# INTRODUCTION TO OPENACC

Steve Abbott, OLCF Intro to HPC, June 2018

OpenACC
More Science, Less Programming

# OUTLINE

## Topics to be covered

- What is OpenACC
- Profile-driven Development
- OpenACC Fundamentals
- OpenACC Data Directives
- OpenACC Loop Optimizations
- Where to Get Help

# ABOUT THIS SESSION

- The objective of this session is to give you a brief introduction of OpenACC programming for NVIDIA GPUs
- There will be a hands on session mixed in where you get to try this out, and it will lead us into profiling tools
- Feel free to interrupt with questions

# INTRODUCTION TO OPENACC

**OpenACC** is a directives-based programming approach to **parallel computing** 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

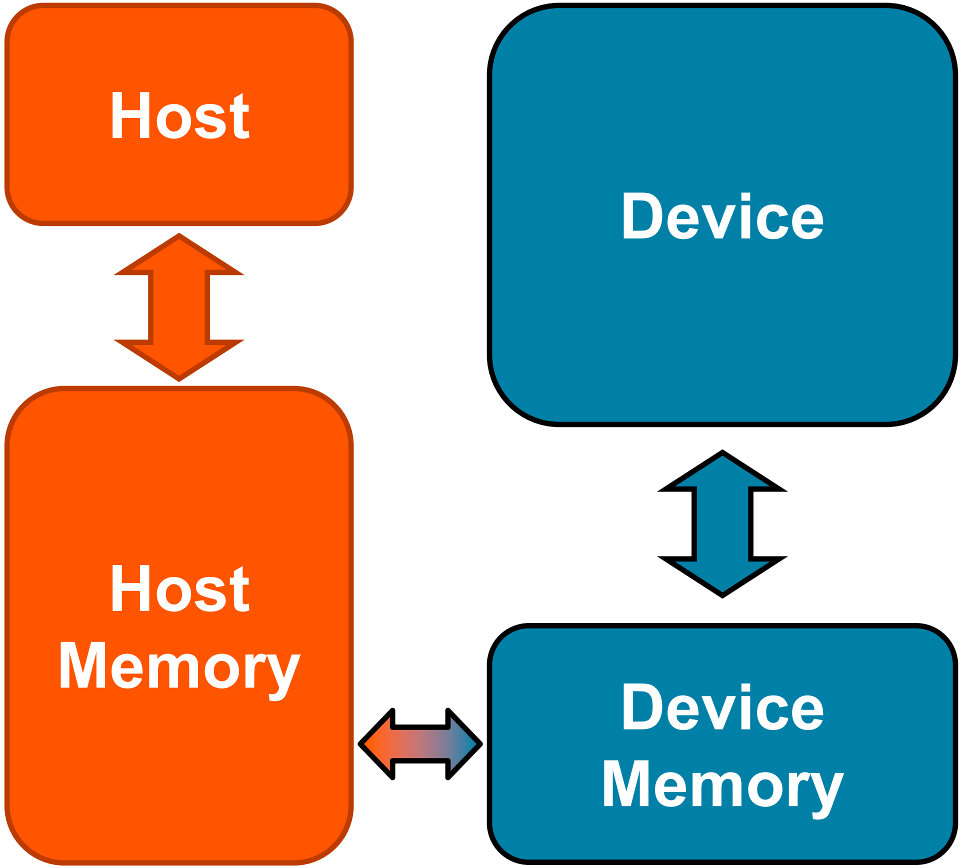
Applications

| Libraries | Compiler Directives | Programming Languages |
| --- | --- | --- |
| **Easy to use**<br>**Most Performance** | **Easy to use**<br>**Portable code**<br>**OpenACC** | **Most Performance**<br>**Most Flexibility** |

# 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

## 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 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

**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

## Incremental

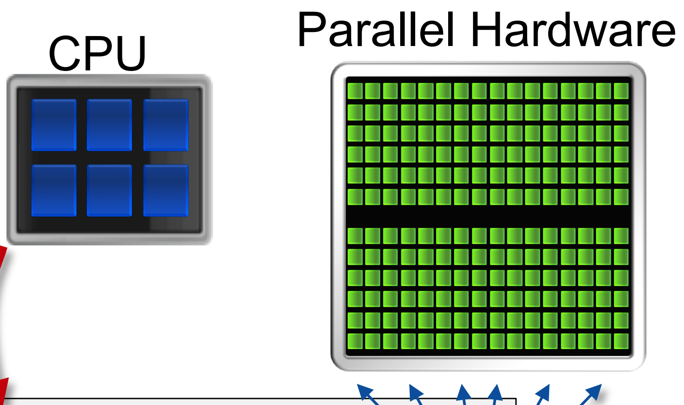
- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

## 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




```
int main(){

 <sequential code>

 #pragma acc kernels
 {
  <parallel code>
 }

}
```

**Compiler Hint**

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

## Incremental

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

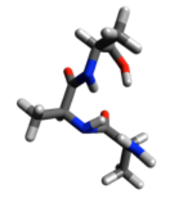
## Single Source

- 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.

