# Module I: Introduction to OpenACC

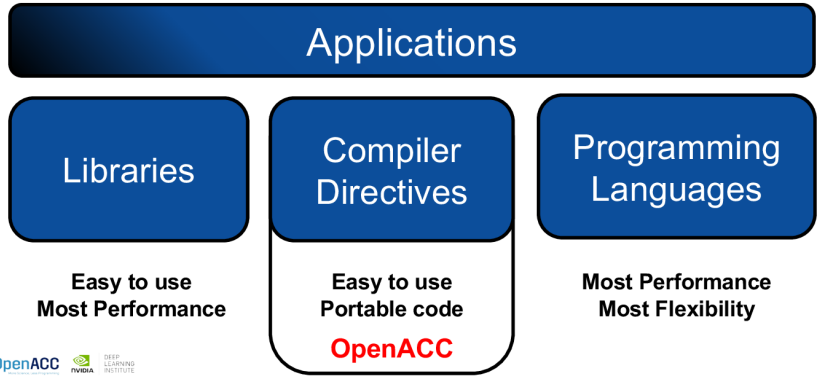
OpenACC More Science, Less Programming | NVIDIA | DEEP LEARNING INSTITUTE

Similarly to **OpenMP, OpenACC** is a directives-based programming approach to **parallel computing** but designed for **performance** and **portability** on CPUs and GPUs for HPC.

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

# 3 WAYS TO ACCELERATE APPLICATIONS





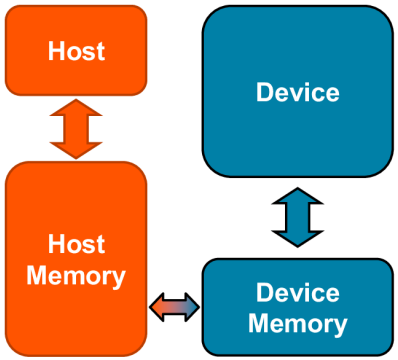

# OPENACC PORTABILITY

## Describing a generic parallel machine

OpenACC is designed to be portable to many existing and future parallel platforms

The programmer need not think about specific hardware details, but rather express the parallelism in generic terms

An OpenACC program runs on a *host* (typically a CPU) that manages one or more parallel *devices* (GPUs, etc.). The host and device(s) are logically thought of as having separate memories.



# OPENACC

## Three major strengths

| Incremental | Single Source | Low Learning Curve |
| --- | --- | --- |
| | | |

OpenACC NVIDIA DEEP LEARNING INSTITUTE

# OPENACC

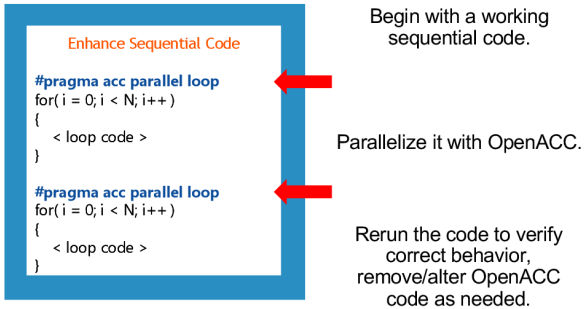
**Incremental**

Maintain existing sequential code

Add annotations to expose parallelism

After verifying correctness, annotate more of the code

OpenACC NVIDIA DEEP LEARNING INSTITUTE




Begin with a working sequential code.

Parallelize it with OpenACC.

Rerun the code to verify correct behavior, remove/alter OpenACC code as needed.

# OPENACC

**Incremental**

Maintain existing sequential code

Add annotations to expose parallelism

After verifying correctness, annotate more of the code

**Single Source**

**Low Learning Curve**

OpenACC NVIDIA DEEP LEARNING INSTITUTE

# OPENACC

**Supported Platforms**

POWER

Sunway

x86 CPU

x86 Xeon Phi

NVIDIA GPU

PEZY-SC

**Single Source**

Rebuild the same code on multiple architectures

Compiler determines how to parallelize for the desired machine

Sequential code is maintained

The compiler can **ignore** your OpenACC code additions, so the same code can be used for **parallel** or **sequential** execution.

```
int main(){

...

#pragma acc parallel loop
for(int i = 0; i < N; i++)
< loop code >

}
```

