OPENACC

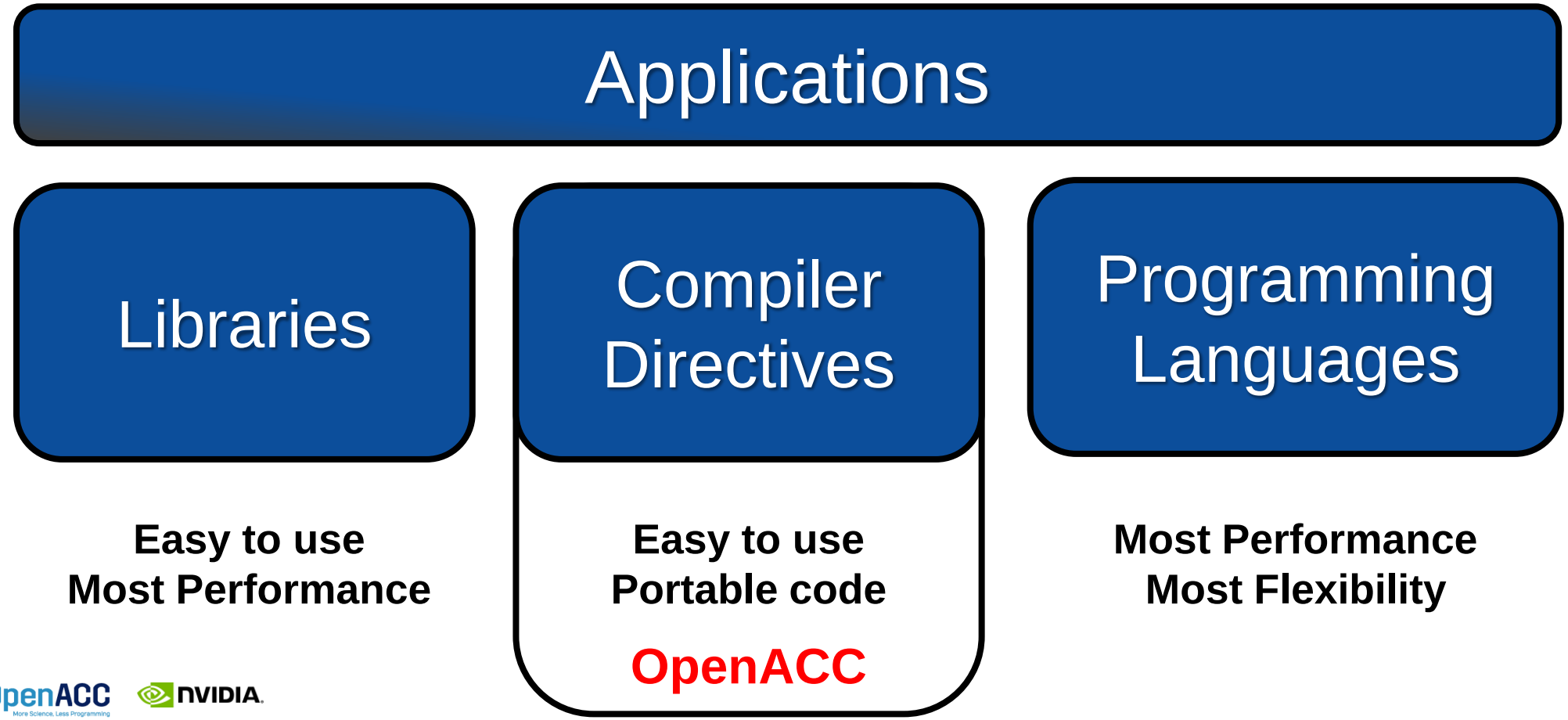
Introduction to OpenACC

OpenACC
More Science, Less Programming

NVIDIA

INTRODUCTION TO OPENACC

3 WAYS TO ACCELERATE APPLICATIONS



OpenACC More Science, Less Programming

NVIDIA

OPENACC IS…

a directives-based **parallel programming model** designed for **performance** and **portability**.