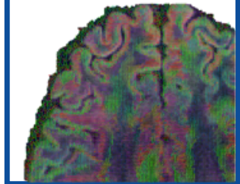
# OPENACC SUCCESSES



**LSDalton**

Quantum Chemistry
Aarhus University

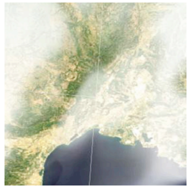
12X speedup
1 week



**PowerGrid**

Medical Imaging
University of Illinois

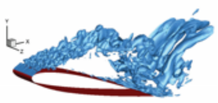
40 days to
2 hours



**COSMO**

Weather and Climate
MeteoSwiss, CSCS

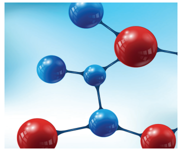
40X speedup
3X energy efficiency



**INCOMP3D**

CFD
NC State University

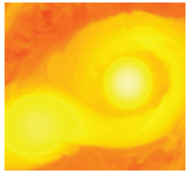
4X speedup



**NekCEM**

Comp Electromagnetics
Argonne National Lab

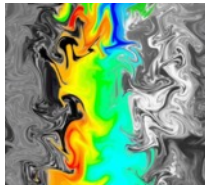
2.5X speedup
60% less energy



**MAESTRO**
**CASTRO**

Astrophysics
Stony Brook University

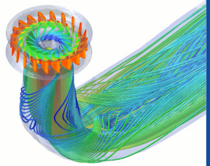
4.4X speedup
4 weeks effort



**CloverLeaf**

Comp Hydrodynamics
AWE

4X speedup
Single CPU/GPU code



FINE/Turbo

CFD
NUMECA
International

10X faster routines
2X faster app

# 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

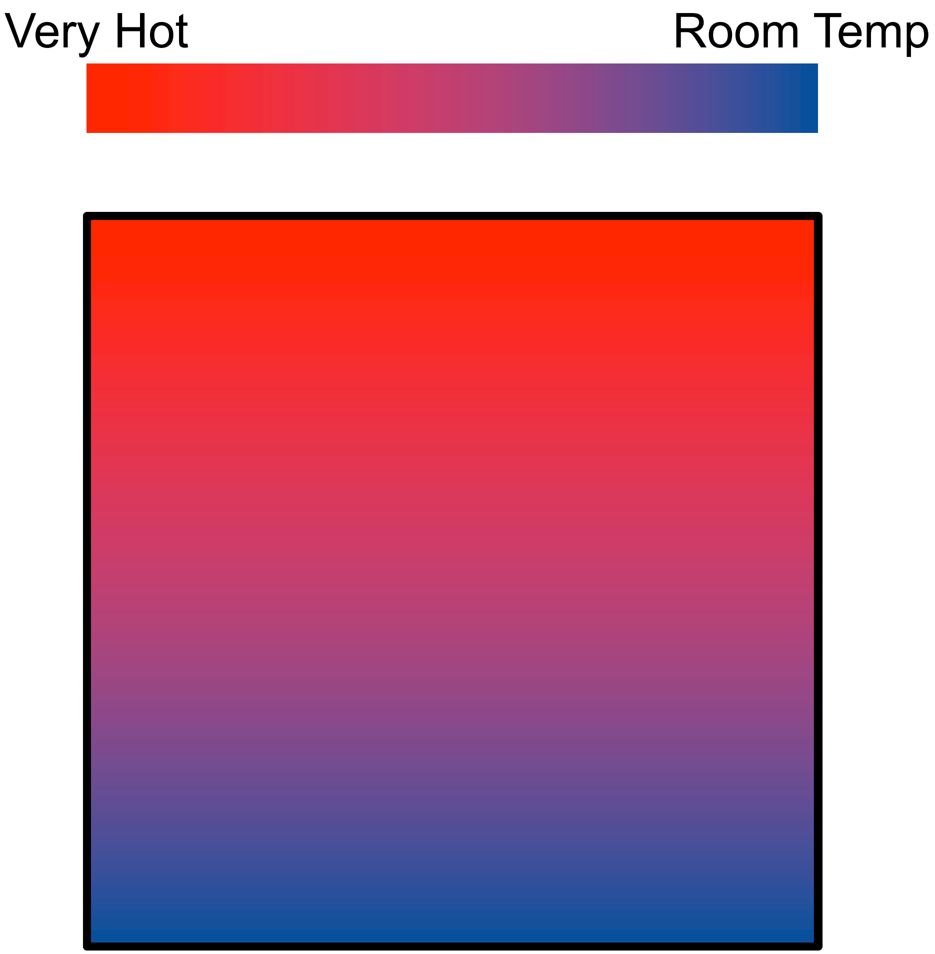
# LAPLACE HEAT TRANSFER

## Introduction to lab code - visual

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

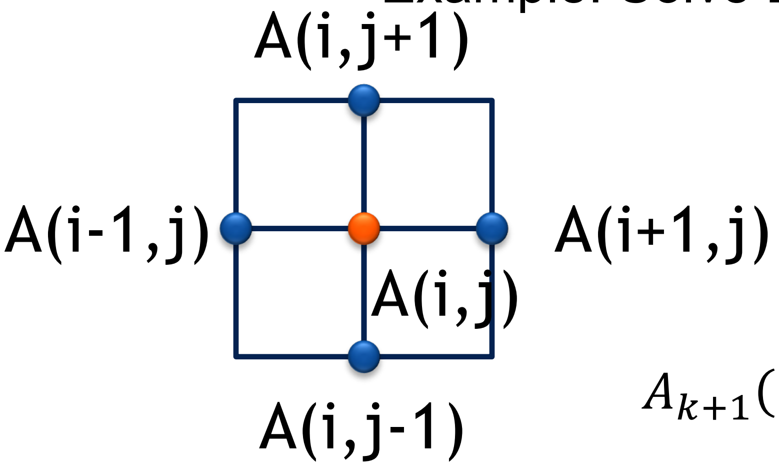
We will apply a consistent heat to the top of the plate.

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

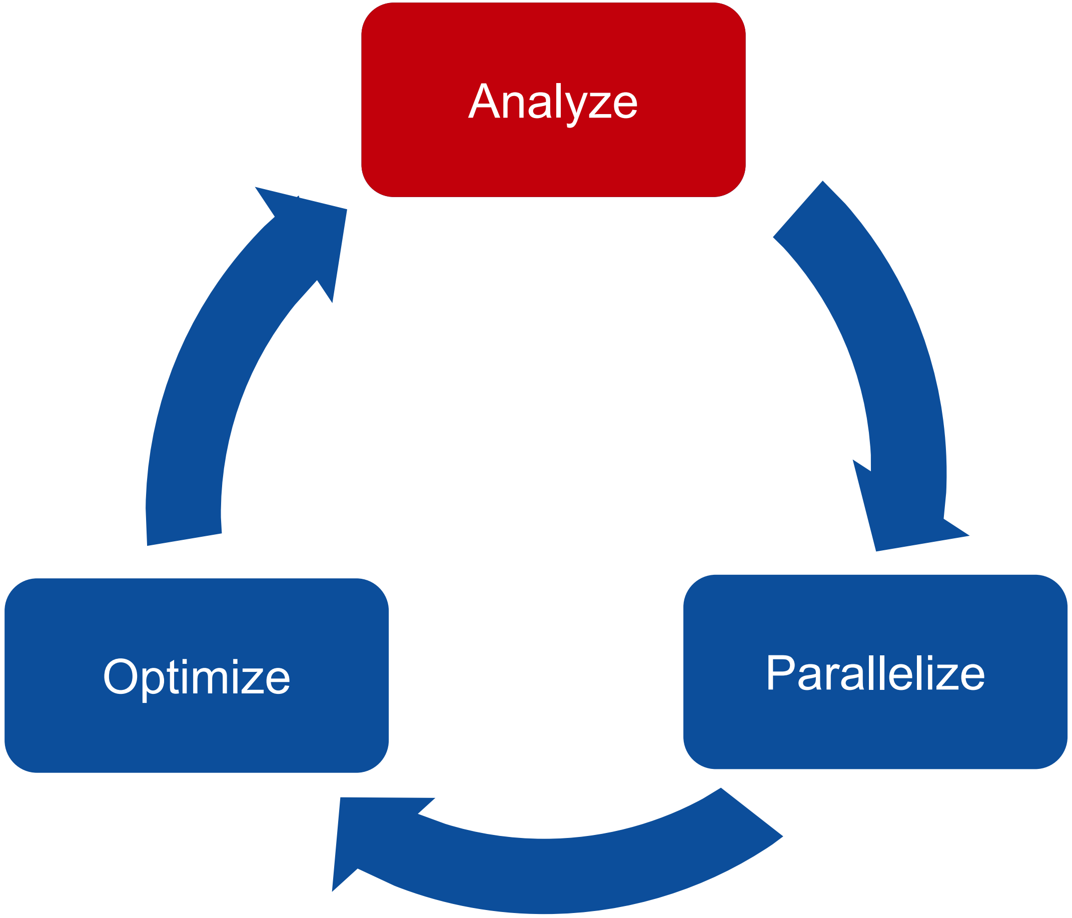
Compute max error for convergence

Swap input/output arrays

# PROFILE-DRIVEN DEVELOPMENT

# 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 and check for correctness.
- **Optimize** your code to improve observed speed-up from parallelization.



# PROFILING SEQUENTIAL CODE

## Profile Your Code

Obtain detailed information about how the code ran.

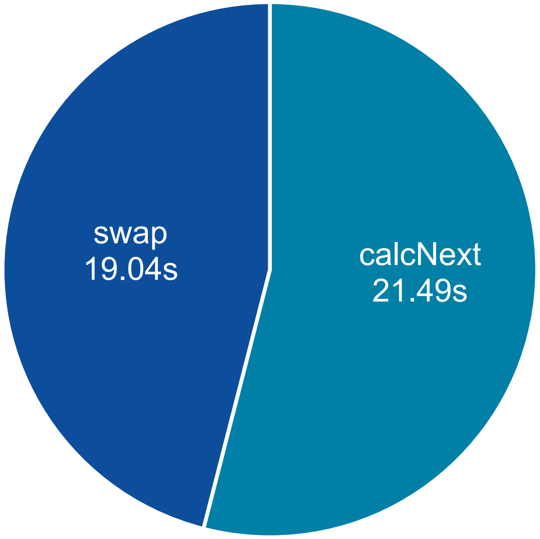
This can include information such as:

- 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



# OPENACC PARALLEL DIRECTIVE

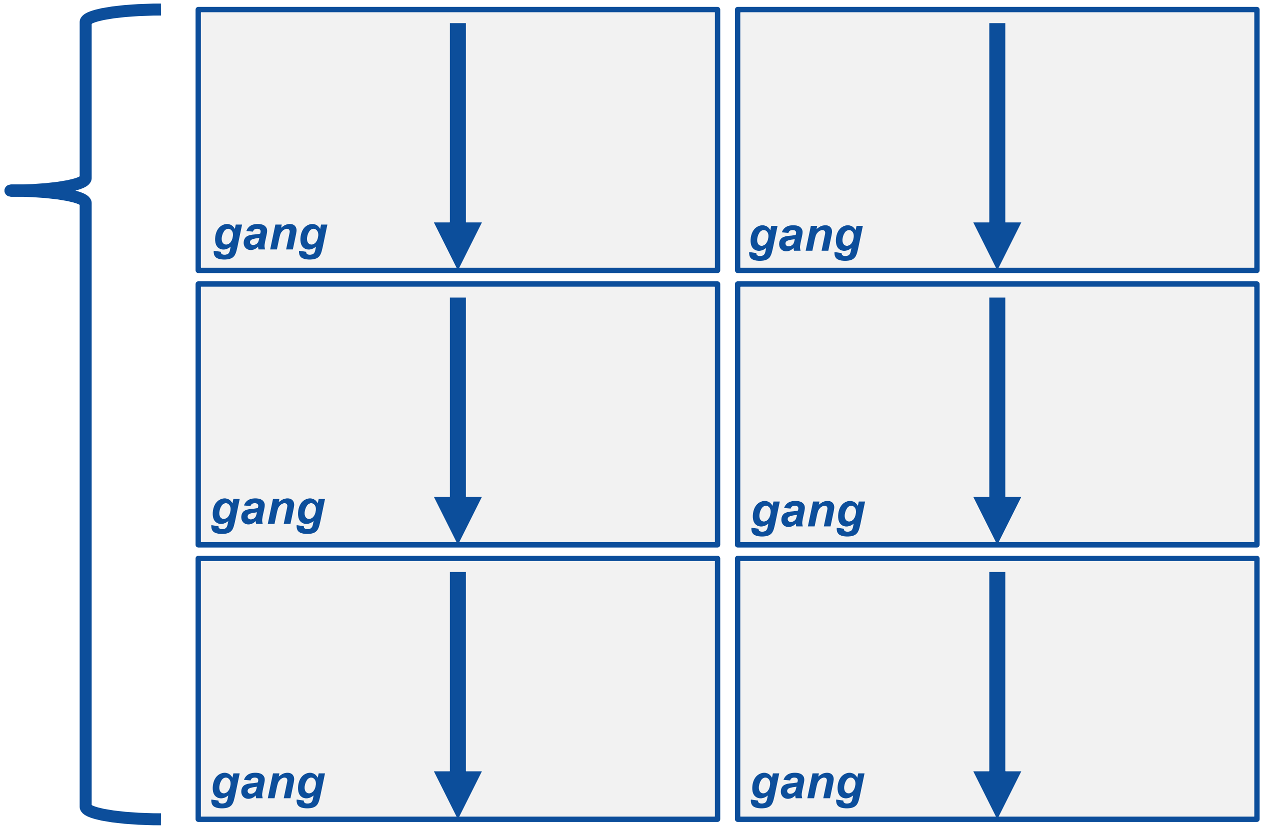
# OPENACC PARALLEL DIRECTIVE

## Expressing parallelism

```
#pragma acc parallel
{
```

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

```
}
```



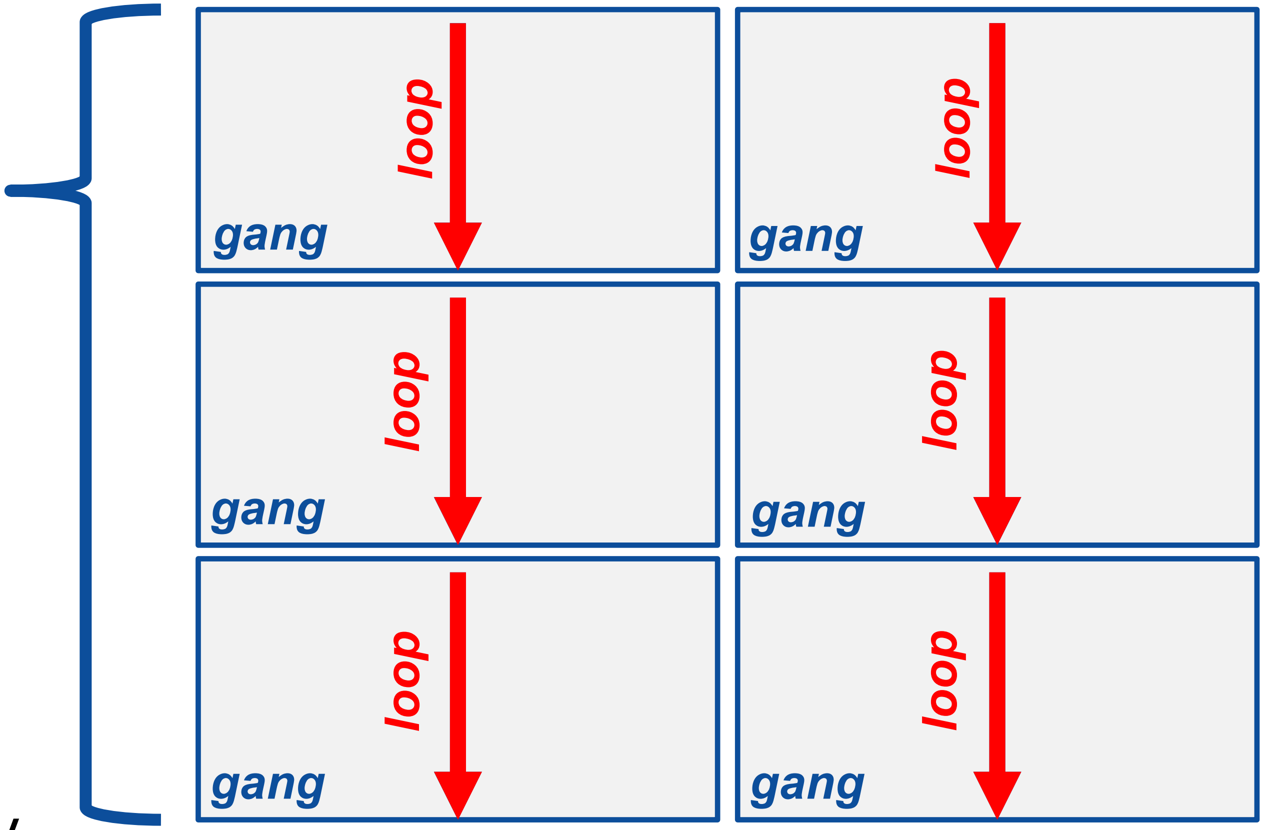
# OPENACC PARALLEL DIRECTIVE

## Expressing parallelism

```
#pragma acc parallel
{

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

}
```




This loop will be executed redundantly on each gang

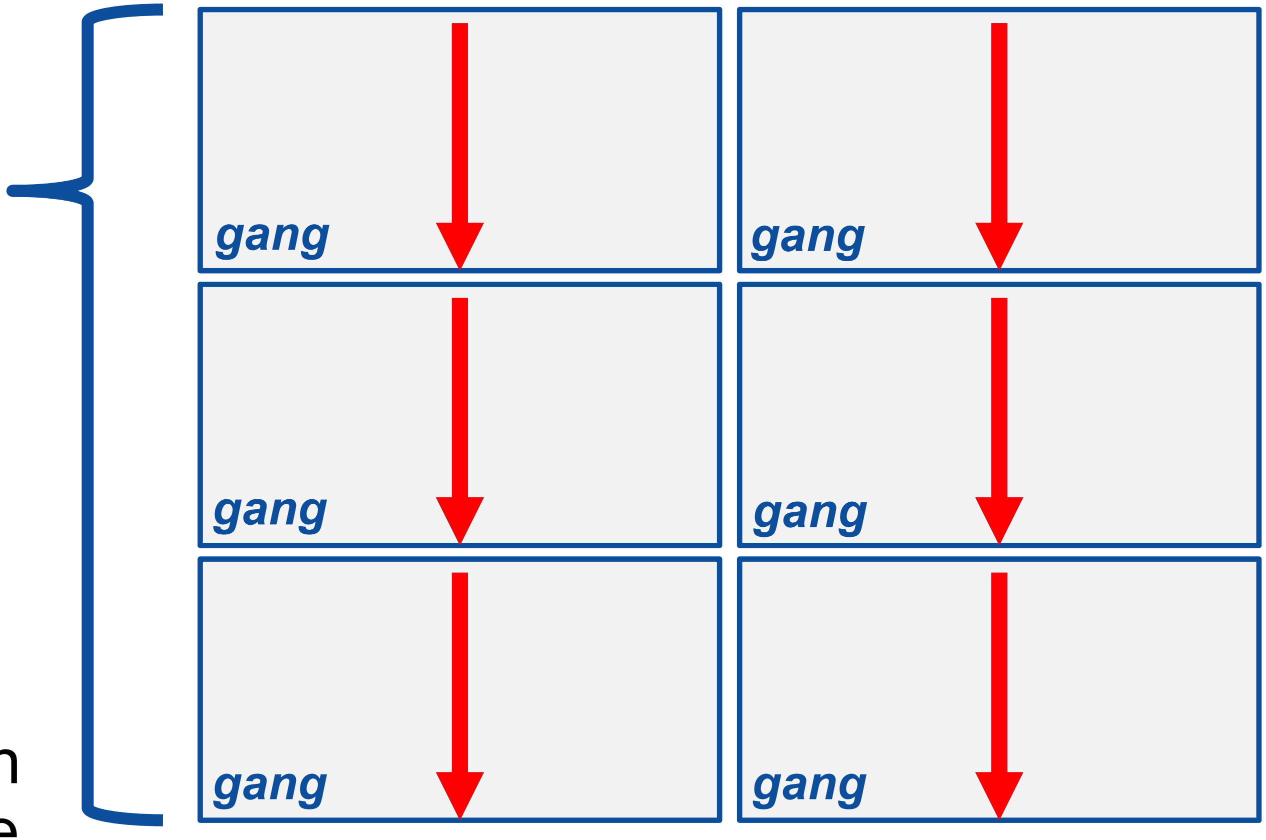
