# Programming GPUs with OpenX

HPC Summer School - 2025

N. Shukla

HPC Application Engineer

CINECA

# Contents: Topics Explored

1

## House keeping

Where to find the course material and what problem will you solve?

2

## OpenACC directives

Parallelising and profiling with OpenACC Toolkit on NVIDIA GPUs

3

## Data Management

Explicit data management, Data regions and clause, Unstructured Data Lifetimes

4

## Loop Optimisation

Overlapping kernel execution & data transfer on Single/Multi GPU

# Scan and Git Clone or Just Download Zip File

## Download here

https://gitlab.hpc.cineca.it/cinecasummerschool/hpc25cineca/-/tree/main/Day-5/openACC

## ONLY 4 exercise

A skeleton of exercises are provided.

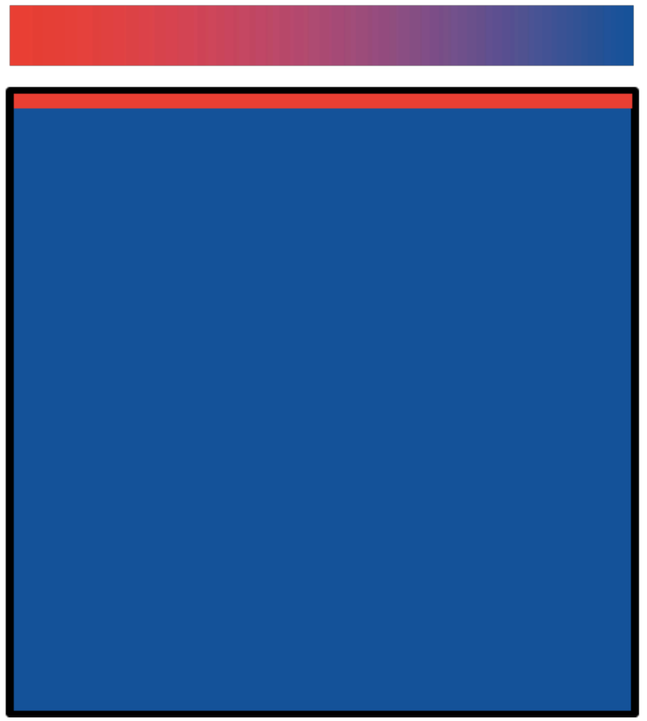
# 0 Case study: Parallelise a serial Laplace 2D

# Laplace Heat Transfer

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.

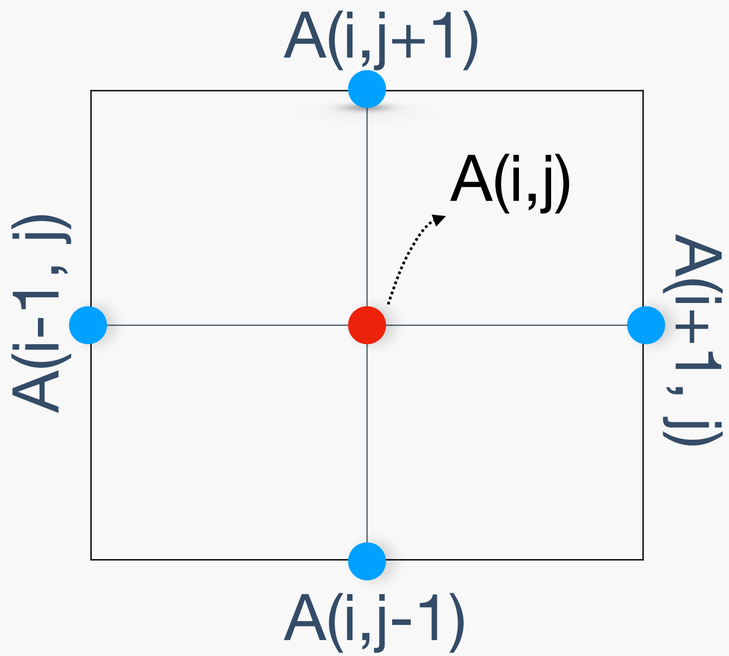


# Code Description

- Iteratively converges to correct value (e.g. Temperature)
- by computing new values at each point from the average of neighboring points.
- Example: Solve Laplace equation in 2D

$$\nabla^2 f(x,y) = 0$$

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



# Code Description

```
while (error > tol && niter < niter_max)
error = 0.0;

  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; ++i) {
      Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);

      error = fmax(error, fabs(Anew[idx] - A[idx])); }
      }
  for (int j = 1; j < n-1; ++j)
    for (int i = 1; i < m-1; ++i)
      A[j][i] = Anew[j][i]
```

Iterate until converged

Iterate cross matrix elements

Compute new quantity

Compute max error for converge

Swap input/output arrays

# Compiling code with NVHPC

# NVIDIA’s HPC Compilers (AKA PGI)

## Compiling sequential code

Instruct compiler to print feedback about the compiled code

- -Minfo = accel informs about what parts of the code were accelerated via OpenX
- -Minfo = opt informs about all code optimisations
- -Minfo=all gives all code feedback, whether positive or negative

```
nvc -fast my_program.c
nvc++ -fast  my_program.cpp
nvfortran -fast  my_program.cf90
```

# NVIDIA’s HPC Compilers (AKA PGI)

## -Minfo flag

Instruct compiler to print feedback about the compiled code

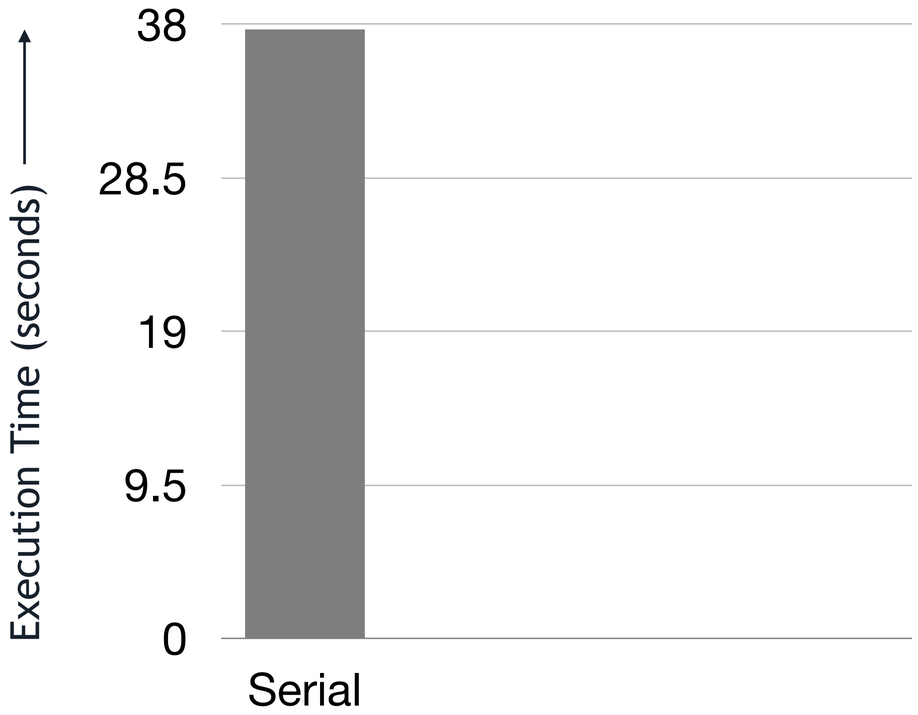
- -Minfo = accel informs about what parts of the code were accelerated via OpenX
- -Minfo = opt informs about all code optimisations
- -Minfo=all gives all code feedback, whether positive or negative

```
nvc -fast -Minfo=all my_program.c
nvc++ -fast -Minfo=all  my_program.cpp
nvfortran -fast -Minfo=all  my_program.cf90
```