Add Simple Compiler Directive

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

OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang
      for (i = 0; i < n; ++i) {
  #pragma acc loop vector
         for (j = 0; j < n; ++j) {
           c[i][j] = a[i][j] + b[i][j];
           ...
         }
      }
  }
  ...
}
```

OpenACC
Directives for Accelerators

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore

OPENACC: INCREMENTAL

Incremental

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

Enhance Sequential Code

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

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

Begin with a working sequential code.

Parallelize it with OpenACC.

Rerun the code to verify correctness and performance

OPENACC: SINGLE SOURCE

Supported Platforms

POWER

Sunway

x86 CPU

ARM CPU

AMD GPU

NVIDIA GPU

PEZY-SC

Single Source

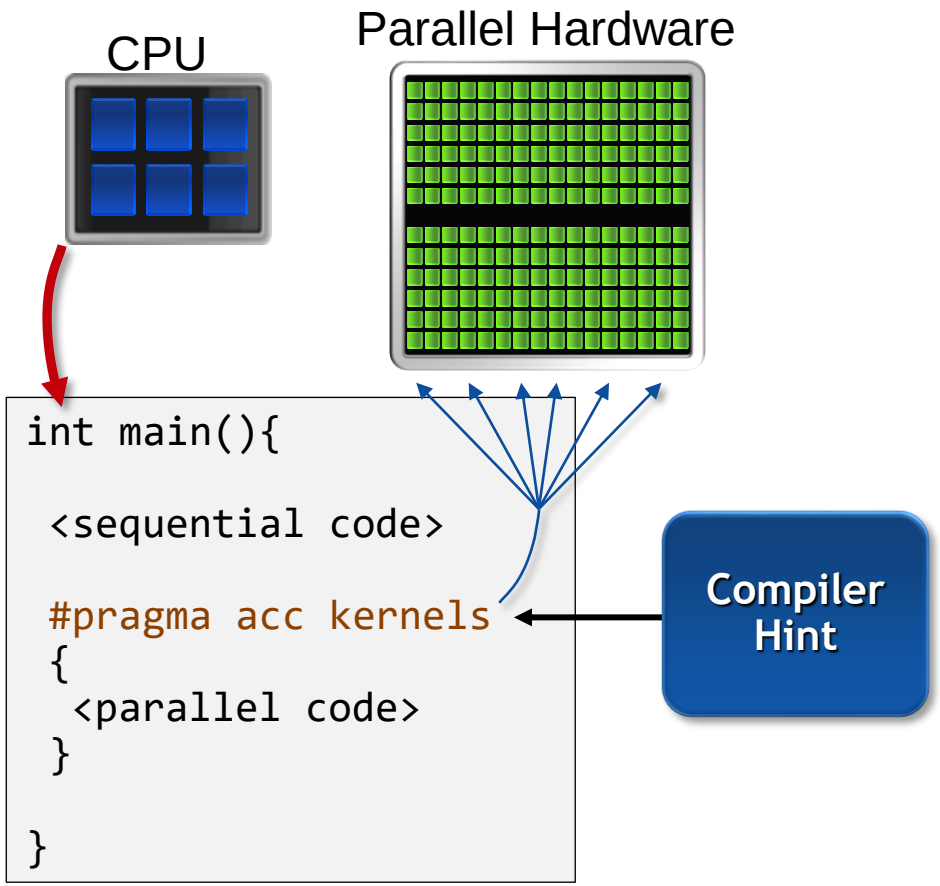
- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

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 More Science, Less Programming NVIDIA

OPENACC: PROGRAMMABILITY



The programmer will give hints to the compiler.

The compiler parallelizes the code.

Low Learning Curve