# OPENACC PARALLEL DIRECTIVE

## 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; 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 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

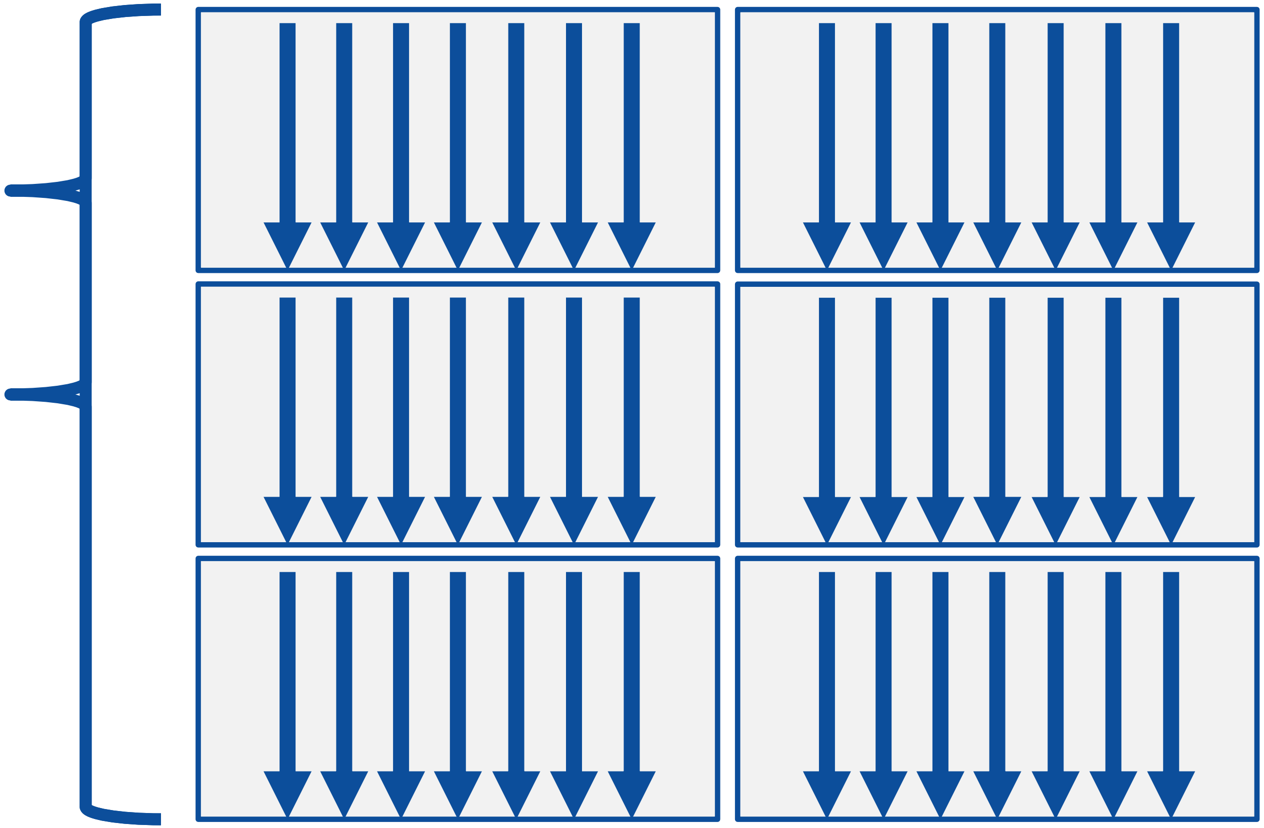
## Expressing parallelism

```
#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.



OpenACC More Science, Less Programming NVIDIA DEEP LEARNING INSTITUTE

# OPENACC PARALLEL LOOP 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

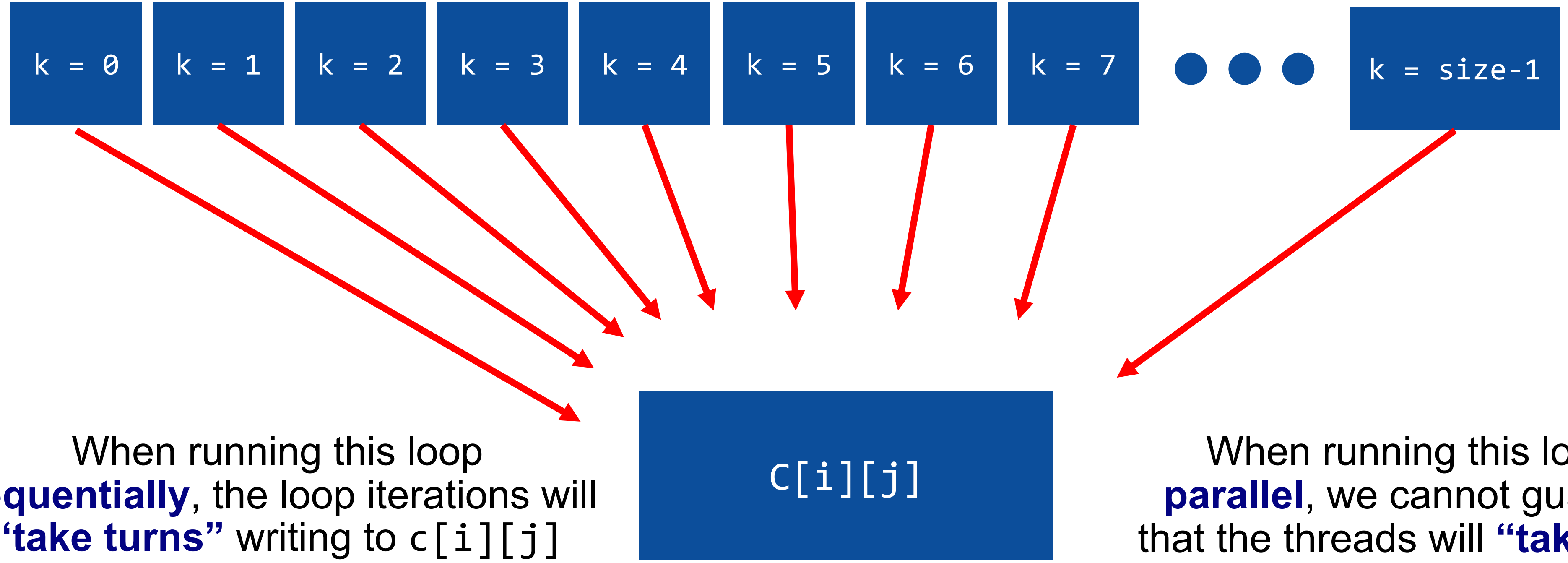
# REDUCTION CLAUSE

- 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**