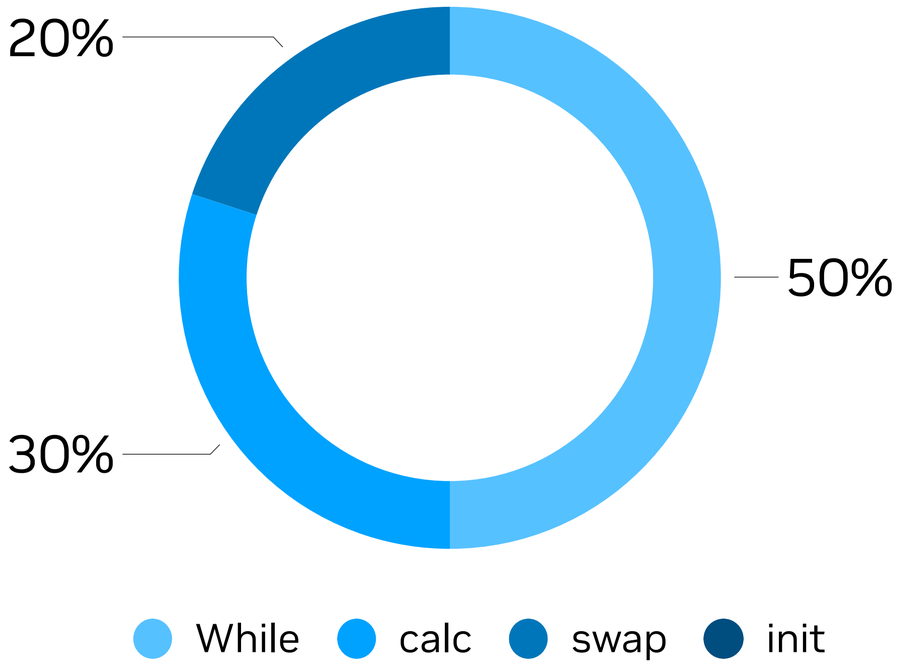
# Code Description

Simulation was performed 1000 Iterations

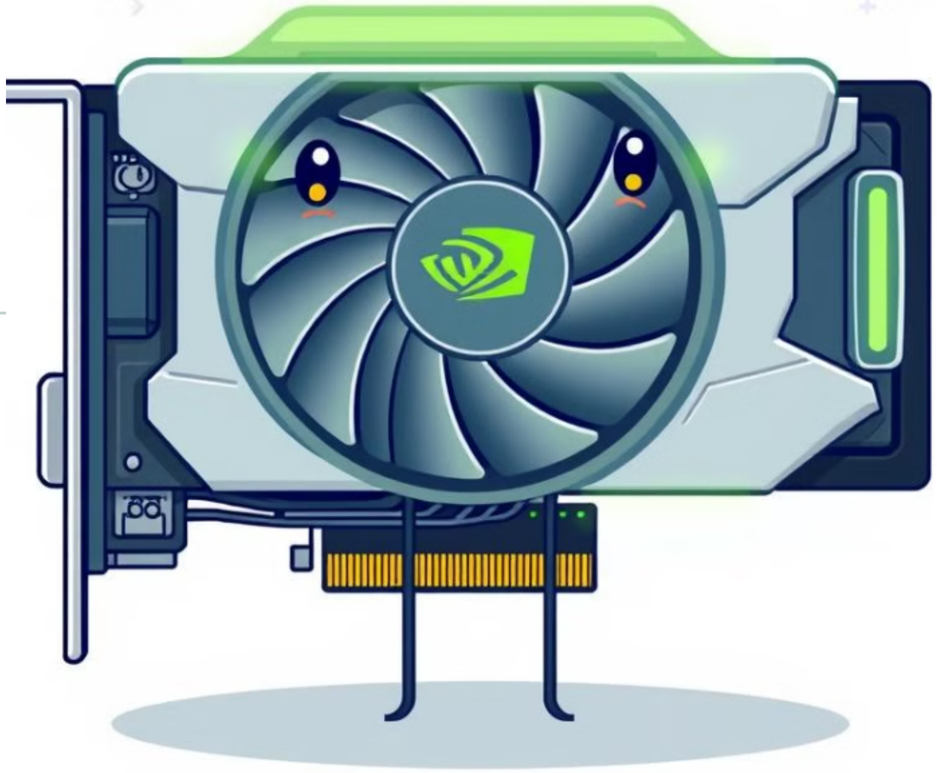
Total Runtime: 37.69 seconda



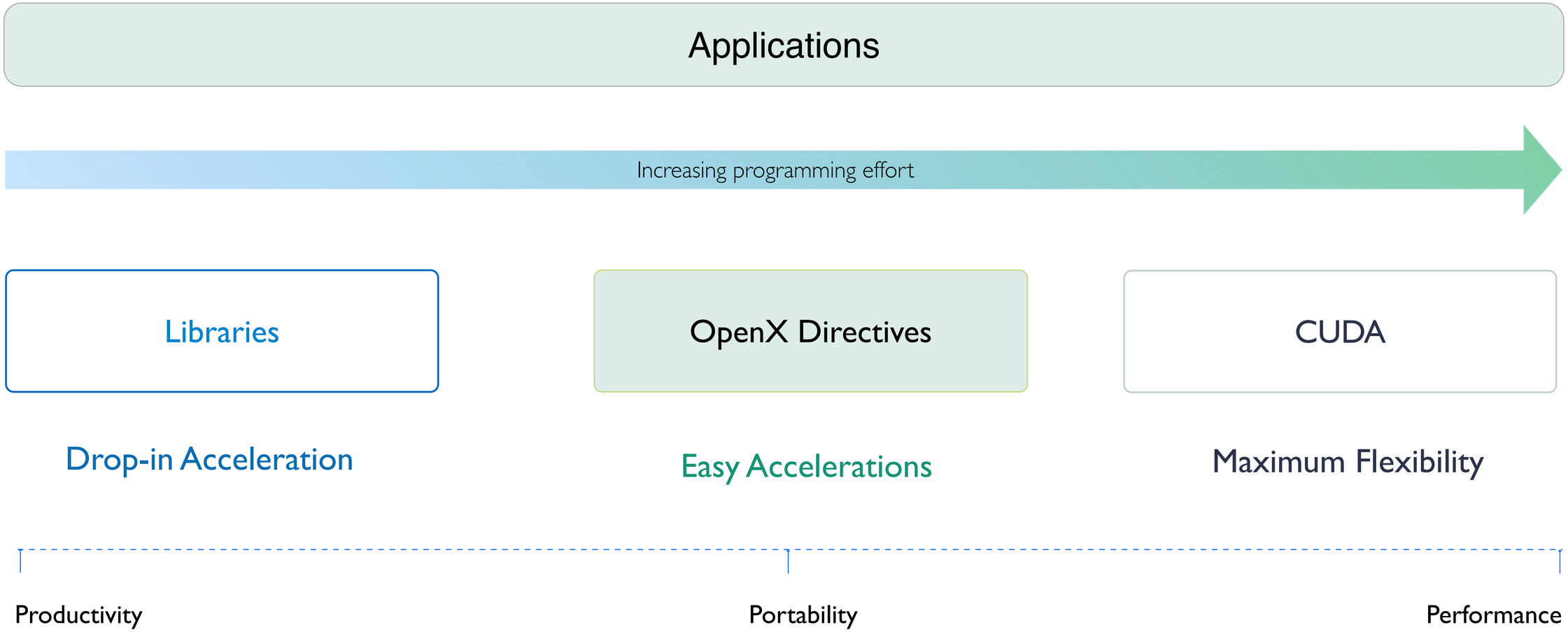

Profiling sequential code



# 1 Accelerating your application on GPU



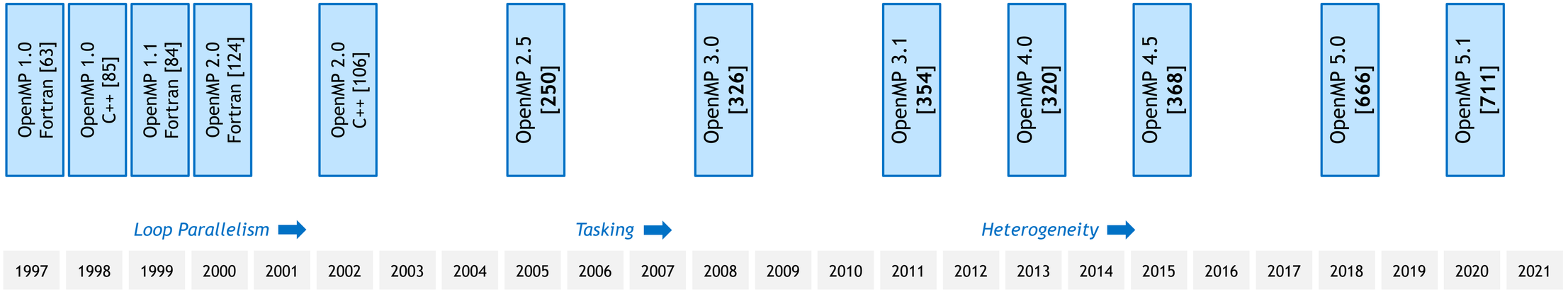
# Ways to Parallels an Applications on Nvidia GPUs



Are you familiar with directive based parallelism?

# OpenX (X = OMP, ACC): 1997-2021

Looking at TIME (pace of innovation) and SPACE (specification length)

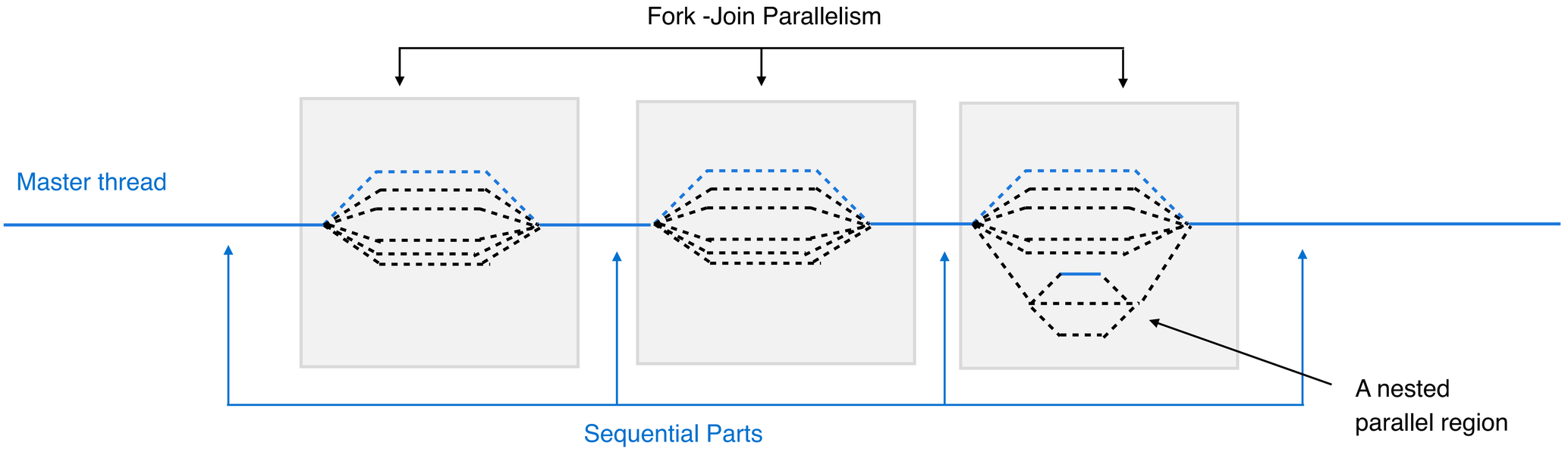