OpenACC NVIDIA DEEP LEARNING INSTITUTE

# OPENACC

**Incremental**

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

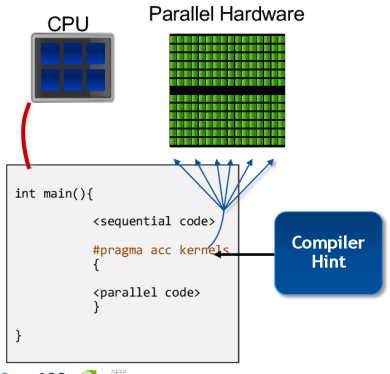
**Single Source**

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

**Low Learning Curve**

OpenACC NVIDIA DEEP LEARNING INSTITUTE

# OPENACC




The programmer will give hints to the compiler about which parts of the code to parallelize.

The compiler will then generate parallelism for the target parallel hardware.

**Low Learning Curve**

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.

OpenACC NVIDIA DEEP LEARNING INSTITUTE

# OPENACC

**Incremental**

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

**Single Source**

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

**Low Learning Curve**

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.



# EXPRESSING PARALLELISM WITH OPENACC

# PGI COMPILER BASICS

## -Minfo flag

The Minfo flag will instruct the compiler to print feedback about the compiled code

-Minfo=accel will give us information about what parts of the code were accelerated via OpenACC

-Minfo=opt will give information about all code optimizations

-Minfo=all will give all code feedback, whether positive or negative

```
$       pgcc –fast –Minfo=all main.c
$       pgc++ -fast -Minfo=all main.cpp
```



# GCC COMPILER BASICS

## gcc, gc++ and gfortran

The command to compile C code is ‘gcc’

The command to compile C++ code is ‘g++’

The command to compile Fortran code is ‘gfortran’

The -O2 flag sets the optimization level to 2 (a safe starting point)

```
$       gcc -O2 main.c
$       g++ -O2 main.cpp
```

# GCC COMPILER BASICS

## Compiler feedback

The -fopt-info flag will print limited compiler feedback

The -flto-report flag will also print link-time optimizations, but should be used sparingly due to volume of information

```
$      gcc -O2 -fopt-info main.c
$      g++ -O2 -fopt-info main.cpp
```



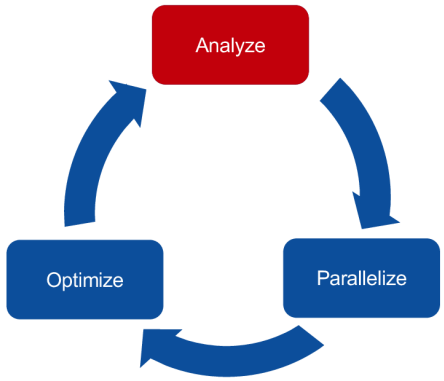
# PROFILING SEQUENTIAL CODE

# OPENACC DEVELOPMENT CYCLE

**Analyze** your code to determine most likely places needing parallelization or optimization.

**Parallelize** your code by starting with the most time consuming parts, check for correctness and then analyze it again.

**Optimize** your code to improve observed speed-up from parallelization.






# PROFILING SEQUENTIAL CODE

**Step 1: Run Your Code**

Record the time it takes for your sequential program to run.

Note the final results to verify correctness later.

Always run a problem that is representative of your real jobs.

**Terminal Window**

```
$ pgcc -fast jacobi.c laplace2d.c
$       ./a.out
          0, 0.250000
        100, 0.002397
        200, 0.001204
        300, 0.000804
        400, 0.000603
        500, 0.000483
        600, 0.000403
        700, 0.000345
        800, 0.000302
        900, 0.000269
        total: 39.432648 s
```

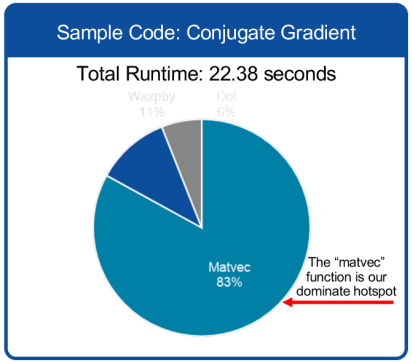
# PROFILING SEQUENTIAL CODE

**Step 2: Profile Your Code**

Obtain detailed information about how the code ran.