```
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];
```

# WITHOUT A REDUCTION

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




When running this loop **sequentially**, the loop iterations will **"take turns"** writing to `c[i][j]`

When running this loop in **parallel**, we cannot guarantee that the threads will **"take turns"**

# 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)` |

# 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.

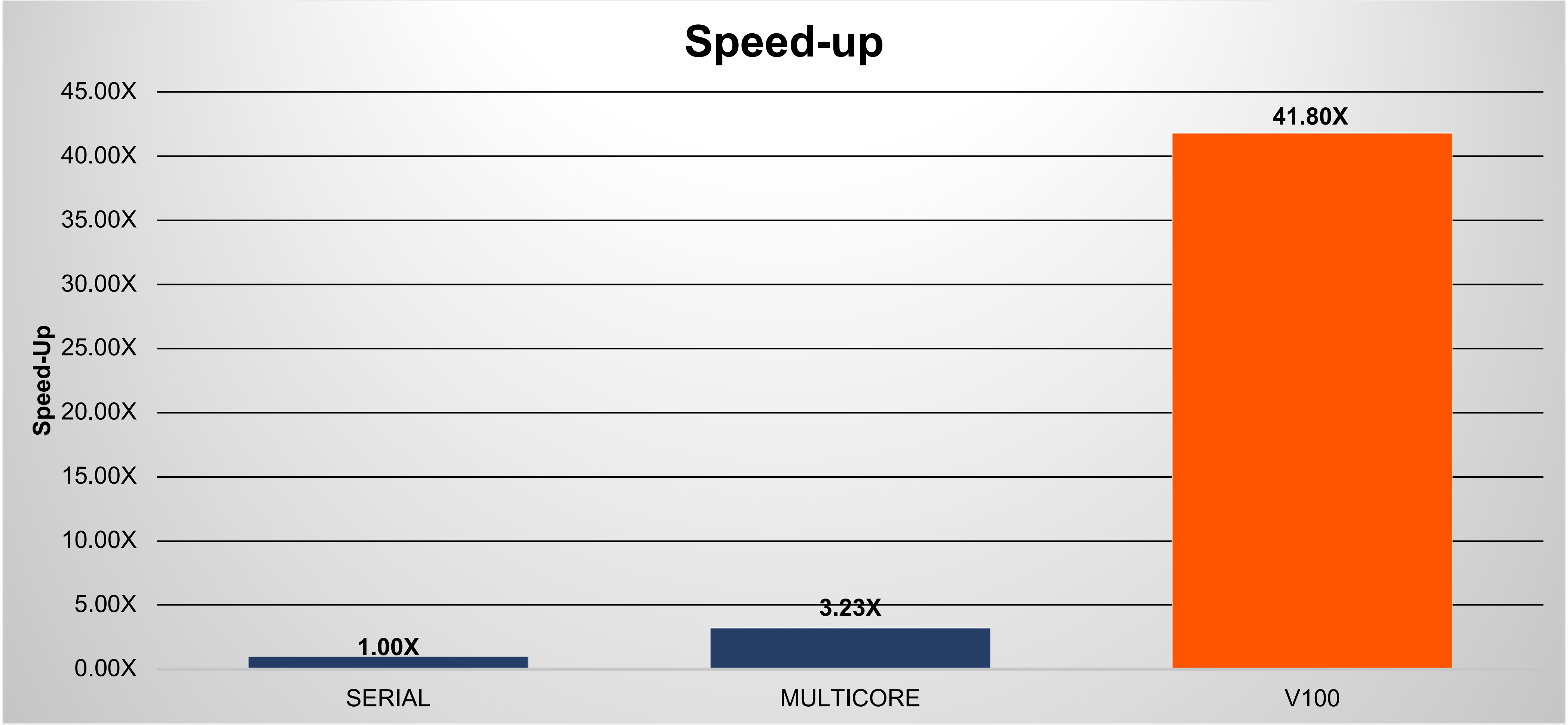
# BUILDING THE CODE (GPU)

```
$ pgcc -fast -ta=tesla -Minfo=accel 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
```

# BUILDING THE CODE (MULTICORE)

```
$ pgcc -fast -ta=multicore -Minfo=accel 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



# TRY IT OUT!

- Grab the OpenACC tests code
- Put it somewhere compute accessible ($MEMBERWORK/trn001 might be a good start)
- Start an interactive batch job on Titan, and make sure can build and run the code
    - Use the PGI compiler (TIP: module load PrgEnv-pgi)
    - Build (TIP: make)
    - Run and verify the GPU code works (TIP: aprun –n1 ./openacc_demo_c)

# EXPECTED CPU OUPUT

```
OpenACC_tests> aprun -n 1 ./openacc_demo_c
Initialize check:
A[   0] = 0 (0) B[0] = 0 (0)
A[100] = 100 (100) B[100] = 200 (200)
A[1623] = 1623 (1623) B[1623] = 3246 (3246)
A[111111] = 111111 (111111) B[111111] = 222222 (222222)
saxpy check:
C[   0] = 0 (0)
C[100] = 400 (400)
C[1623] = 6492 (6492)
C[111111] = 444444 (444444)
sumC check: 2e+14 (2e+14)
Total time: 0.2237s
Init time: 0.0949s
SAXPY time: 0.0547s
SumC time: 0.0739s
Application 17826144 resources: utime ~0s, stime ~0s, Rss ~121276, inblocks ~98, outblocks
```

# TRY IT OUT!

- Use the following OpenACC directives to parallelize the code

  [#pragma/!] acc parallel loop [reduction(+:)]

- Edit the makefile and add TO COMPILE: -acc –ta=tesla
- Run with: aprun –n1 ./openacc_demo_c

(or checkout the git branch “Stage1”, but try it!)

# PAY ATTENTION TO THE COMPILER!

```
cc -Minfo=all -acc -ta=tesla -o openacc_demo_c openacc_demo.c
main:
     25, Accelerator kernel generated
         Generating Tesla code
         26, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
     25, Generating implicit copyout(A[:10000000],B[:10000000])
     46, Accelerator kernel generated
         Generating Tesla code
         47, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
     46, Generating implicit copyout(C[:10000000])
         Generating implicit copyin(B[:10000000],A[:10000000])
     67, Accelerator kernel generated
         Generating Tesla code
         67, Generating reduction(+:sumC)
         68, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
     67, Generating implicit copyin(C[:10000000])
```

# RUN THE GPU VERSION NOW

```
OpenACC_tests> aprun -n 1 ./openacc_demo_c
Initialize check:
A[  0] = 0 (0) B[0] = 0 (0)
A[100] = 100 (100) B[100] = 200 (200)
A[1623] = 1623 (1623) B[1623] = 3246 (3246)
A[111111] = 111111 (111111) B[111111] = 222222 (222222)
saxpy check:
C[  0] = 0 (0)
C[100] = 400 (400)
C[1623] = 6492 (6492)
C[111111] = 444444 (444444)
sumC check: 2e+14 (2e+14)
Total time: 0.5025s
Init time: 0.3881s
SAXPY time: 0.1026s
SumC time: 0.0116s
Application 17826388 resources: utime ~0s, stime ~1s, Rss ~267308,
inblocks ~257, outblocks ~123
```

Why did we get slower?

# PROFILE!

- Running nvprof is a little tricky on Titan
    - Binary must be on a compute accessible file system (lustre)
    - Need some magic flags
- Run: PMI_NO_FORK=1 aprun –n 1 –b nvprof ./openacc_demo_c

Titan magic to make nvprof/cuda-memcheck/cuda-gdb work!
Not needed on most other machines.