**Members of OpenMP ARB = 33**

# Revisit: OpenMP Application Program Interface (API)

Allows programmers to develop threaded parallel codes on shared memory computational units

- Directives are understood by OpenMP aware compilers (others are free to ignore)
- Generates parallel threaded code
  - Original thread becomes thread “0”
  - Share resources of the original thread (or rank)
  - Data-sharing attributes of variables can be specified based on usage patterns

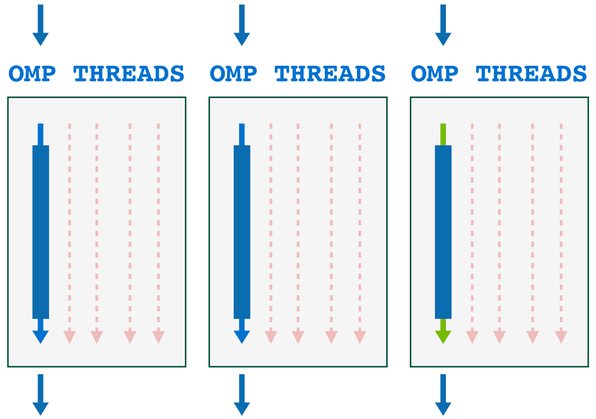


# Revisit: OpenMP Application Program Interface (API)

Allows programmers to develop threaded parallel codes on shared memory computational units

