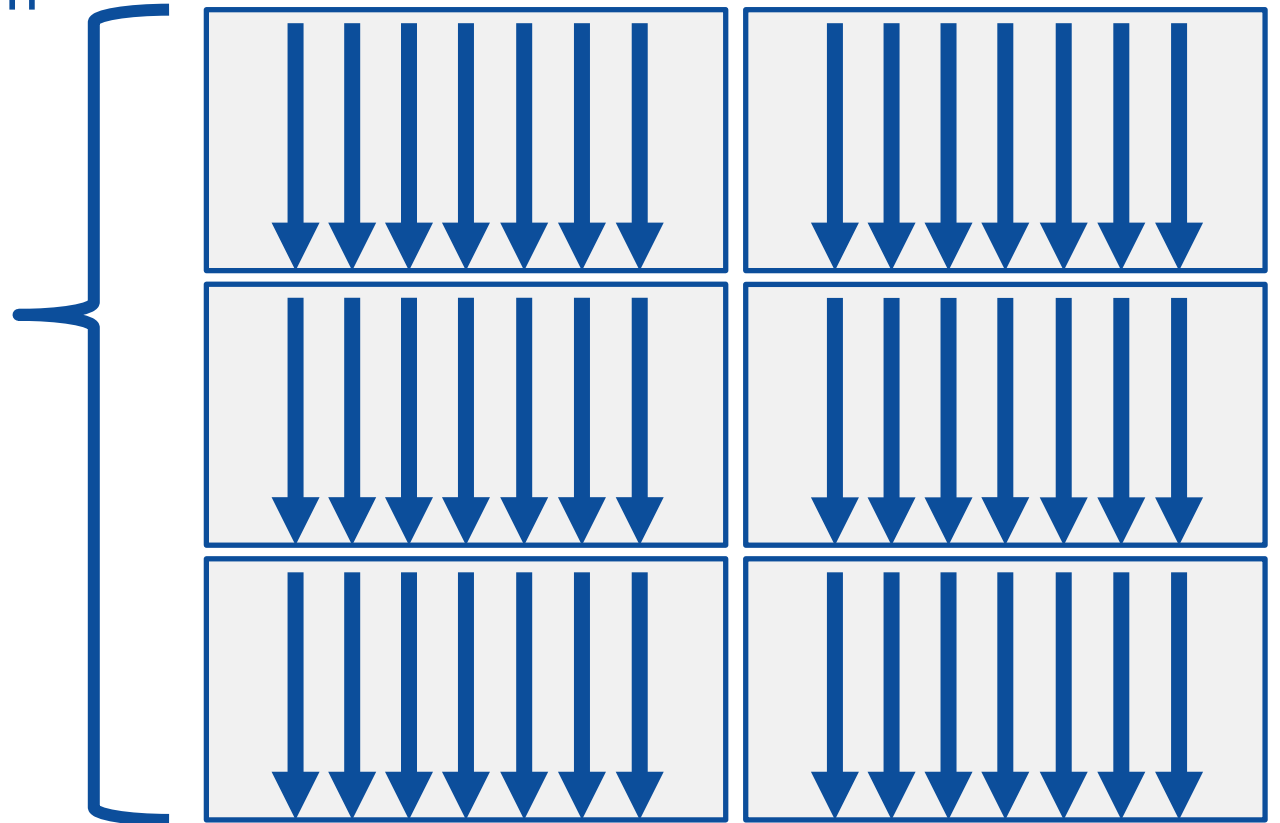


# EXPRESSING PARALLELISM

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

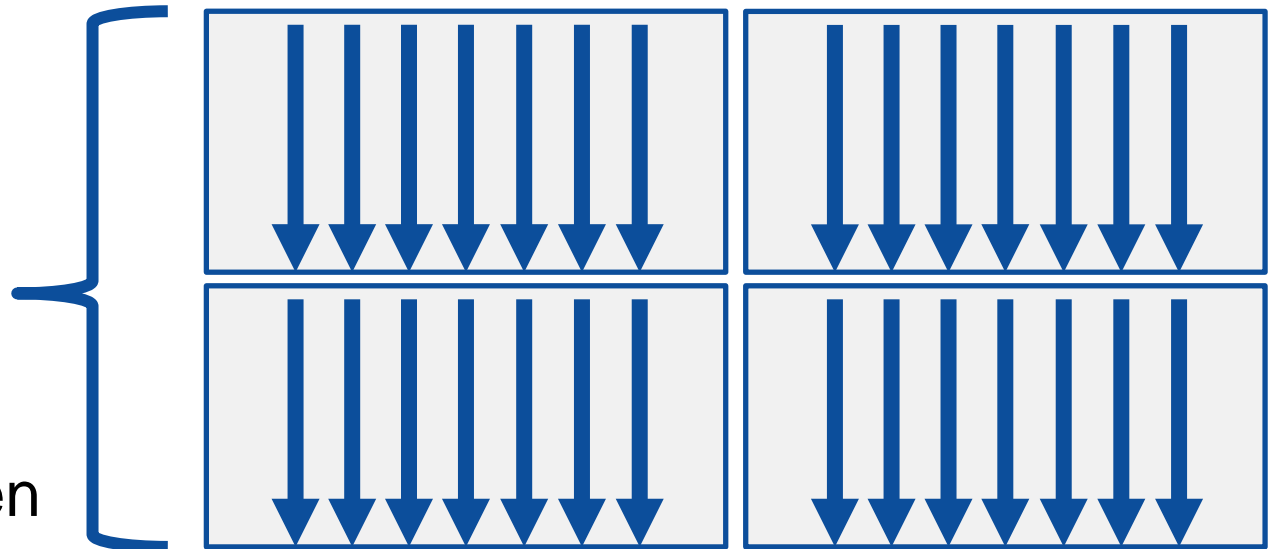


# EXPRESSING PARALLELISM

## 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
    kernels region.
}
```

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



# OPENACC KERNELS DIRECTIVE

## Fortran array syntax

```
!$acc kernels  
a(:) = 1  
b(:) = 2  
c(:) = a(:) + b(:)  
!$acc end kernels
```

```
!$acc parallel loop  
c(:) = a(:) + b(:)
```

- One advantage that the kernels directive has over the parallel directive is Fortran array syntax
- The parallel directive must be paired with the loop directive, and the loop directive does not recognize the array syntax as a loop
- The kernels directive can correctly parallelize the array syntax

# 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.

# COMPILING PARALLEL CODE

# COMPILING PARALLEL CODE (PGI)

## CODE

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