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

OpenACC More Science, Less Programming NVIDIA

DIRECTIVE-BASED HPC PROGRAMMING

Who's Using OpenACC?

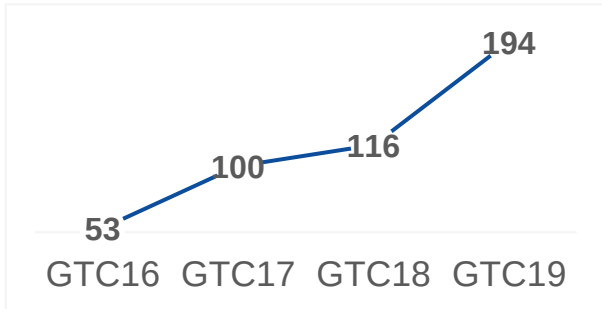
3 OF TOP 5 HPC APPS

Intersect360 RESEARCH

5 OF 13 CAAR CODES

summit

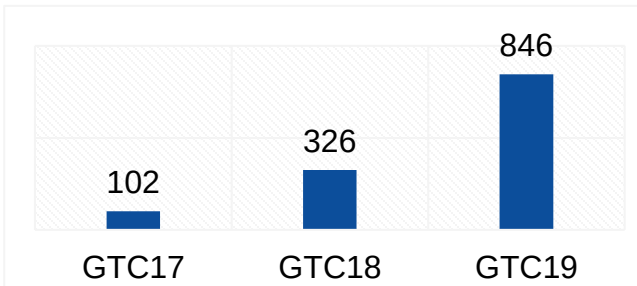
ACCELERATED APPS



725 TRAINED EXPERTS

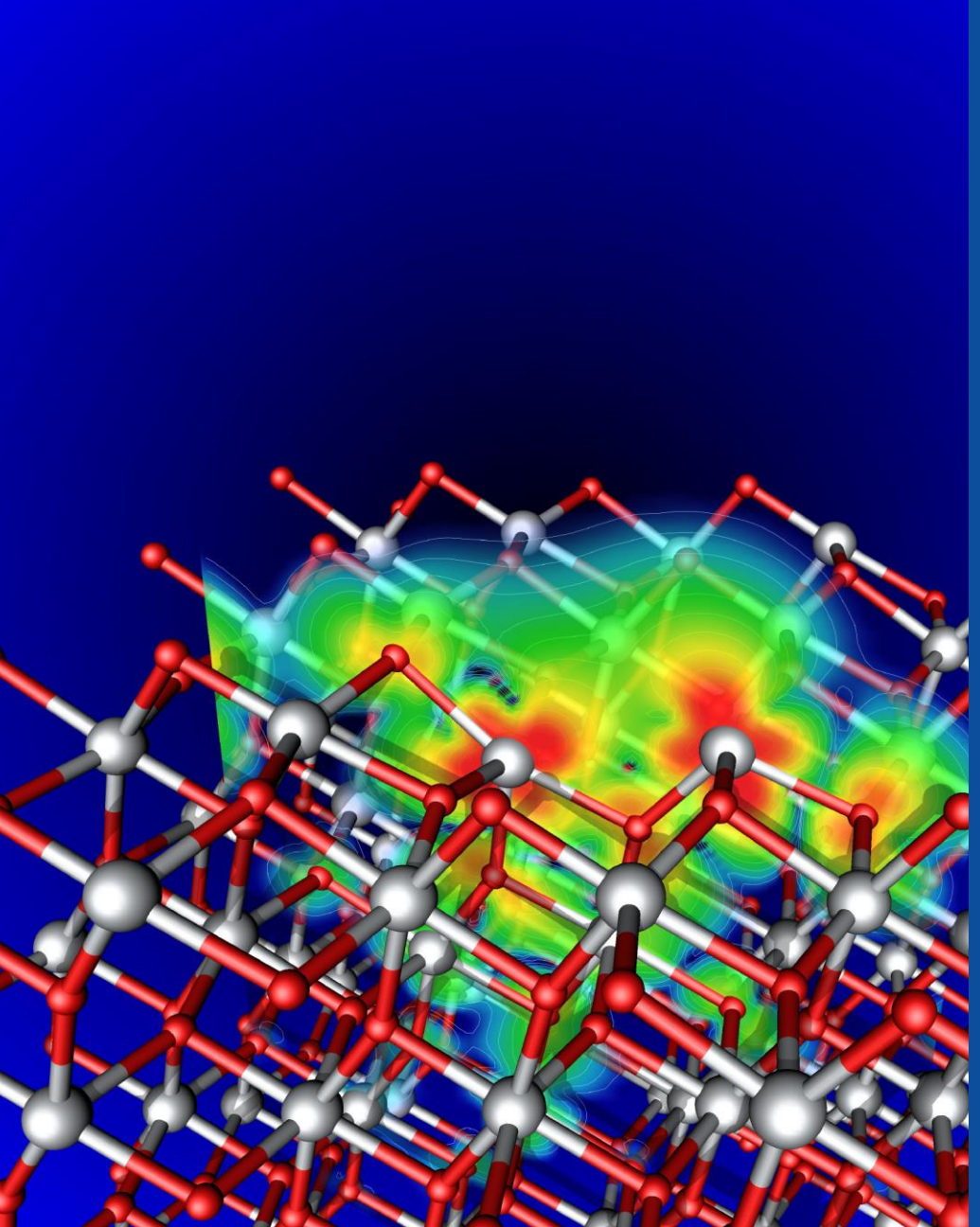
OpenACC HACKATHON
Advancing High Performance Computing

SLACK MEMBERS



160,000+ DOWNLOADS

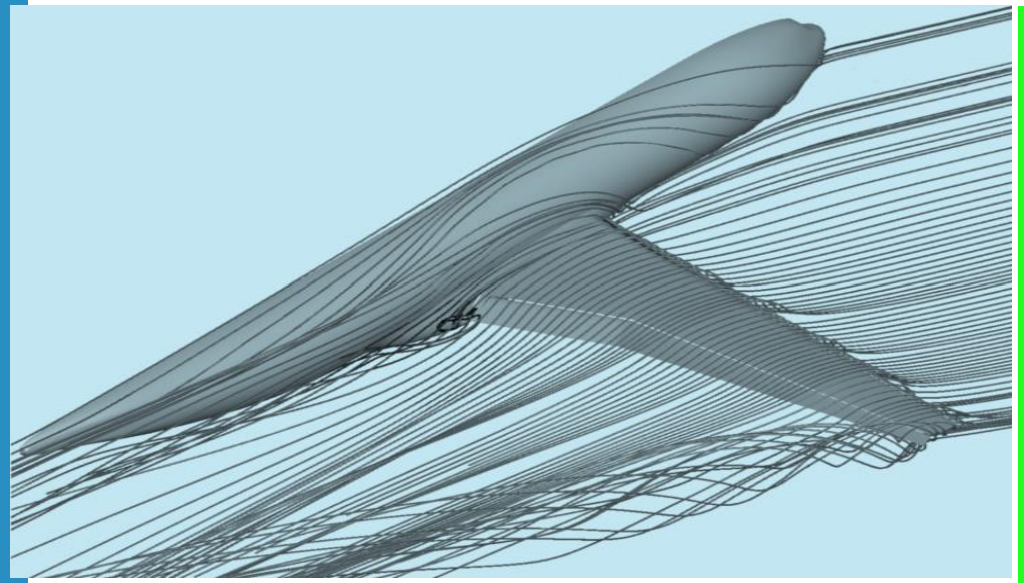
PGI®
Community EDITION

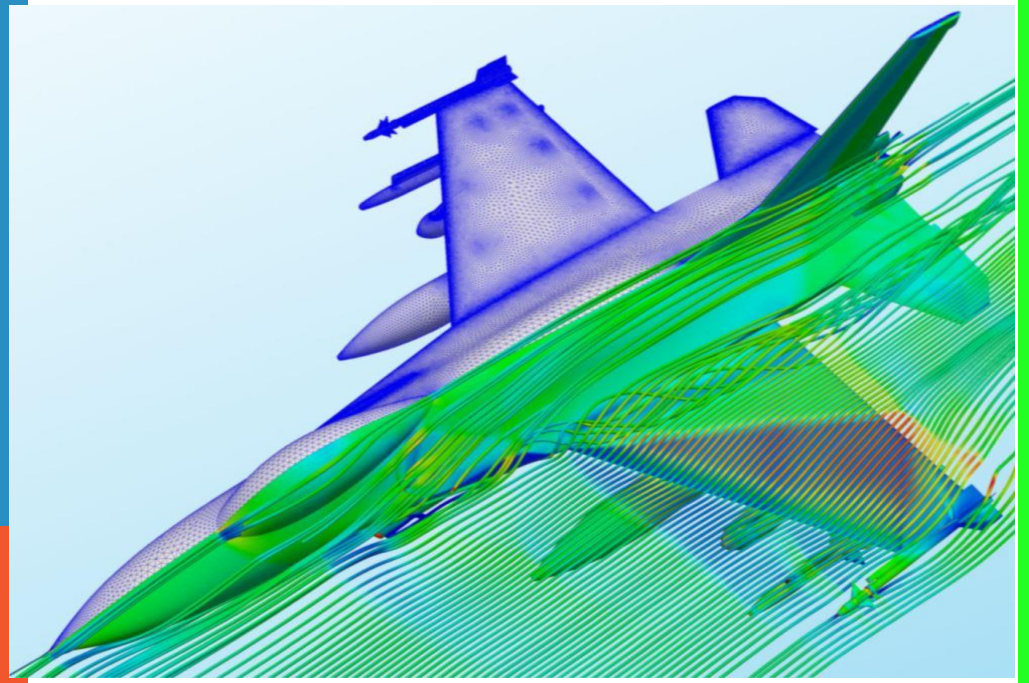


VASP

Prof. Georg Kresse
Computational Materials Physics
University of Vienna

> For VASP, OpenACC is ***the*** way forward for GPU acceleration. Performance is similar and in some cases better than CUDA C, and OpenACC dramatically decreases GPU development and maintenance efforts. We're excited to collaborate with NVIDIA and PGI as an early adopter of CUDA Unified Memory.





HiFUN

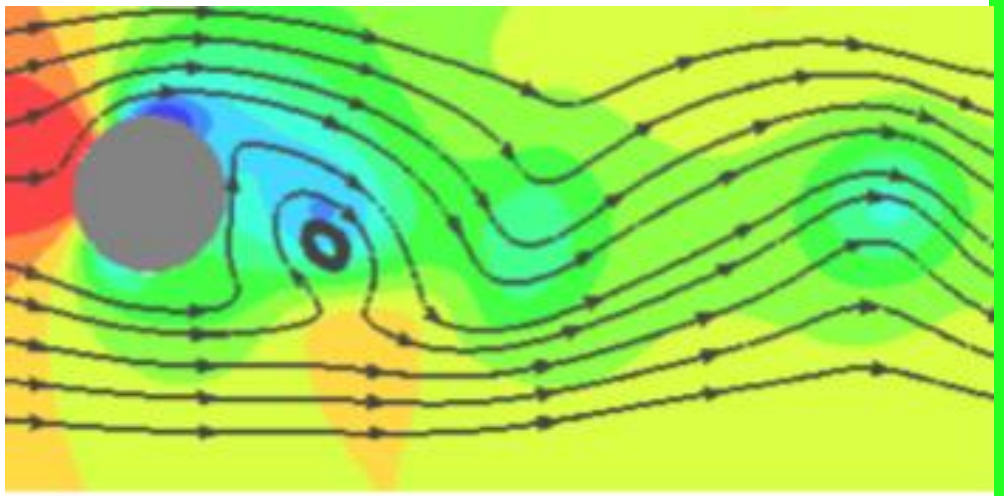
Name: Munikrishna Nagaram

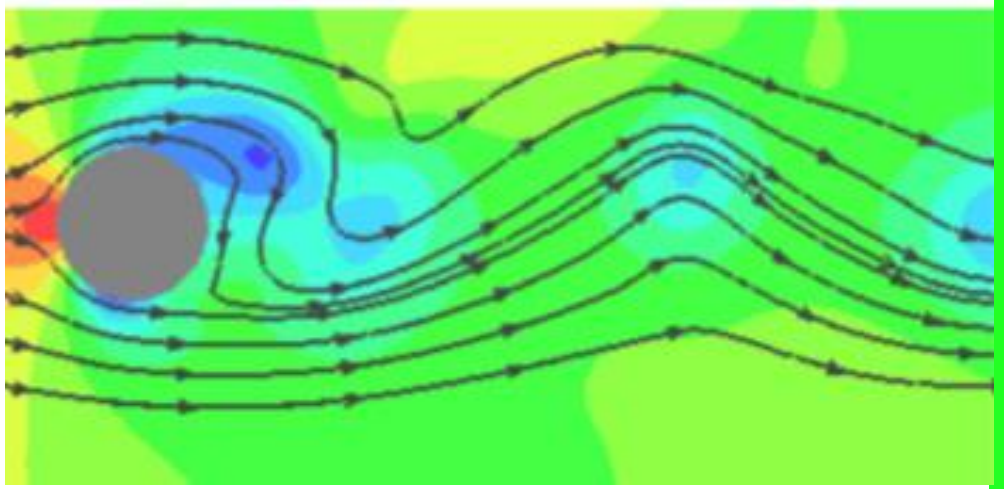
Title: Chief Technology Officer

Company: S & I Engineering Solutions Pvt. Ltd.
Bangalore, India