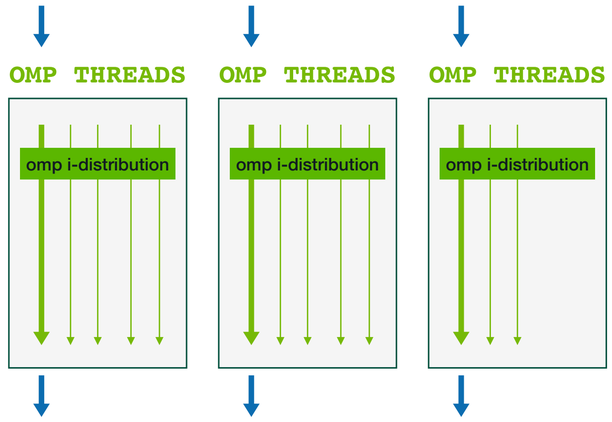
- Creates a team of OpenMP threads that execute the structured-block that follows
- Number of threads property is generally specified by OMP_NUM_THREADS

*#pragma omp parallel*




All threads will execute the region

*#pragma omp parallel for*




All threads will execute a part of the iterations

# Revisit: OpenMP Worksharing

## Serial

```
for (int i = 0; i < N; ++i)
{
  C[i] = A[i] + B[i];
}
```

- 1 thread/process will execute each iteration sequentially
- Total time = time_for_single_iteration * N

## Parallel

```
#pragma omp parallel
for (int i = 0; i < N; ++i)
{
  C[i] = A[i] + B[i];
}
```

- Say, OMP_NUM_THREADS = 4
- 4 threads will execute each iteration redundantly (overwriting values of C)
- Total time = time_for_single_iteration * N

## Parallel worksharing

```
#pragma omp parallel for
for (int i = 0; i < N; ++i)
{
  C[i] = A[i] + B[i];
}
```

- Say, OMP_NUM_THREADS = 4
- 4 threads will execute each iteration (roughly N/4 per thread)
- Total time = time_for_single_iteration * N/4

Checkpoint-0:  00-laplace2d_serial: parallelize with OpenMP

# Exercise0  00-Laplace2D_Serial: Parallelize With OpenMP

```
while ( error > tol && iter < iter_max ) {
  error=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]);
      error = max(error, 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++;
}
```

## Things to do

- Make these two loops parallel with OpenMP directives
- by changing the number of OpenMP threads, repeating the execution with: export OMP_NUM_THREADS = (1 to 8 )
- Do not forget to record the time

# NVIDIA’s HPC Compilers (AKA PGI)

## Building and running enabled OpenMP code

- nvc -mp -fast -Minfo=all my_program.c -o bin
- nvc++ -mp  -fast -Minfo=all my_program.cpp. -o bin
- nvfortran -mp  -fast -Minfo=all my_program.cf90.  -o bin

# Solution

# Solution0 - 01-Laplace2D-Openmp: Parallelize With OpenMP

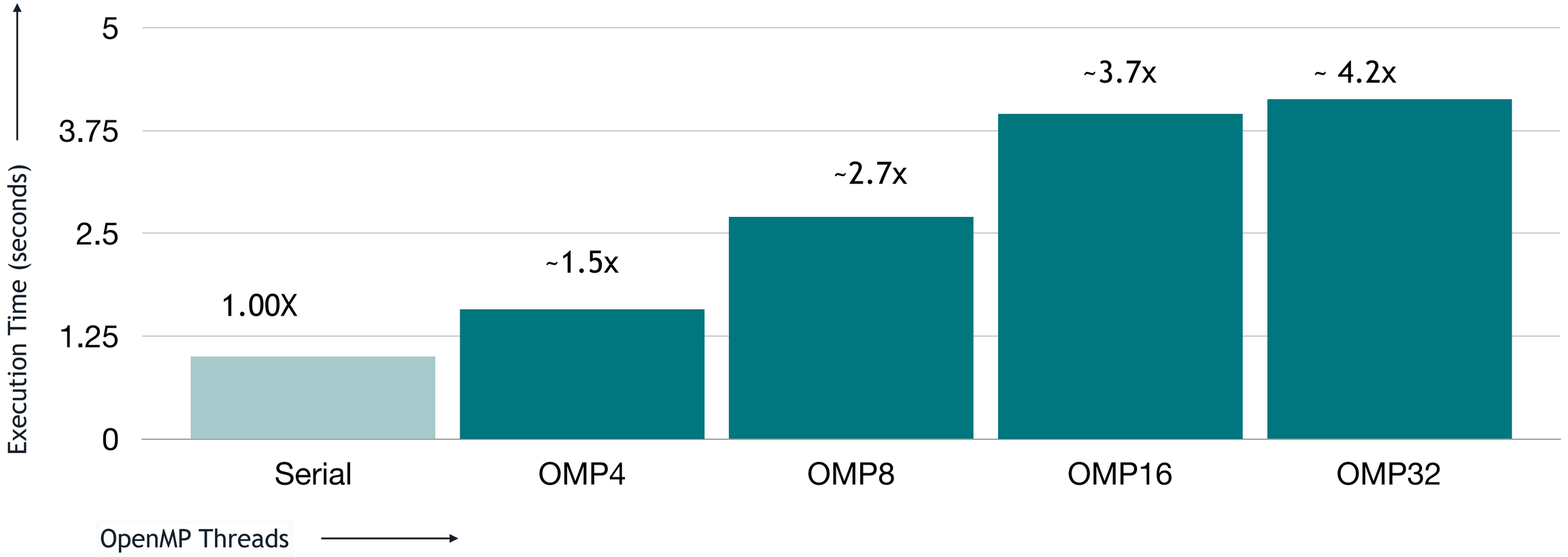
```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma omp parallel for shared(m, n, Anew, A) reduction(max:error)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                   A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma omp parallel for shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

# Solution0 - 01-Laplace2D-Openmp: Parallelize With OpenMP

```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma omp parallel for collapse(2) shared(m, n, Anew, A) reduction(max:error)
  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]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma omp parallel for collapse(2) shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