## COMPILING

```
$ pgcc -fast -acc -ta=multicore -Minfo=accel main.c
```

## FEEDBACK

main:

7, Generating Multicore code

8, #pragma acc loop gang

# COMPILING PARALLEL CODE (PGI)

## CODE

```
7: #pragma acc kernels
8: for(int i = 0; i < N; i++)
9:   a[i] = 0;
```

## COMPILING

```
$ pgcc -fast -acc -ta=multicore -Minfo=accel main.c
```

## FEEDBACK

```
main:
  8, Loop is parallelizable
    Generating Multicore code
    8, #pragma acc loop gang
```

# COMPILING PARALLEL CODE (PGI)

## CODE

```
7: #pragma acc kernels
8: for(int i = 1; i < N; i++)
9:   a[i] = a[i-1] + a[i];
```

Non-parallel loop

## COMPILING

```
$ pgcc -fast -acc -ta=multicore -Minfo=accel main.c
```

## FEEDBACK

```
main:
  8, Loop carried dependence of a-> prevents parallelization
     Loop carried backward dependence of a-> prevents vectorization
```



# COMPILING PARALLEL CODE (PGI)

## CODE

```
7: #pragma acc parallel loop
8: for(int i = 1; i < N; i++)
9:   a[i] = a[i-1] + a[i];
```

Non-parallel loop

## COMPILING