# RUN THE GPU VERSION NOW

```
==1467== Profiling application: ./openacc_demo_c
==1467== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.98%  20.031ms        10  2.0031ms  1.7280us  2.8053ms  [CUDA memcpy HtoD]
                   44.66%  17.901ms        10  1.7901ms  2.2080us  2.5109ms  [CUDA memcpy DtoH]
                    2.12%  848.38us         1  848.38us  848.38us  848.38us  main_67_gpu
                    1.80%  721.88us         1  721.88us  721.88us  721.88us  main_46_gpu
                    1.05%  420.03us         1  420.03us  420.03us  420.03us  main_25_gpu
                    0.39%  157.95us         1  157.95us  157.95us  157.95us  main_67_gpu_red
      API calls:   58.84%  287.27ms         2  143.64ms     860ns  287.27ms  cuDevicePrimaryCtxRetain
                   20.99%  102.46ms         1  102.46ms  102.46ms  102.46ms  cuDevicePrimaryCtxRelease
                    9.27%  45.268ms         1  45.268ms  45.268ms  45.268ms  cuMemHostAlloc
                    4.60%  22.457ms         1  22.457ms  22.457ms  22.457ms  cuMemFreeHost
```

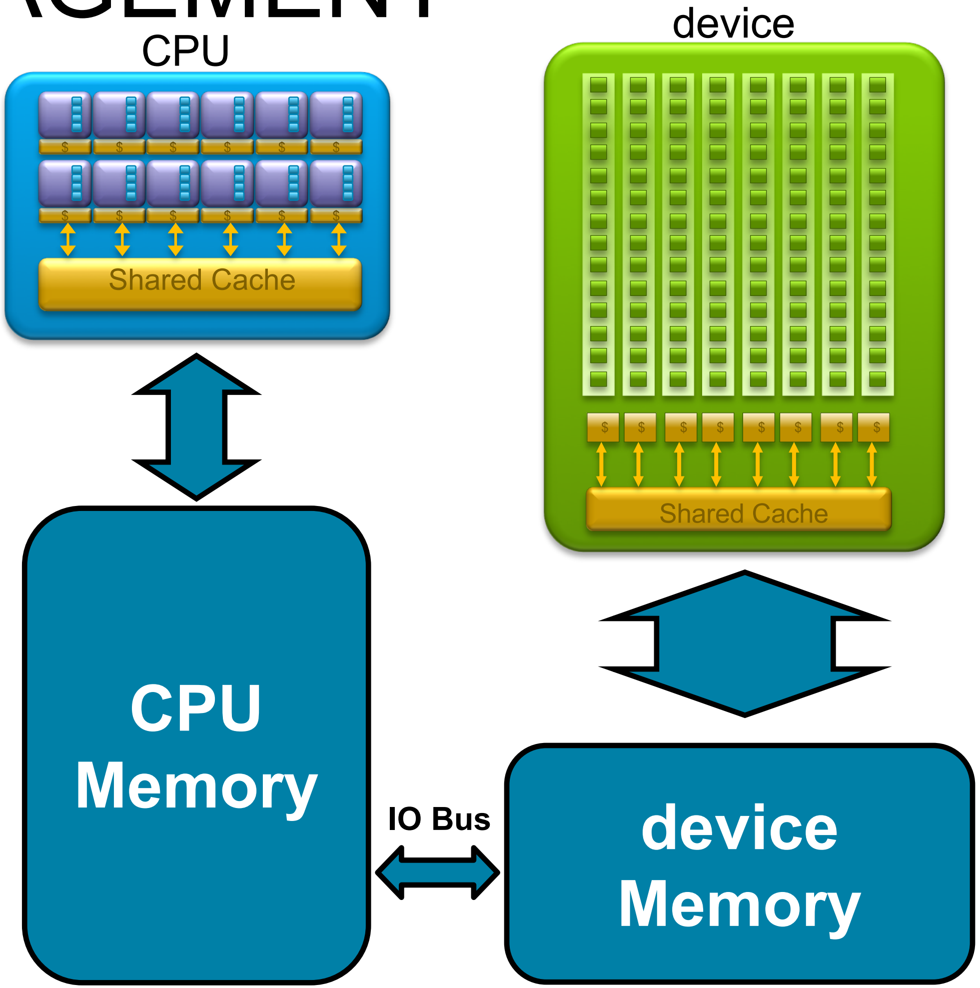
Time spent in :
Data Movement
Initialization
NOT Compute!

# OPTIMIZE DATA MOVEMENT

# EXPLICIT MEMORY MANAGEMENT

## Key problems

- Many parallel accelerators (such as devices) have a separate memory pool from the host
- These separate memories can become out-of-sync and contain completely different data
- Transferring between these two memories can be a very time consuming process



# OPENACC DATA DIRECTIVE

## Definition

- The data directive defines a lifetime for data on the device
- During the region data should be thought of as residing on the accelerator
- Data clauses allow the programmer to control the allocation and movement of data

```
#pragma acc data clauses
{

  < Sequential and/or Parallel code >

}
```

```
!$acc data clauses

  < Sequential and/or Parallel code >

!$acc end data
```

# DATA CLAUSES

`copy( list )` **Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.**

**Principal use:** For many important data structures in your code, this is a logical default to input, modify and return the data.

`copyin( list )` **Allocates memory on GPU and copies data from host to GPU when entering region.**

**Principal use:** Think of this like an array that you would use as just an input to a subroutine.

`copyout( list )` **Allocates memory on GPU and copies data to the host when exiting region.**

**Principal use:** A result that isn't overwriting the input data structure.

`create( list )` **Allocates memory on GPU but does not copy.**

**Principal use:** Temporary arrays.

# ARRAY SHAPING

- Sometimes the compiler needs help understanding the *shape* of an array
- The first number is the start index of the array
- In C/C++, the second number is how much data is to be transferred
- In Fortran, the second number is the ending index

```
copy(array[starting_index:length])
```

C/C++

```
copy(array(starting_index:ending_index))
```

Fortran

# ARRAY SHAPING (CONT.)

## Multi-dimensional Array shaping

```
copy(array[0:N][0:M])
```

C/C++

Both of these examples copy a 2D array to the device

```
copy(array(1:N, 1:M))
```

Fortran

# ARRAY SHAPING (CONT.)

## Partial Arrays

```
copy(array[i*N/4:N/4])
```

C/C++

Both of these examples copy only ¼ of the full array

```
copy(array(i*N/4:i*N/4+N/4))
```

Fortran

# STRUCTURED DATA DIRECTIVE

## Example

```
#pragma acc data copyin(a[0:N],b[0:N]) copyout(c[0:N])
{
  #pragma acc parallel loop
  for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
  }
}
```

**Action**

**Host Memory**

A | B | C'

**Device memory**

A | B | C'

# OPTIMIZED DATA MOVEMENT

```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
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++;
}
```