"OpenACC allowed us to port our legacy CFD code to hybrid CPU+GPU platforms in a form that is readable and maintainable, and enabled us to see a speed-up of 3.0x on our HiFUN MPI solver."

IBM CFD





Name: Somnath Roy

Mechanical Engineering Department
Indian Institute of Technology
Kharagpur

"Using OpenACC to accelerate our immersed boundary incompressible CFD code, we're seeing an order of magnitude reduction in computing time running on GPUs. Routines involving our search algorithm and matrix solvers perform especially well with OpenACC, and improve the overall scalability of the code."

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 instructions the compiler in it 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.

EXAMPLE CODE

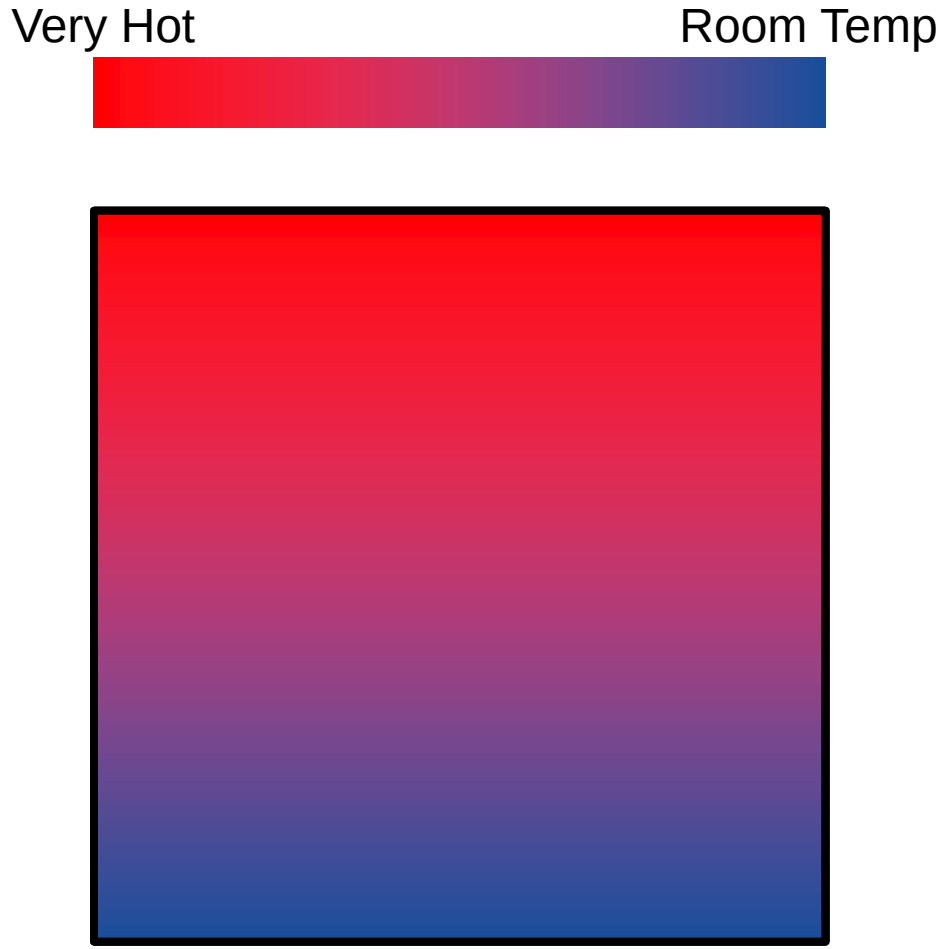
LAPLACE HEAT TRANSFER

Introduction to lab code - visual

We will observe a simple simulation of heat distributing across a metal plate.

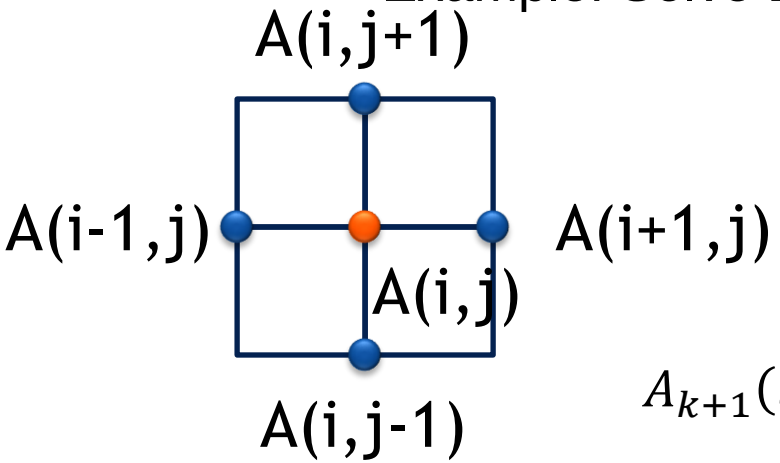
We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.



EXAMPLE: JACOBI ITERATION

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
- Common, useful algorithm
- Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

JACOBI ITERATION: C CODE