```
$ pgcc -fast -acc -ta=multicore -Minfo=accel main.c
```

## FEEDBACK

```
main:
  7, Generating Multicore code
  8, #pragma acc loop gang
```

# KEY CONCEPTS

By end of this module, you should now understand

- The parallel, kernels, and loop directives
- The key differences in functionality and use between the kernels and parallel directives
- When and where to include loop directives
- How the parallel and kernel directives conceptually generate parallelism

# THANK YOU

**OpenACC**  
More Science, Less Programming

# OPENACC RESOURCES

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



**FREE  
Compilers**

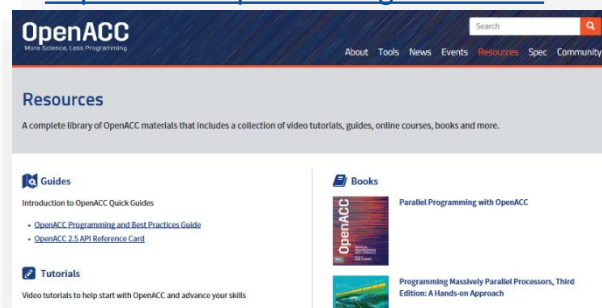
**PGI**  
Community  
EDITION



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

## Resources

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



**OpenACC**  
More Science. Less Programming.

Search

About Tools News Events Resources Spec Community

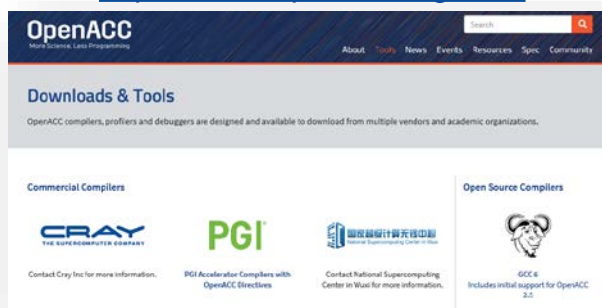
### Resources

A complete library of OpenACC materials that includes a collection of video tutorials, guides, online courses, books and more.

- Guides**
  - Introduction to OpenACC Quick Guides
    - OpenACC Programming and Best Practices Guide
    - OpenACC 2.3 API Reference Card
- Tutorials**
  - Video tutorials to help start with OpenACC and advance your skills
- Books**
  - Parallel Programming with OpenACC
  - Programming Massively Parallel Processors, Third Edition: A Hands-on Approach

## Compilers and Tools

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



**OpenACC**  
More Science. Less Programming.

Search

About Tools News Events Resources Spec Community

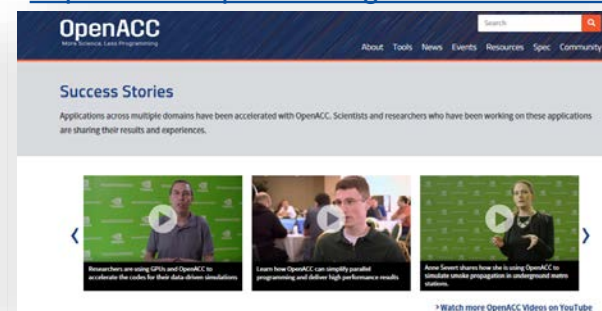
### Downloads & Tools

OpenACC compilers, profilers and debuggers are designed and available to download from multiple vendors and academic organizations.

- Commercial Compilers**
  - CRAY**  
THE SUPERCOMPUTER COMPANY  
Contact Cray Inc for more information.
  - PGI**  
PGI Accelerator Compilers with OpenACC Directives
  - Intel**  
Contact National Supercomputing Center in Wuhan for more information.
- Open Source Compilers**
  - GCC**  
Includes initial support for OpenACC 2.3

## Success Stories

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



**OpenACC**  
More Science. Less Programming.

Search

About Tools News Events Resources Spec Community

### Success Stories

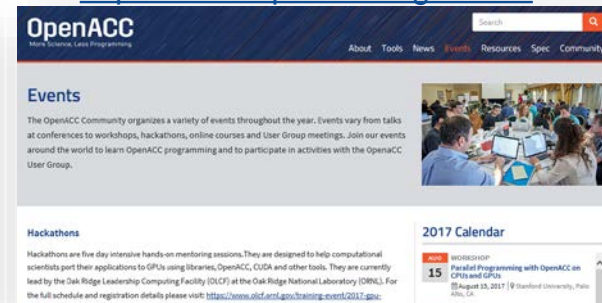
Applications across multiple domains have been accelerated with OpenACC. Scientists and researchers who have been working on these applications are sharing their results and experiences.

- Researchers are using GPUs and OpenACC to accelerate the codes for their data-driven simulations.
- Learn how OpenACC can simplify parallel programming and deliver high performance results.
- Acme Server shows how she is using OpenACC to simulate smolder propagation in underground mine systems.

Watch more OpenACC Videos on YouTube

## Events

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



**OpenACC**  
More Science. Less Programming.

Search

About Tools News Events Resources Spec Community

### Events

The OpenACC Community organizes a variety of events throughout the year. Events vary from talks at conferences to workshops, hackathons, online courses and User Group meetings. Join our events around the world to learn OpenACC programming and to participate in activities with the OpenACC User Group.

- Hackathons**  
Hackathons are five day intensive hands-on mentoring sessions. They are designed to help computational scientists port their applications to GPUs using libraries, OpenACC, CUDA and other tools. They are currently lead by the Oak Ridge Leadership Computing Facility (OLCF) at the Oak Ridge National Laboratory (ORNL). For the full schedule and registration details please visit: <https://www.olcf.ornl.gov/training-event2017-gpu/>.
- 2017 Calendar**
  - JULY** WORKSHOP  
Parallel Programming with OpenACC on CPUs and GPUs  
15 August 15, 2017 | Stanford University, Palo Alto, CA