**Copy A to/from the accelerator only when needed.**

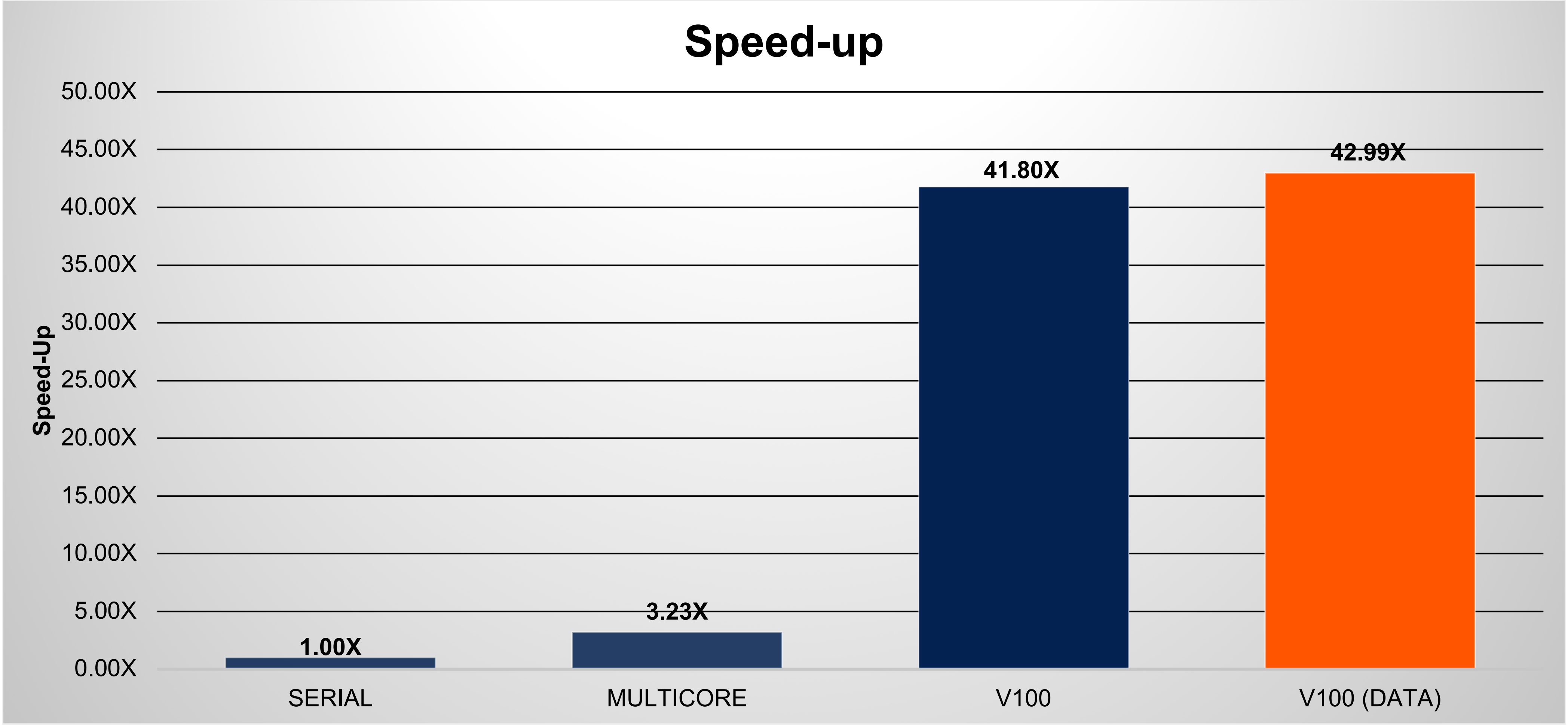
**Copy initial condition of Anew, but not final value**

# REBUILD THE CODE

```
pgcc -fast -ta=tesla -Minfo=accel laplace2d_uvm.c
main:
     60, Generating copy(A[:m*n])
         Generating copyin(Anew[:m*n])
     64, Accelerator kernel generated
         Generating Tesla code
         64, Generating reduction(max:error)
         65, #pragma acc loop gang /* blockIdx.x */
         67, #pragma acc loop vector(128) /* threadIdx.x */
     67, Loop is parallelizable
     75, Accelerator kernel generated
         Generating Tesla code
         76, #pragma acc loop gang /* blockIdx.x */
         78, #pragma acc loop vector(128) /* threadIdx.x */
     78, Loop is parallelizable
```

Now data movement only happens at our data region.

# OPENACC SPEED-UP



# TRY IT OUT!

- Take your existing code, and try to enclose it with a data region using the following directives/clause

  [#pragma/!] acc data [copyin, copyout, create]

```
cc -Minfo=all -acc -ta=tesla -o openacc_demo_c openacc_demo.c
main:
     28, Generating copyin(A[:N],B[:N],C[:N])
         Generating copyout(sumC)
     31, Accelerator kernel generated
         Generating Tesla code
         32, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     52, Accelerator kernel generated
         Generating Tesla code
         53, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     72, Accelerator kernel generated
         Generating Tesla code
         72, Generating reduction(+:sumC)
         73, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

# RUN THE NEW CODE

```
> aprun -n 1 ./openacc_demo_c
Initialize check:
A[   0] = 0 (0) B[0] = 0 (0)
A[100] = 0 (100) B[100] = 0 (200)
A[1623] = 0 (1623) B[1623] = 0 (3246)
A[111111] = 0 (111111) B[111111] = 0 (222222)
saxpy check:
C[   0] = 0 (0)
C[100] = 0 (400)
C[1623] = 0 (6492)
C[111111] = 0 (444444)
sumC check: 2e+14 (2e+14)
Total time: 0.2814s
Init time: 0.2792s
SAXPY time: 0.0007s
SumC time: 0.0013s
Application 17826786 resources: utime ~0s, stime ~1s, Rss ~150268,
inblocks ~256, outblocks ~122
```

Compute got really fast!

# RUN THE NEW CODE

```
> aprun -n 1 ./openacc_demo_c
Initialize check:
A[   0] = 0 (0) B[0] = 0 (0)
A[100] = 0 (100) B[100] = 0 (200)
A[1623] = 0 (1623) B[1623] = 0 (3246)
A[111111] = 0 (111111) B[111111] = 0 (222222)
saxpy check:
C[   0] = 0 (0)
C[100] = 0 (400)
C[1623] = 0 (6492)
C[111111] = 0 (444444)
sumC check: 2e+14 (2e+14)
Total time: 0.2814s
Init time: 0.2792s
SAXPY time: 0.0007s
SumC time: 0.0013s
Application 17826786 resources: utime ~0s, stime ~1s, Rss ~150268,
inblocks ~256, outblocks ~122
```

But we're getting wrong answers!

Compute got really fast!

# DATA SYNCHRONIZATION

# OPENACC UPDATE DIRECTIVE

**update:** Explicitly transfers data between the host and the device

Useful when you want to synchronize data in the middle of a data region

Clauses:

- **self:** makes host data agree with device data
- **device:** makes device data agree with host data

```
#pragma acc update self(x[0:count])
#pragma acc update device(x[0:count])
```

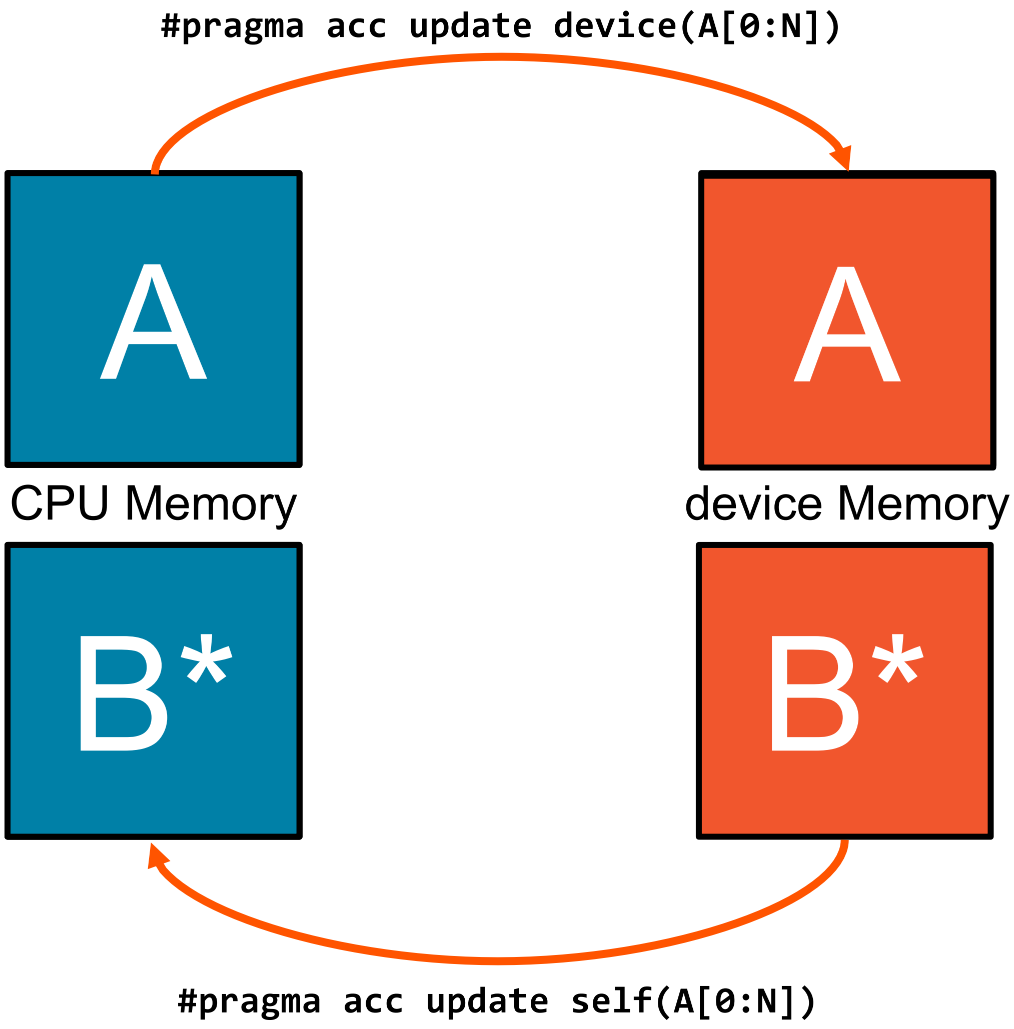
C/C++