# Performance Speed Up (Higher Is Better)

Simulation was performed 1000 Iterations on Leonardo

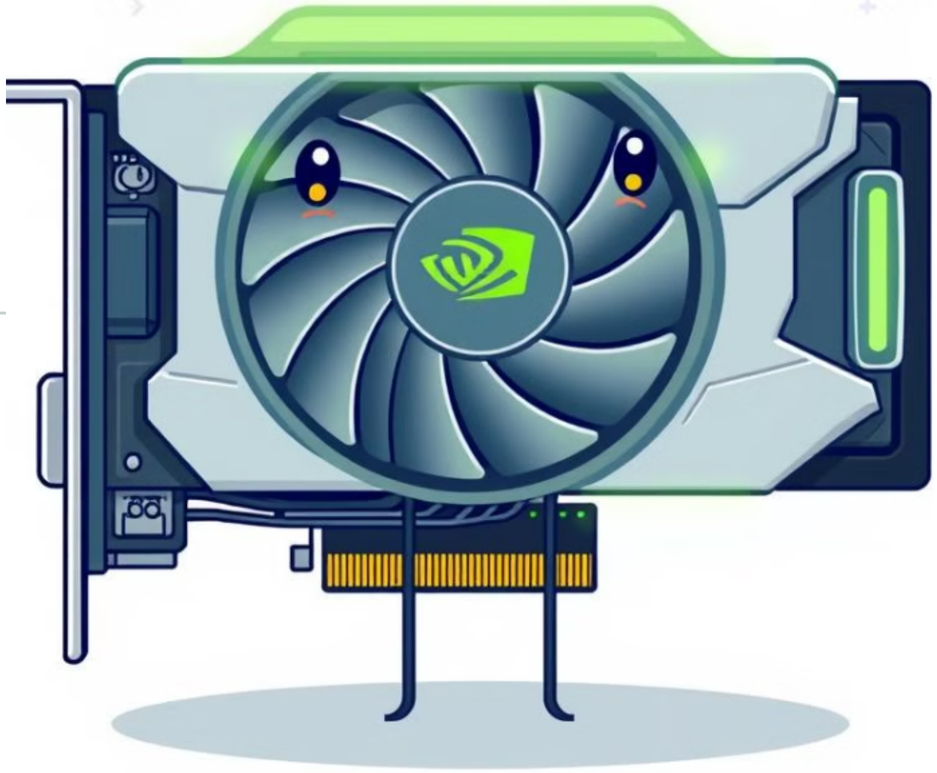


Will your legacy OpenMP code perform well on the GPU?

Course objectives:

2

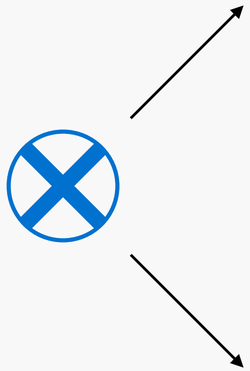
Enable you to accelerate your application with OpenACC



# What is OpenX offloadings?

# What Is OpenX?

designed for performance and portability



OpenACC
More Science, Less Programming

Main focus is to target to offloading code onto GPUs

OpenMP

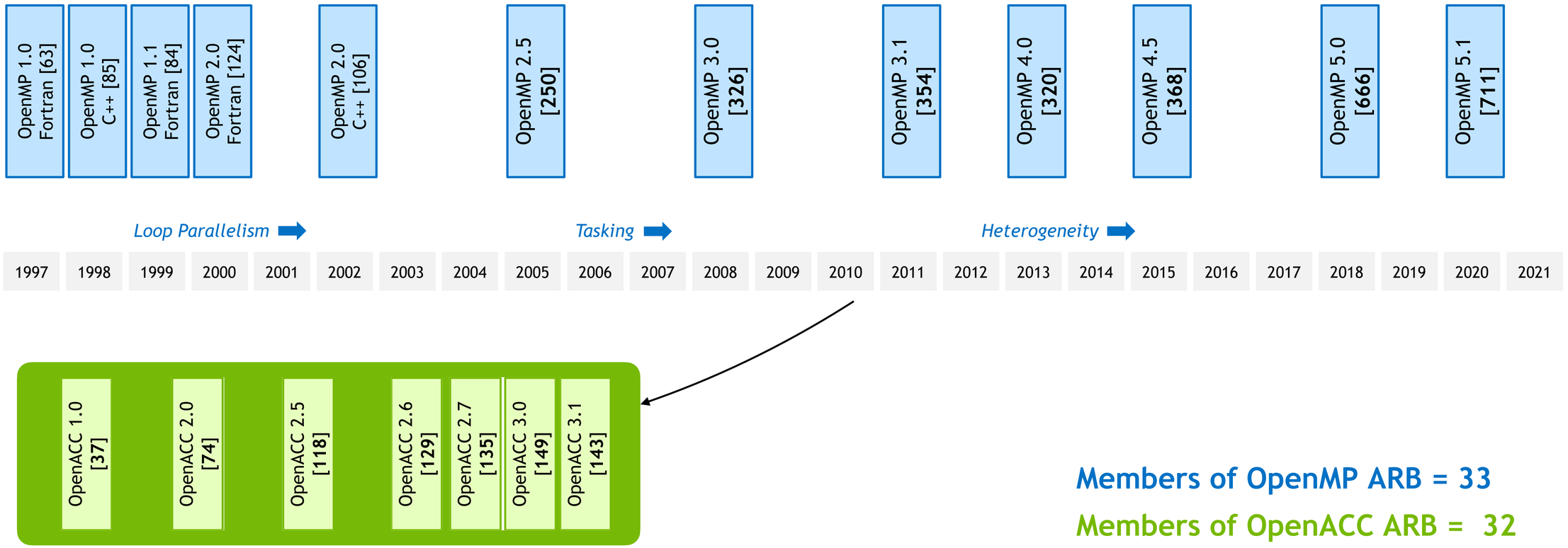
from v 4.0 allows offloading of tasks onto GPUs

**IMPORTANT**

This course will not give you any concrete hints if OpenMP is better than OpenACC

# OpenX (X = OMP, ACC): 1997-2021

Looking at TIME (pace of innovation) and SPACE (specification length)




**Members of OpenMP ARB = 33**

**Members of OpenACC ARB = 32**

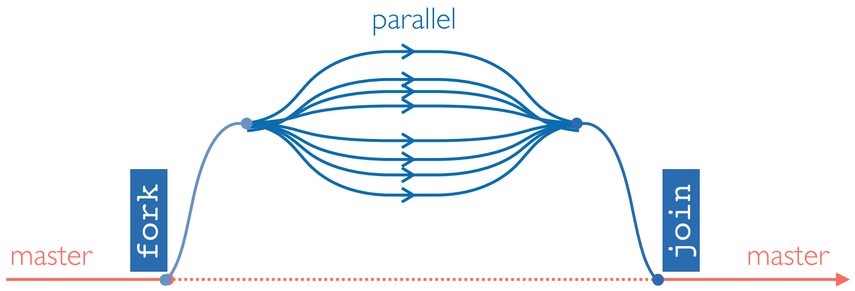
# Same Basic Principle: Fork-Join Model

## OpenACC

Specifically targets GPU accelerators

It started after OpenMP

Basic principle: fork-join model




OpenACC is more descriptive

**Compiler support**

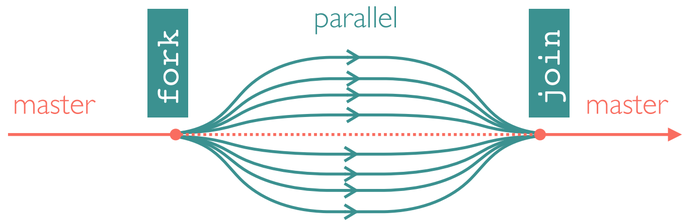
PGI, Cray, NVIDIA

## OpenMP

Designed to replace low-level and multi-threaded programming solutions like POSIX, threads or Pthreads

Intend to target independent processor (shared memory)