This can include information such as:
- Total runtime
- Runtime of individual routines
- Hardware counters

Identify the portions of code that took the longest to run. We want to focus on these "hotspots" when parallelizing.






# PROFILING SEQUENTIAL CODE

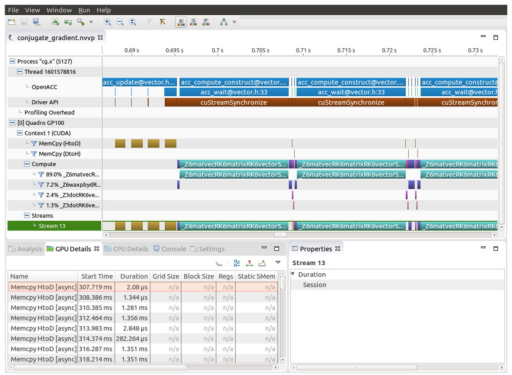
## Introduction to PGProf

Gives visual feedback of how the code ran

Gives numbers and statistics, such as program runtime

Also gives runtime information for individual functions/loops within the code

Includes many extra features for profiling parallel code



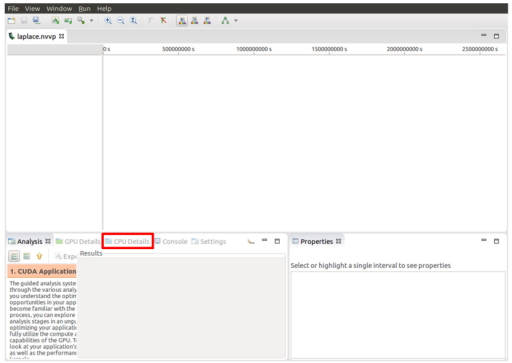
# PROFILING SEQUENTIAL CODE

## First sight when using PGPROF

Profiling a simple, sequential code

Our sequential program will on run on the CPU

To view information about how our code ran, we should select the “CPU Details” tab





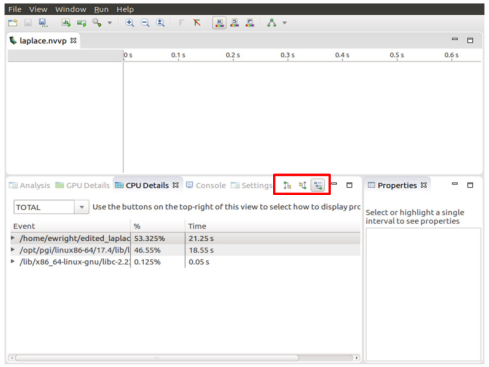

# PROFILING SEQUENTIAL CODE

## CPU Details

Within the “CPU Details” tab, we can see the various parts of our code, and how long they took to run

We can reorganize this info using the three options in the top-right portion of the tab

We will expand this information, and see more details about our code



# PROFILING SEQUENTIAL CODE

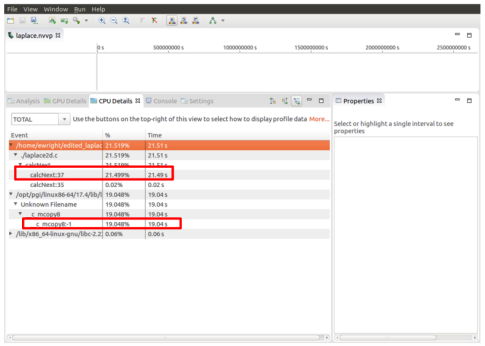
## CPU Details

We can see that there are two places that our code is spending most of its time

21.49 seconds in the “calcNext” function

19.04 seconds in a memcpy function

The c_mcopy8 that we see is actually a compiler optimization that is being applied to our “swap” function



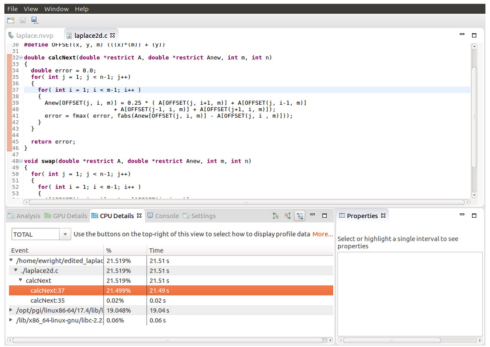



# PROFILING SEQUENTIAL CODE

## PGPROF

We are also able to select the different elements in the CPU Details by double-clicking to open the associated source code

Here we have selected the “calcNext:37” element, which opened up our code to show the exact line (line 37) that is associated with that element



# PROFILING SEQUENTIAL CODE

**Step 2: Profile Your Code**

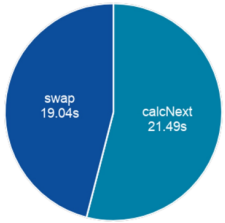
Obtain detailed information about how the code ran.