```
!$acc update self(x(1:end_index))
!$acc update device(x(1:end_index))
```

Fortran

# OPENACC UPDATE DIRECTIVE

The data must exist on both the CPU and device for the update directive to work.



# SYNCHRONIZE DATA WITH UPDATE

```
int* allocate_array(int N){
  int* A=(int*) malloc(N*sizeof(int));
  #pragma acc enter data create(A[0:N])
  return A;
}

void deallocate_array(int* A){
  #pragma acc exit data delete(A)
  free(A);
}

void initialize_array(int* A, int N){
  for(int i = 0; i < N; i++){
    A[i] = i;
  }
  #pragma acc update device(A[0:N])
}
```

- Inside the **initialize** function we alter the host copy of **'A'**
- This means that after calling **initialize** the host and device copy of **'A'** are out-of-sync
- We use the **update** directive with the **device** clause to update the device copy of **'A'**
- Without the **update** directive later compute regions will use incorrect data.

# TRY IT OUT!

- Take your existing code, and add the update directive to get the answers off the GPU

[#pragma/!] acc update [self/device]

Remember! Data slicing rules apply here too! Pay attention to the compiler output!

```
cc -Minfo=all -acc -ta=tesla -o openacc_demo_c openacc_demo.c
main:
     28, Generating copyin(A[:N],B[:N],C[:N])
         Generating copyout(sumC)
     31, Accelerator kernel generated
         Generating Tesla code
         32, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     43, Generating update self(B[100],A[100],B[111111],A[111111],A[:1],B[:1],B[1623],A[1623])
     53, Accelerator kernel generated
         Generating Tesla code
         54, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     63, Generating update self(C[100],C[111111],C[:1],C[1623])
     73, Accelerator kernel generated
         Generating Tesla code
         73, Generating reduction(+:sumC)
         74, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

# ONE MORE OPTIMIZATION

- We don't need to copy A,B,C, just create!

[#pragma/!] acc data create

```
cc -Minfo=all -acc -ta=tesla -o openacc_demo_c openacc_demo.c
main:
     28, Generating create(A[:N],B[:N],C[:N])
         Generating copyout(sumC)
     31, Accelerator kernel generated
         Generating Tesla code
         32, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     43, Generating update self(B[100],A[100],B[111111],A[111111],A[:1],B[:1],B[1623],A[1623])
     53, Accelerator kernel generated
         Generating Tesla code
         54, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     63, Generating update self(C[100],C[111111],C[:1],C[1623])
     73, Accelerator kernel generated
         Generating Tesla code
         73, Generating reduction(+:sumC)
         74, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

# RUN THE NEW CODE

```
> aprun -n 1 ./openacc_demo_c
Initialize check:
A[   0] = 0 (0) B[0] = 0 (0)
A[100] = 100 (100) B[100] = 200 (200)
A[1623] = 1623 (1623) B[1623] = 3246 (3246)
A[111111] = 111111 (111111) B[111111] = 222222 (222222)
saxpy check:
C[   0] = 0 (0)
C[100] = 400 (400)
C[1623] = 6492 (6492)
C[111111] = 444444 (444444)
sumC check: 2e+14 (2e+14)
Total time: 0.2258s
Init time: 0.1779s
SAXPY time: 0.0008s
SumC time: 0.0014s
Application 17827100 resources: utime ~0s, stime ~1s, Rss ~150284,
inblocks ~257, outblocks ~131
```

Right Answers!

Fast compute!

# PROFILE AGAIN

```
➢ PMI_NO_FORK=1 aprun -n 1 -b nvprof ./openacc_demo_c
…
==1684== Profiling application: ./openacc_demo_c
==1684== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   38.97%  842.08us         1  842.08us  842.08us  842.08us  main_73_gpu
                   33.28%  719.04us         1  719.04us  719.04us  719.04us  main_53_gpu
                   19.38%  418.69us         1  418.69us  418.69us  418.69us  main_31_gpu
                    7.30%  157.66us         1  157.66us  157.66us  157.66us  main_73_gpu_red
                    1.07%  23.168us        13  1.7820us  1.5040us  3.1040us  [CUDA memcpy DtoH]
      API calls:   61.93%  287.19ms         2  143.59ms     723ns  287.19ms  cuDevicePrimaryCtxRetain
```

Time spent in :
Initialization

Setting up GPUs takes some (almost) constant time, and this is a very small code. This 0.2s won't matter in a real simulation code.

# OPENACC RESOURCES

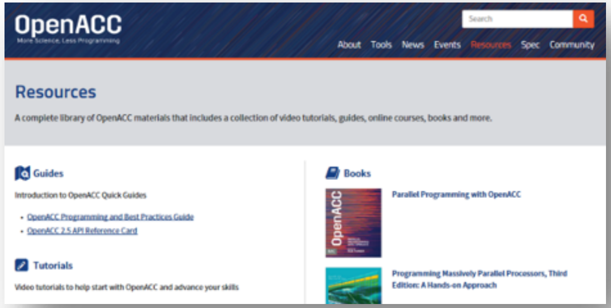
Guides • Talks • Tutorials • Videos • Books • Spec • Code Samples • Teaching Materials • Events • Success Stories • Courses • Slack • Stack Overflow

**FREE Compilers**

PGI Community EDITION

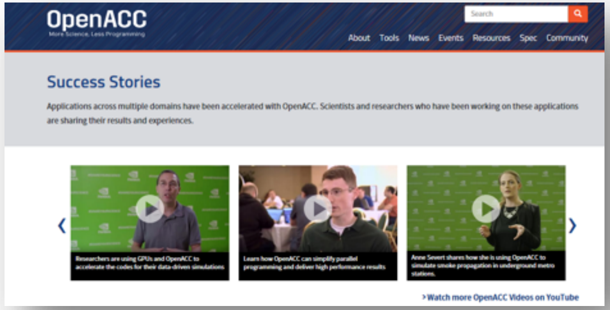
## Resources

https://www.openacc.org/resources



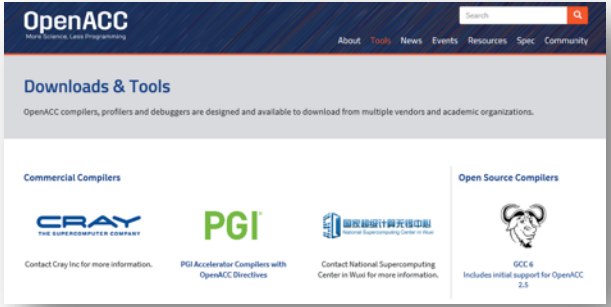

## Success Stories

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



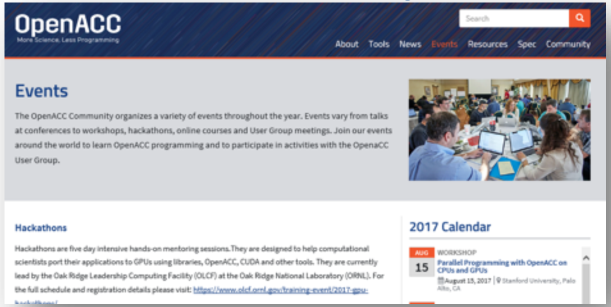

## Compilers and Tools

https://www.openacc.org/tools




## Events

https://www.openacc.org/events




OpenACC More Science, Less Programming
NVIDIA DEEP LEARNING INSTITUTE

# CLOSING REMARKS

# KEY CONCEPTS

## In this lab we discussed…

- How to profile a serial code to identify loops that should be accelerated
- How to use OpenACC's parallel loop directive to parallelize key loops
- How to use OpenACC's data clauses to control data movement
- To always check accuracy first!!