```
while ( err > tol && iter < iter_max ) {
  err=0.0;


  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]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }


  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

OpenACC More Science, Less Programming

NVIDIA

OPENACC PARALLEL LOOP DIRECTIVE

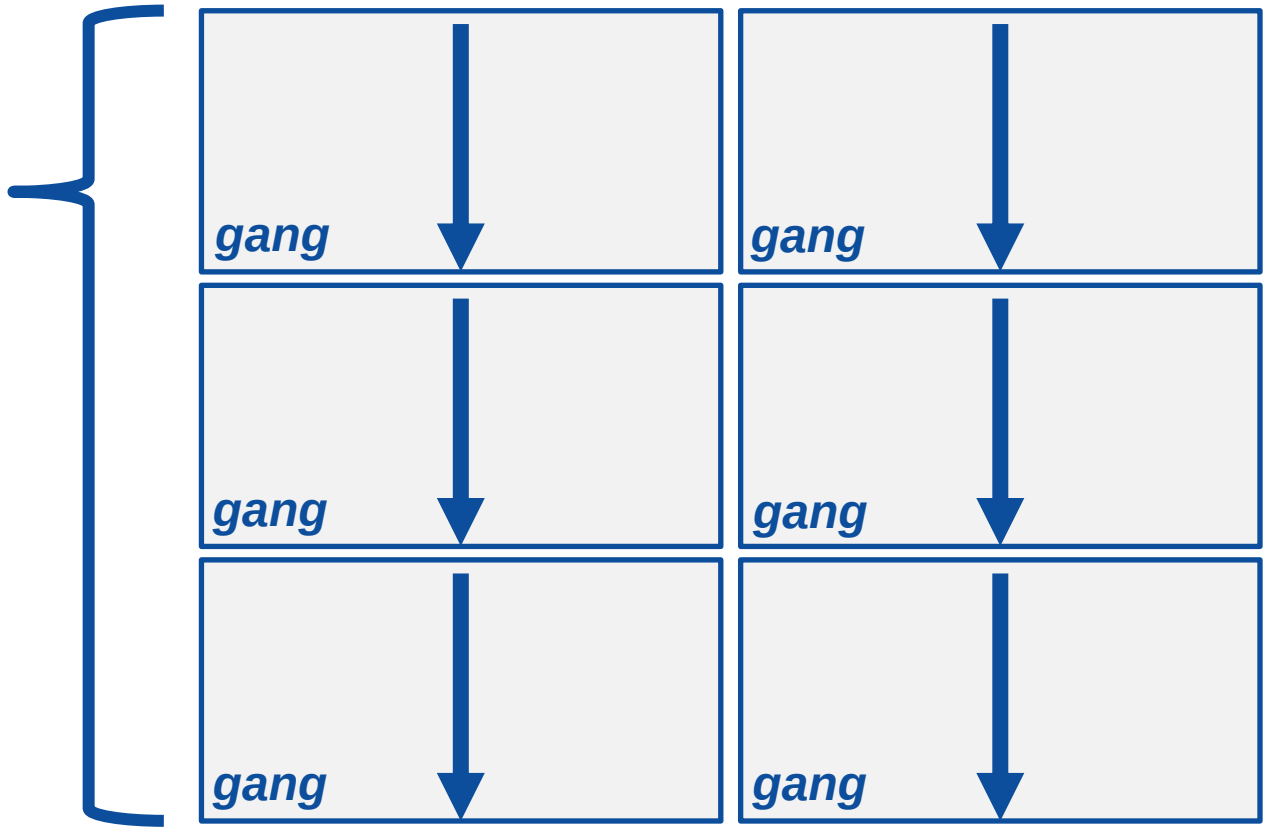
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```
#pragma acc parallel
{
```

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

```
}
```



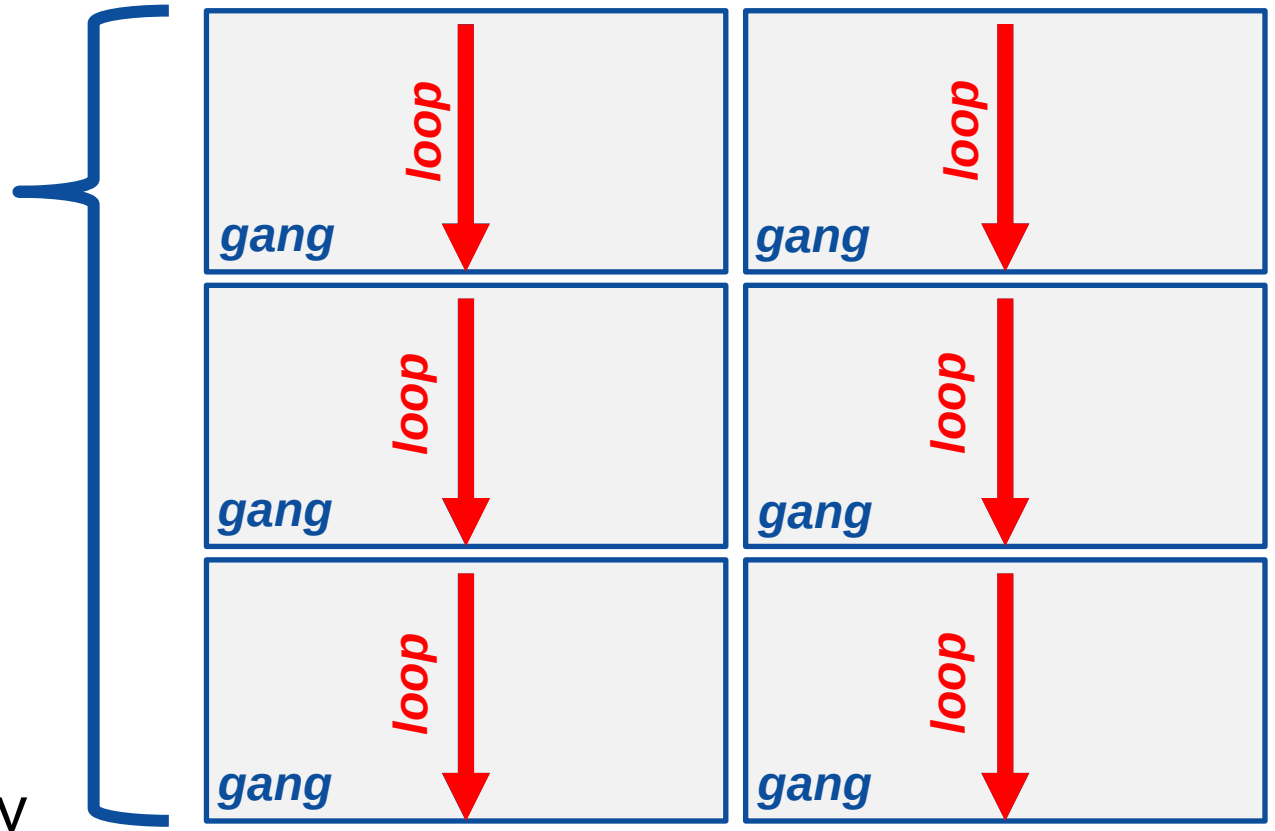
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```
#pragma acc parallel
{

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

}
```