Basic principle: fork-join model




OpenMP 4./4.5 onwards; offloading capabilities

OpenMP is more prescriptive

**Compiler support**

GCC, Intel, IBM XL, LLVM/Clang etc

# OpenX Uses Host-Device Model

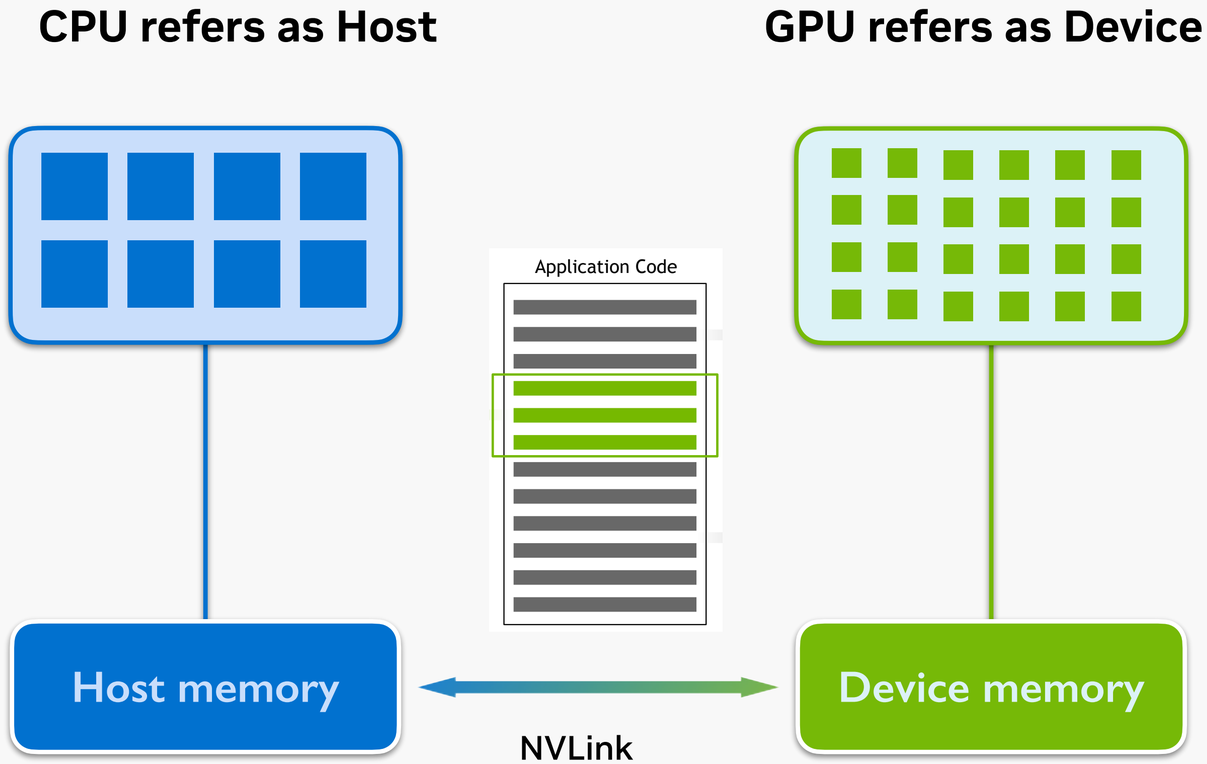
CPU and GPU work together for best benefit and performances

## Device model

- Host-centric model
- One host device and multiple devices of the same type
- Devices are connected to host CPU via interconnect such as PCIe or NVLink
- Host and device have separate memory spaces

## GPGPU execution model

- A function which runs on a GPU is called a **Kernels**
- Kernels are executed as a set of threads that can run concurrently
- Each thread is mapped to a single CUDA core on the GPU
- CUDA threads executes in a Single Program Multiple Data (SPMD)



# Before Start, Let Agree on Some Key Jargon

English is English, there is no debate

## specification

***noun*** [ C or U ]

**UK** /ˌspes.ɪ.fɪˈkeɪ.ʃᵊn/ **US** /ˌspes.ə.fəˈkeɪ.ʃᵊn/

(informal **spec**)

**C1**

**a detailed description of how something should be done, made, etc.:**

- *All products are made exactly **to** the customer's specifications.*
- *A specification has been **drawn up** for the new military aircraft.*
- *a job specification*
- *The cars have been built **to** a **high** specification (= a high standard).*

## standard

***noun***

**UK** /ˈstændəd/ **US**

[ C ] • MEASURES •

( written abbreviation **std.**)

**an official rule, unit of measurement, or way of operating that is used in a particular area of manufacturing or services:**

- **a common/a global/an international standard** *MP3 is a recognized global standard for audio encoding and compression.*
- *an **accountancy/industry/legal standard***
- *Several states were given notice that they do not **meet the standard** for ozone levels.*
- ***safety standards***
- *a **federal/government/national standard***
- ***enforce/tighten standards***

A specification can or cannot be adopted. A standard can be adopted de jure or de facto

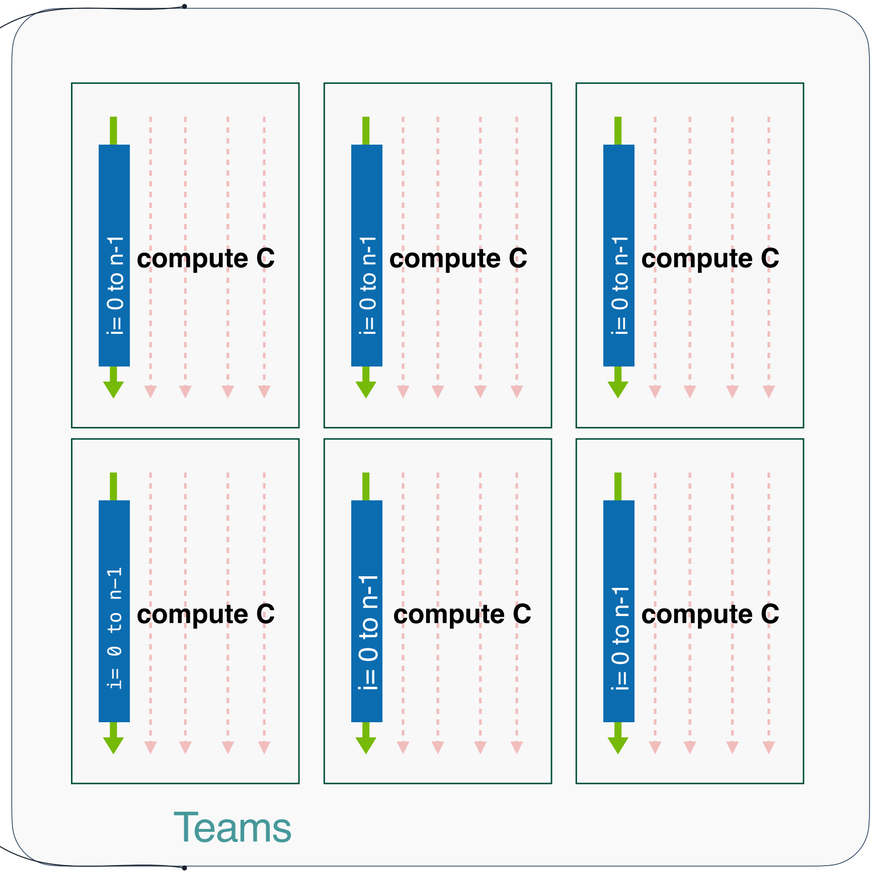
Source: https://dictionary.cambridge.org/dictionary/english/specification
Source: https://dictionary.cambridge.org/dictionary/english/standard