This can include information such as:
- Total runtime
- Runtime of individual routines
- Hardware counters

Identify the portions of code that took the longest to run. We want to focus on these "hotspots" when parallelizing.

**Lab Code: Laplace Heat Transfer**

Total Runtime: 39.43 seconds






# PROFILING SEQUENTIAL CODE

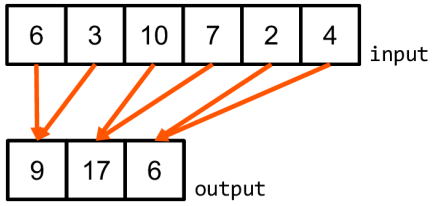
**Step 3: Identify Parallelism**

Observe the loops contained within the identified hotspots

Are these loops parallelizable?
Can the loop iterations execute independently of each other?
Are the loops multi-dimensional, and does that make them very large?

Loops that are good to parallelize tend to have a lot of iterations to map to parallel hardware.

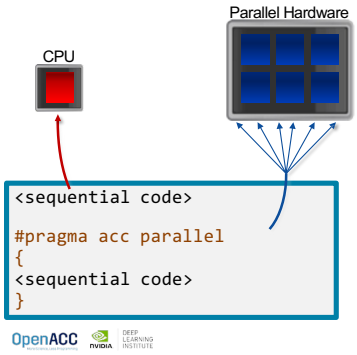
```
void pairing(int *input, int *output, int N){

for(int i = 0; i < N; i++)
    output[i] = input[i*2] + input[i*2+1];
}
```



# 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

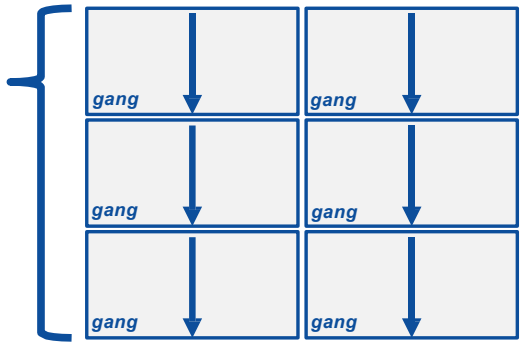
The code contained within a parallel directive is executed redundantly by all parallel gangs

# 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.



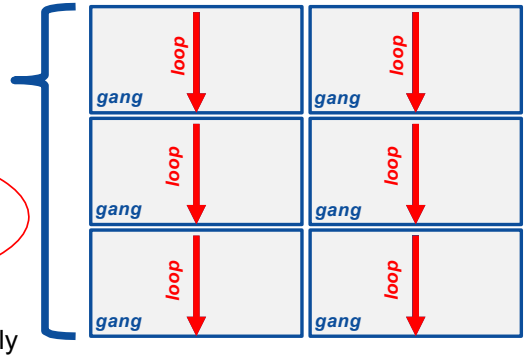

# OPENACC PARALLEL DIRECTIVE

## Expressing parallelism

```
#pragma acc parallel
{

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

}
```

loop

This loop will be executed redundantly on each gang

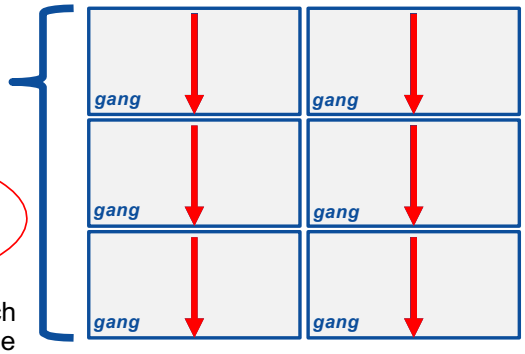


# OPENACC PARALLEL DIRECTIVE

## 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; 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 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.