loop

This loop will be executed redundantly on each ***gang***

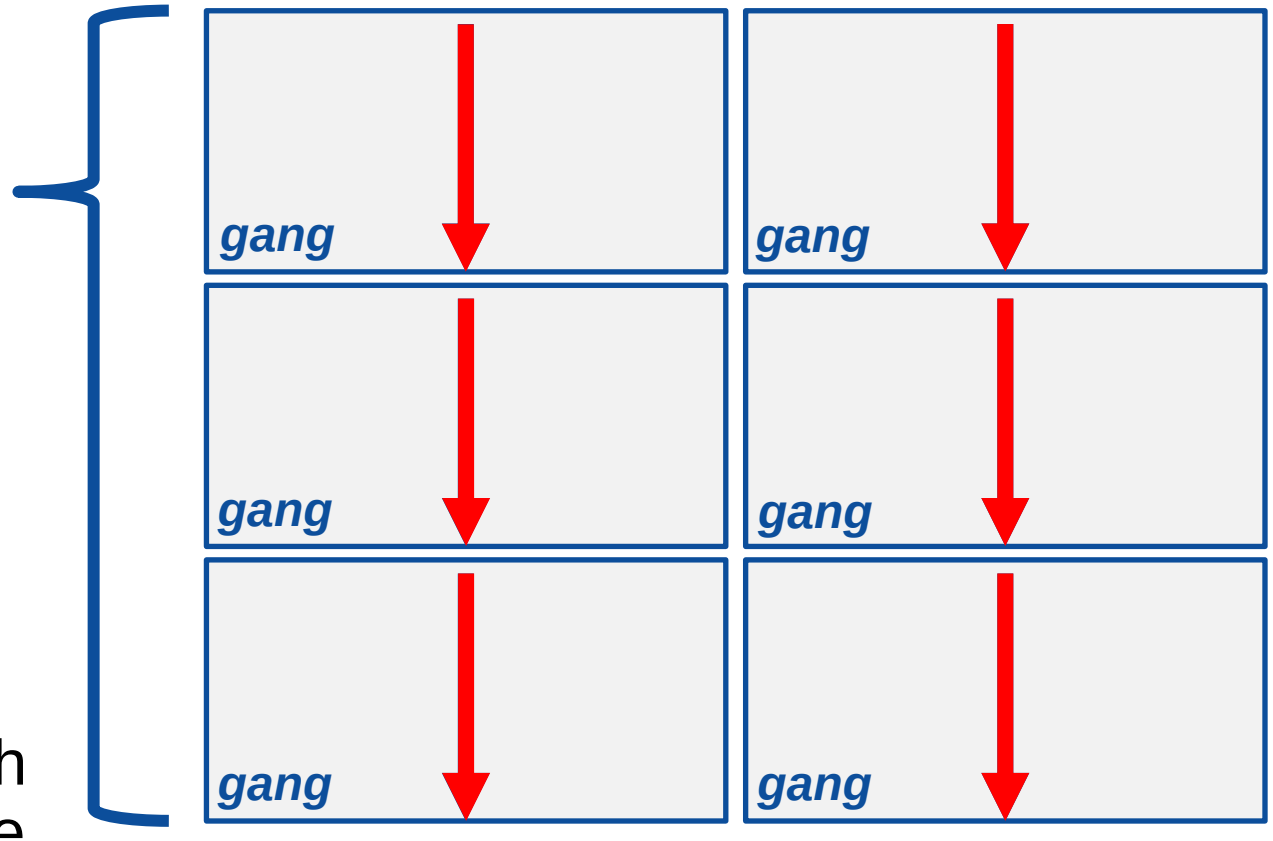


OPENACC PARALLEL DIRECTIVE

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;
}
```

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 LOOP DIRECTIVE

Parallelizing a single loop

C/C++

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

Fortran

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

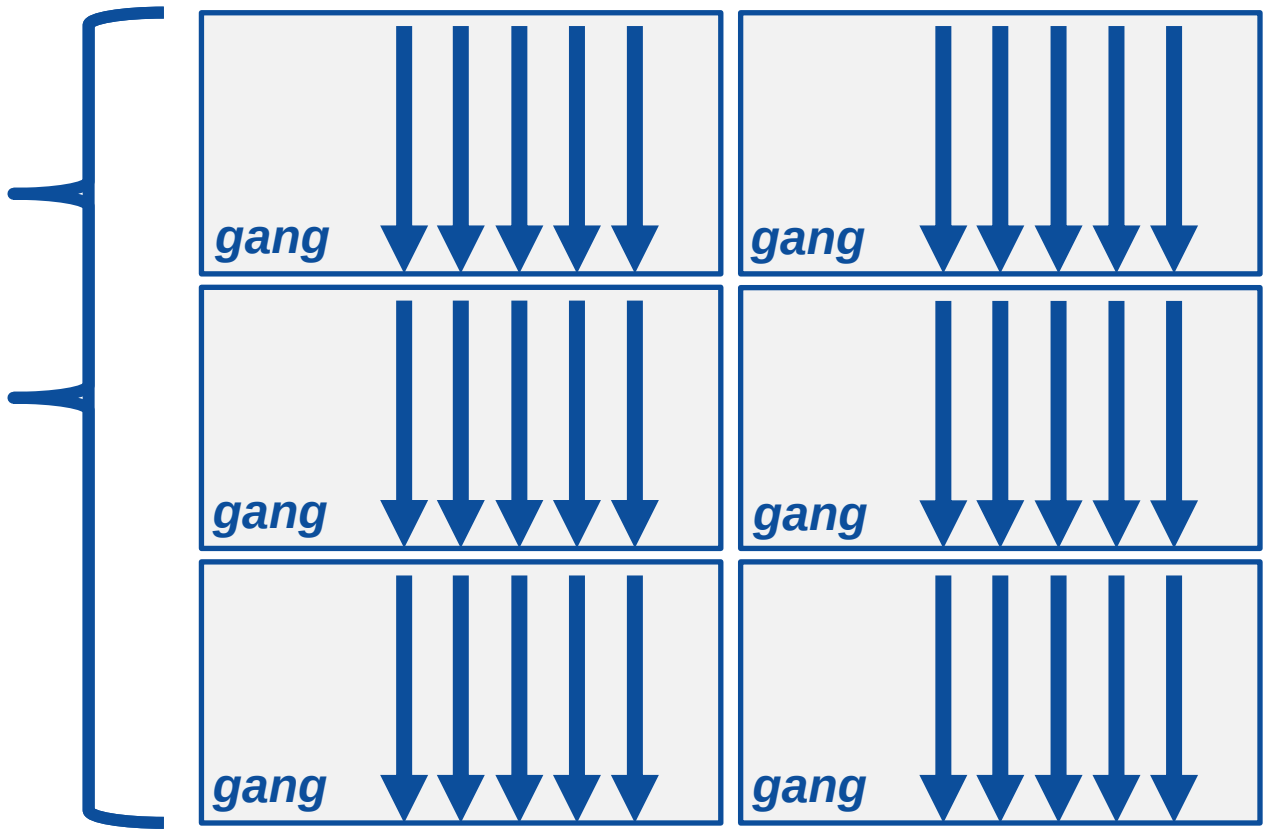
- This pattern is so common that you can do all of this in a single line of code
- In this example, the parallel loop directive applies to the next loop
- This directive both marks the region for parallel execution and distributes the iterations of the loop.
- When applied to a loop with a data dependency,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



PARALLELIZE WITH OPENACC PARALLEL LOOP

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err)
  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]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Parallelize first loop nest, max *reduction* required.

Parallelize second loop.

We didn't detail *how* to parallelize the loops, just *which* loops to parallelize.

OpenACC More Science, Less Programming NVIDIA

REDUCTION CLAUSE

- The **reduction** clause takes many values and "reduces" them to a single value, such as in a sum or maximum
- Each partial result is calculated in parallel