# Offloading Serial Code With OpenX

Instructions to the compiler on how to compile with code

```
#pragma acc parallel
#pragma omp target teams
{
  Compiler will generate 1 or more parallel GANGS or Teams of
  threads which execute block of code redundantly
}
```

```
!$acc parallel
!$omp target teams
DO I = 1, N
  Compiler will generate 1 or more parallel GANGS or Teams of threads
  which execute block of code redundantly
END DO
!$omp end target teams
!$acc end parallel
```



# What and why OpenACC directives?

# OpenACC Directives: Simple, Powerful and Portable

CPU and GPU work together for best benefit and performances

**OpenACC** is …

a directive-based

parallel programming model

designed for

performance and portability

Add Simple Compiler Directive

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

# OpenACC Directives

CPU and GPU work together for best benefit and performances

Simple I Powerful I Portable

- Incremental
- Single Source
- Interoperable
- Performance portable: CPU, GPU, MIC

1 Management Data Movement

2 Initiate Parallel Execution

3 Optimize Loop Mappings

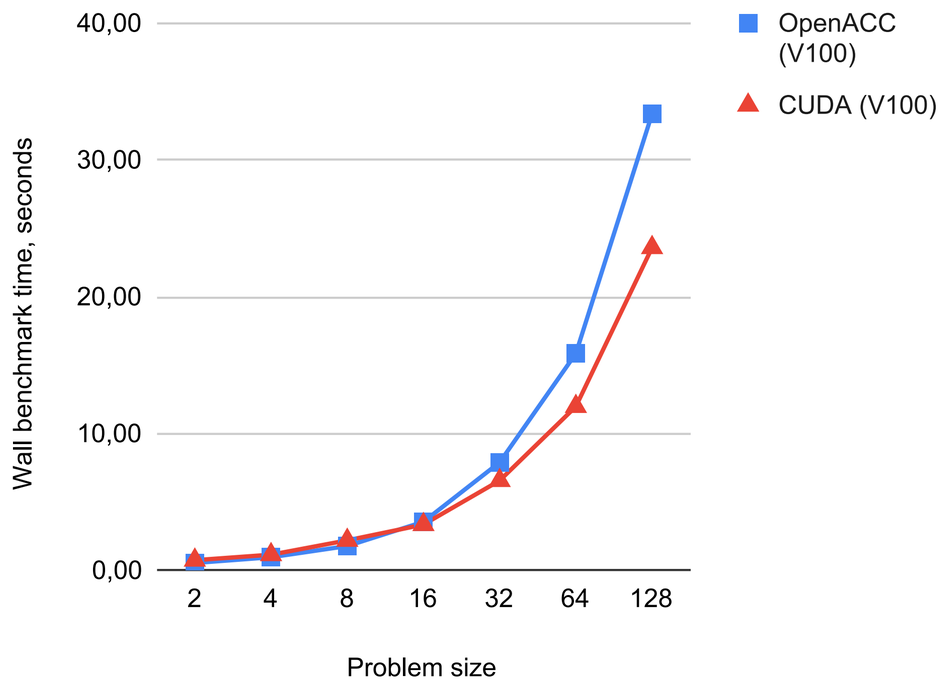
```
#pragma acc data copyin(a,b) copycout(c)
{

  ...
   #pragma acc parallel
   {

   #pragma acc loop gang vector
    for (int idx=0; idx<N; idx++)
        C[idx] = A[idx] + B[idx];
   }

}
```

# OpenACC Directives: Simple, Powerful and Portable

```
main()
 {
   <serial code>
   #pragma acc kernels // Automatically runs on GPU
   <parallel code>
}
```

## OpenACC VS CUDA

- Powerful: Could reach to 98 % speed in comparison to CUDA
- A rough estimate 10,0000 +  Developers using OpenACC
- Scientific research, Industry adoption
- Tool and Ecosystem Support: NVHPC, CRAY, LLVM




Comparison of OpenACC and CUDA performance on Cloverleaf application (doi:10.1088/1742-6596/1740/1/012056)

# OpenACC Directives: Simple, Powerful and Portable

CPU and GPU work together for best benefit and performances

## #pragma acc <directive> <clauses>

**#pragma** in C/C++ is what's known as a "compiler hint." These are very similar to programmer comments, however, the compiler will actually read our pragmas. Pragmas are a way for the programmer to "guide" the compiler, without running the chance damaging the code. If the compiler does not understand the pragma, it can ignore it, rather than throw a syntax error.

**acc** is an addition to our pragma. It specifies that this is an **OpenACC pragma**. Any non-OpenACC compiler will ignore this pragma. Even the nvc/nvc++ compiler can be told to ignore them. (which lets us run our parallel code sequentially!)

**directives** are commands in OpenACC that will tell the compiler to do some action. For now, we will only use directives that allow the compiler to parallelize our code.

**clauses** are additions/alterations to our directives. These include (but are not limited to) optimizations. The way that I prefer to think about it: directives describe a general action for our compiler to do (such as, paralellize our code), and clauses allow the programmer to be more specific (such as, how we specifically want the code to be parallelized).