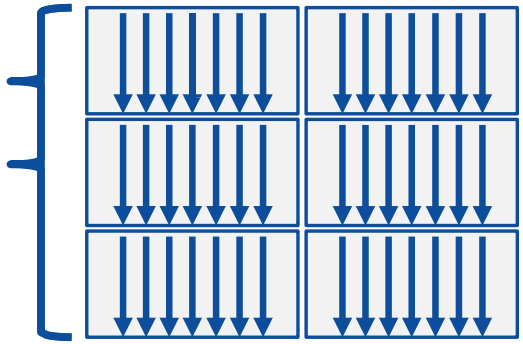
When applied to a loop with a data dependency, parallel loop may produce incorrect results

# 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.



# 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;
```

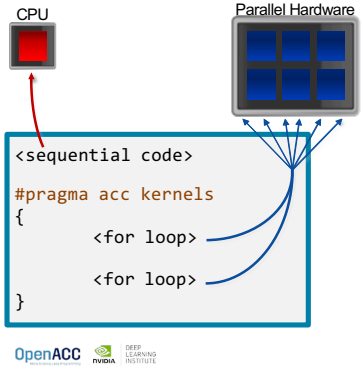
The reduction variable may not be a C struct member, a C++ class or struct member, or a Fortran derived type member

```
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

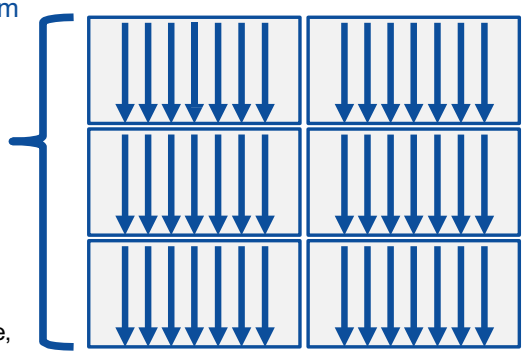
Each loop can be parallelized/optimized in a different way

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



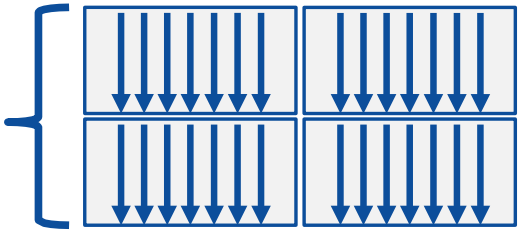
# EXPRESSING PARALLELISM

## 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.



# KERNELS VS PARALLEL

| Kernels | Parallel |
| --- | --- |
| Compiler decides what to parallelize with direction from user | Programmer decides what to parallelize and communicates that to the compiler |
| Compiler guarantees correctness | Programmer guarantees correctness |
| Can cover multiple loop nests | Must decorate each loop nest |

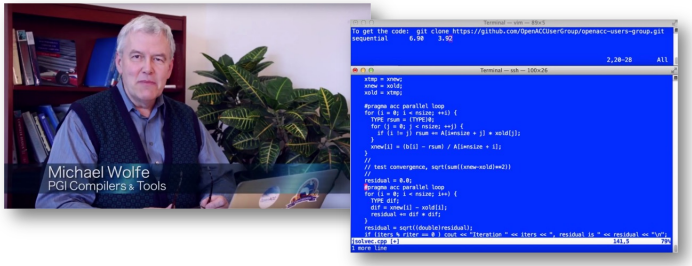


# ADDITIONAL RESOURCES

## YouTube OpenACC Introduction Series by Michael Wolfe

Introduction to Parallel Programming with OpenACC – Part 1

Introduction to Parallel Programming with OpenACC – Part 2




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

**FREE Compilers**

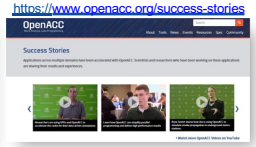
PGI Community Edition

slack

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

**Resources**
https://www.openacc.org/resources

**Success Stories**
https://www.openacc.org/success-stories



**Compilers and Tools**
https://www.openacc.org/tools

**Events**
https://www.openacc.org/events

OpenACC | NVIDIA DEEP LEARNING INSTITUTE