- A **single result** is created by combining the partial results using the specified operation

```
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 acc parallel 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)`

BUILD AND RUN THE CODE

PGI COMPILER BASICS

pgcc, pgc++ and pgfortran

- The command to compile C code is ‘pgcc’
- The command to compile C++ code is ‘pgc++’
- The command to compile Fortran code is ‘pgfortran’
- The -fast flag instructs the compiler to optimize the code to the best of its abilities

```
$ pgcc –fast main.c
$ pgc++ -fast main.cpp
$ pgfortran –fast main.F90
```

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
$ pgfortran –fast –Minfo=all main.f90
```

PGI COMPILER BASICS

-ta flag

- The -ta flag enables building OpenACC code for a “Target Accelerator” (TA)
- -ta=multicore – Build the code to run across threads on a multicore CPU
- -ta=tesla:managed – Build the code for an NVIDIA (Tesla) GPU and manage the data movement automatically (more next module)

```
$ pgcc –fast –Minfo=accel –ta=tesla:managed main.c
$ pgc++ -fast -Minfo=accel –ta=tesla:managed main.cpp
$ pgfortran –fast –Minfo=accel –ta=tesla:managed main.f90
```

PGI COMPILER BASICS

-Mcuda flag

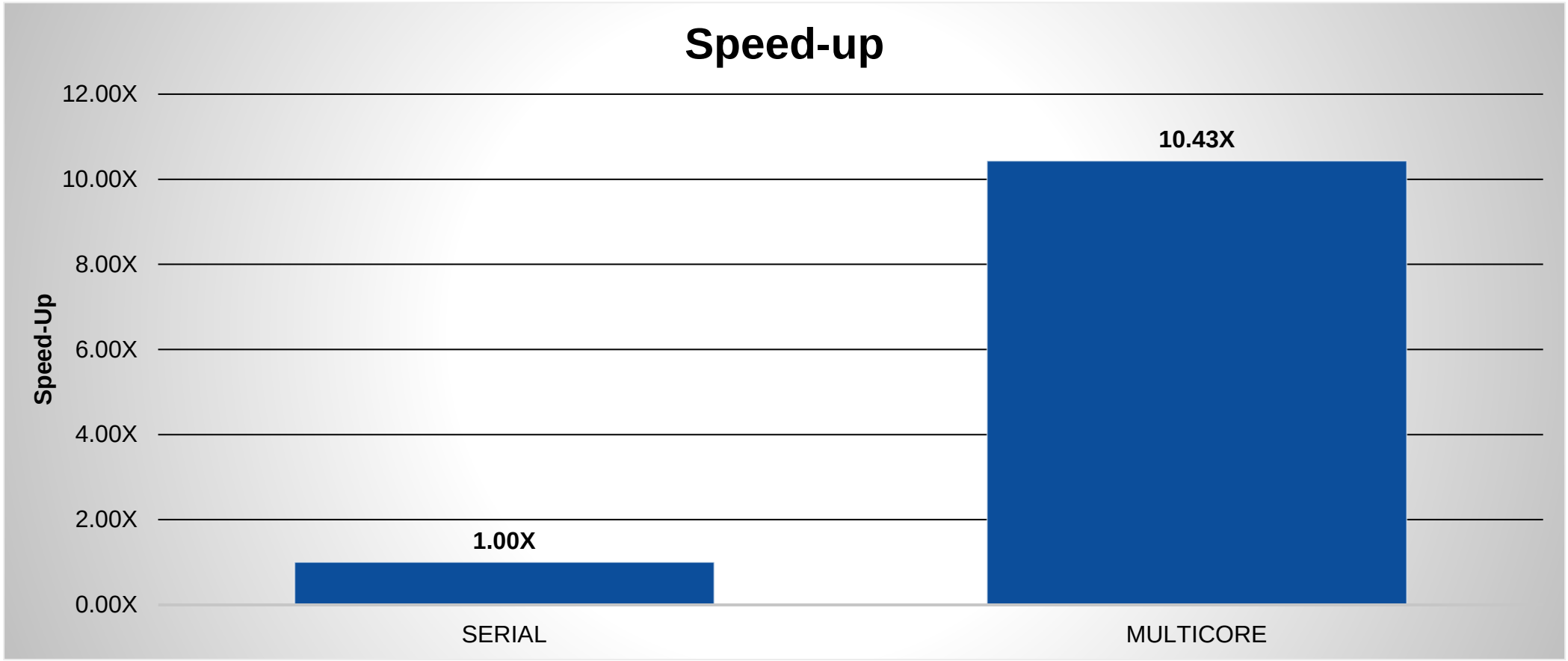
- The -Mcuda flag is needed when using NVTX regions in our code
- -lnvToolsExt – link the NVTX API
- This allows us to use NVTX regions in our code for both CPU and GPU profiling

```
$ pgcc –fast –Minfo=accel –ta=tesla:managed –Mcuda –lnvToolsExt main.c
$ pgc++ -fast -Minfo=accel –ta=tesla:managed –Mcuda –lnvToolsExt main.cpp
$ pgfortran –fast –Minfo=accel –ta=tesla:managed –Mcuda –lnvToolsExt main.f90
```

BUILDING THE CODE (MULTICORE)

```
$ pgcc -fast -ta=multicore -Minfo=accel -Mcuda -lnvToolsExt laplace2d_uvm.c
main:
     63, Generating Multicore code
         64, #pragma acc loop gang
     64, Accelerator restriction: size of the GPU copy of Anew,A is unknown
         Generating reduction(max:error)
     66, Loop is parallelizable
     74, Generating Multicore code
         75, #pragma acc loop gang
     75, Accelerator restriction: size of the GPU copy of Anew,A is unknown
     77, Loop is parallelizable
```

OPENACC SPEED-UP

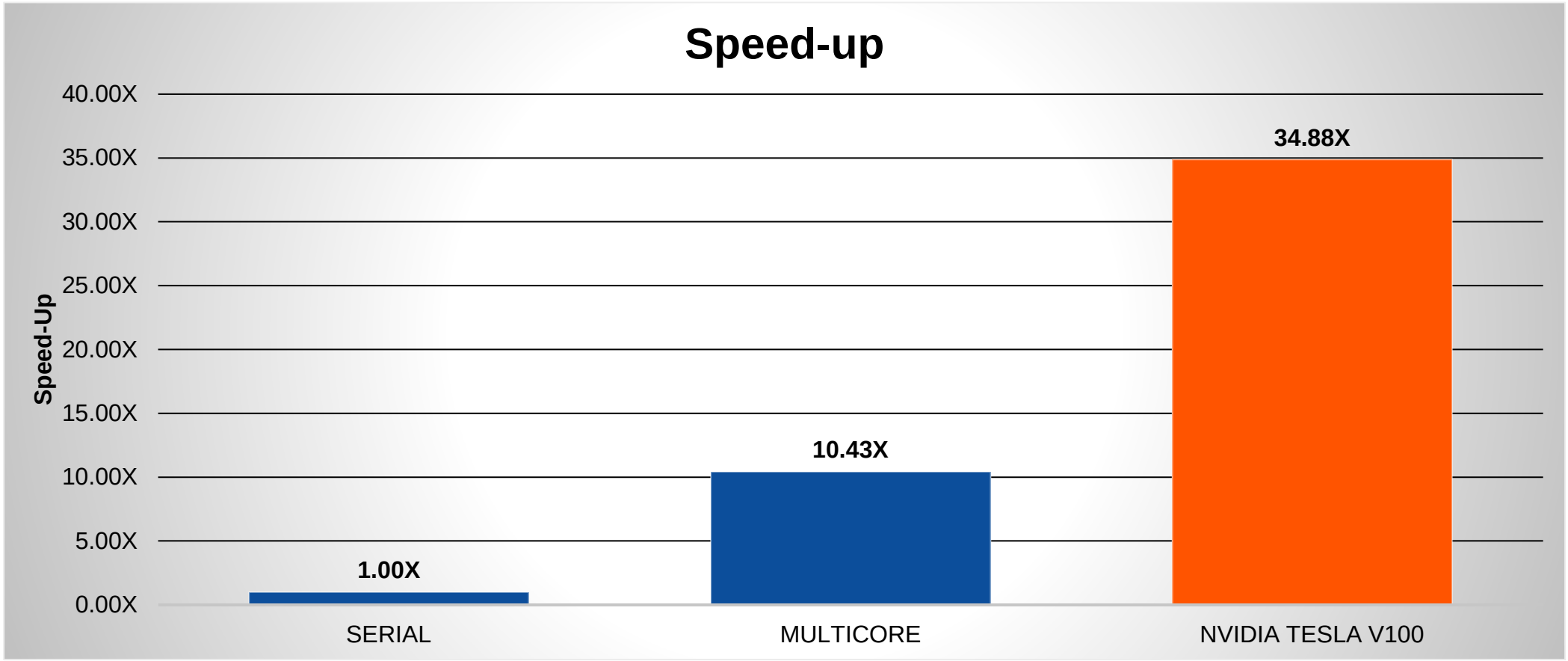


PGI 19.10, NVIDIA Tesla V100, IBM POWER9 22-core CPU @ 3.07GHz

BUILDING THE CODE (GPU)

```
$ pgcc -fast -ta=tesla:managed -Minfo=accel -Mcuda -lnvToolsExt laplace2d_uvm.c
main:
     63, Accelerator kernel generated
         Generating Tesla code
         64, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:error)
         66, #pragma acc loop vector(128) /* threadIdx.x */
     63, Generating implicit copyin(A[:])
         Generating implicit copyout(Anew[:])
         Generating implicit copy(error)
     66, Loop is parallelizable
     74, Accelerator kernel generated
         Generating Tesla code
         75, #pragma acc loop gang /* blockIdx.x */
         77, #pragma acc loop vector(128) /* threadIdx.x */
     74, Generating implicit copyin(Anew[:])
         Generating implicit copyout(A[:])
     77, Loop is parallelizable
```

OPENACC SPEED-UP



OpenACC More Science, Less Programming

NVIDIA

PGI 19.10, NVIDIA Tesla V100, IBM POWER9 22-core CPU @ 3.07GHz