GPU programming can be thought of as a two-step process

**Compute Bound**

- Exposing to the GPU environment

**Data Bound**

- Data management directives

From now we mostly focus on OpenACC

GPU programming can be thought of as a two-step process

**Compute Bound**

- Exposing to the GPU environment

**Data Bound**

- Data management directives

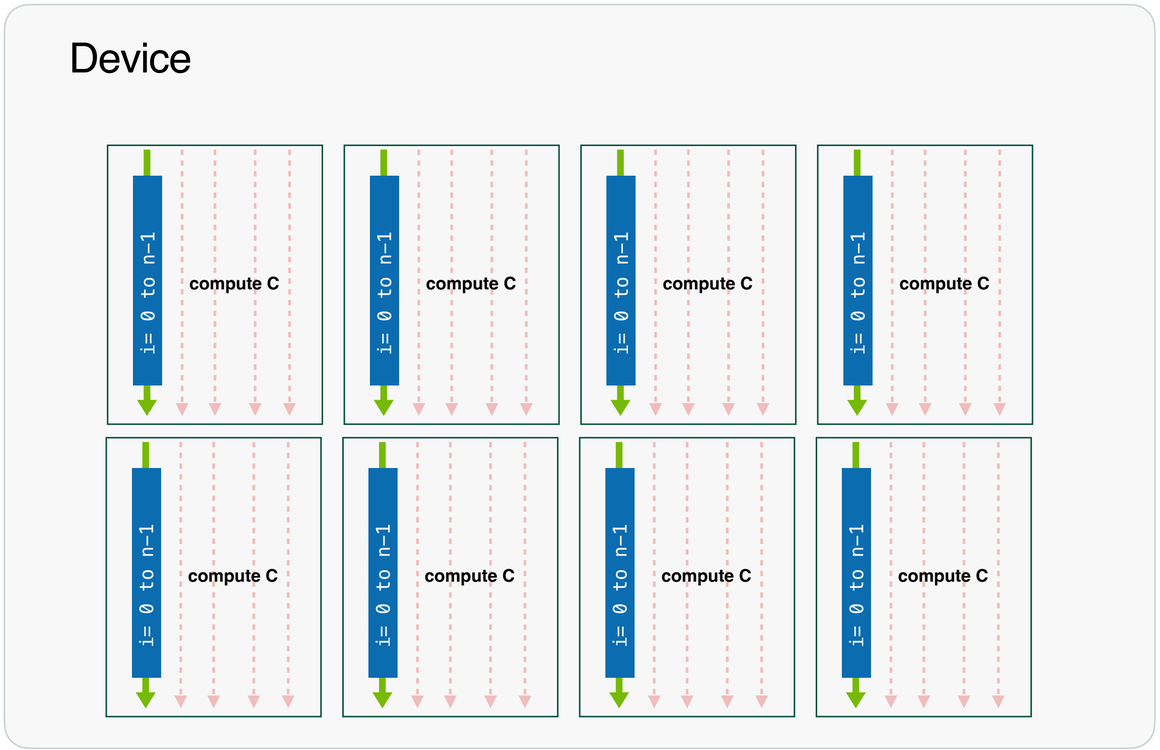
From now we mostly focus on OpenACC

# Parallelism Identified by Programer

**Parallel:** a parallel region of code. The compiler generates a parallel kernel for that region. Each gang will execute the entire loop

```
#pragma acc parallel
{

    for (int i=0; i<N; i++)
      c[i] = a[i] + b[i]
}
```

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

    for (int i=0; i<N; i++)
      a[i]++;
}
```



# Parallelism Identified by Programer

**Parallel:** a parallel region of code. The compiler generates a parallel kernel for that region. Each gang will execute the entire loop

**Loop:** identifies a loop that should be distributed across threads. Parallel and loop are often placed together

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

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

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

Similar to OpenMP, compiler translates the parallel region into a kernel that runs in parallel on the GPU

# Parallelism Identified by Programer

## Compiler determines what can be safely parallelised

**Kernels:** a parallel region of code. It allows the programmer to step back, and rely solely on the compiler

**Loop:** identifies a loop that should be distributed across threads. Parallel and loop are often placed together

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

    for (int i=0; i<N; I++) {d[I] = a*x[I] + y[I]}
}
```

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

}
```

No explicit counterpart

# Kernels Vs Parallel Construct

when fully optimised both will give  similar performance

## Kernel compute directives

- Compiler’s leeway parallelising and optimising a loop
- Only provide by OpenACC
- Compiler guarantees correctness
- Or at least it does two things:
  - Fuse the two loop nests into one loop nest
  - Generates two kernels
- Implicit barrier at the end and between each loop

## Parallel compute directives

- Programmer’s responsibility to identify region for parallelization
- Programmer guarantees correctness
- Parallel just run the same code on multiple threads redundantly
- Guarantees that no dependency occurs iterations
- Implicit barrier at the end of the parallel region

When fully optimized, both will give similar performance

# Development Cycle

### Analyze

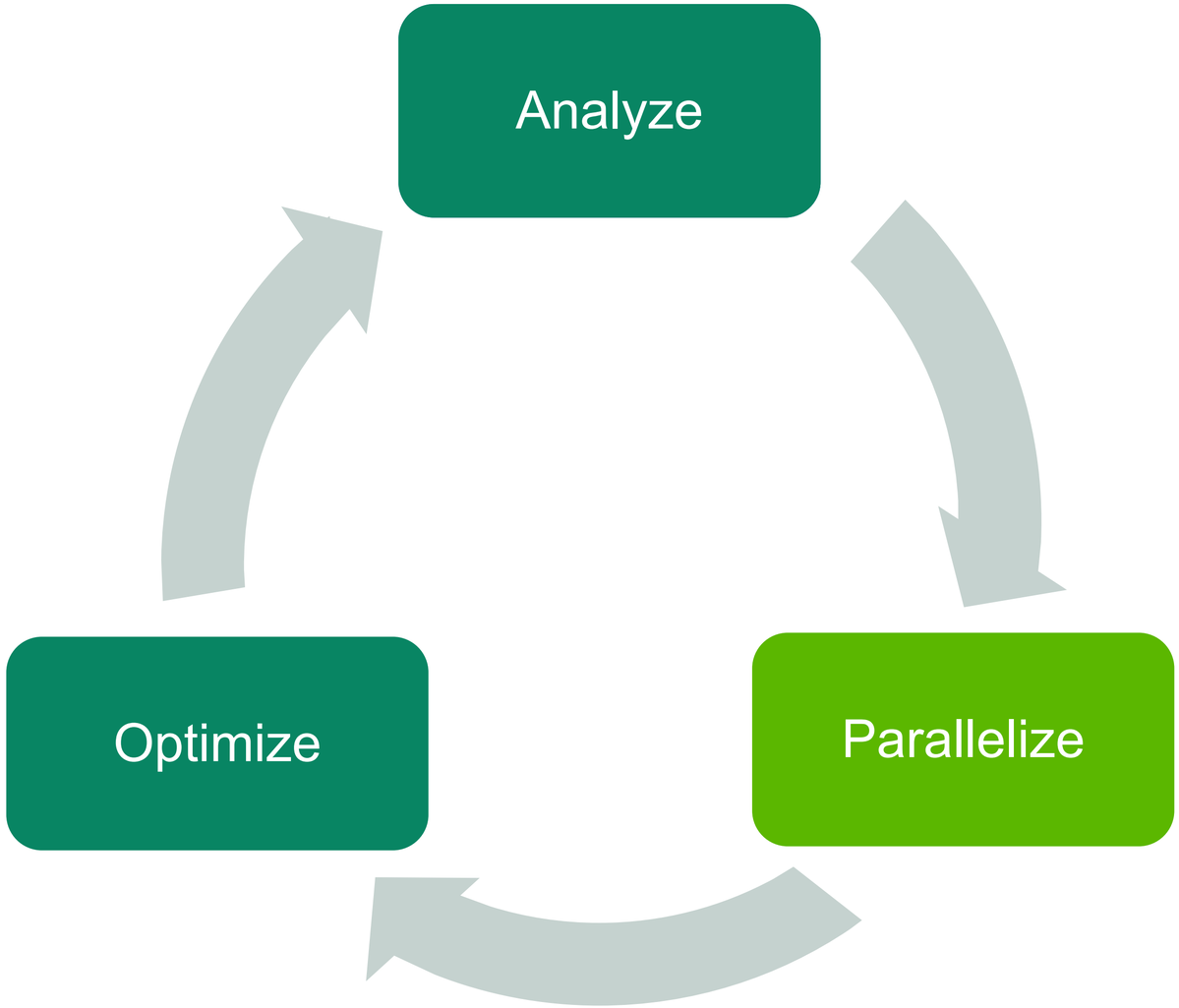
- Your code to determine most likely places needing parallelisation or optimisation

### Parallelise

- Your code by starting with the most time consuming parts, check the correctness and analysis again

### Optimize

- Your code to improve and observed speedup from parallelisation



# Compiling OpenX code with NVHPC

# Building and Running Enabled OpenACC Code

## The NVIDIA HPC SDK includes the new NVIDIA HPC compiler supporting OpenACC C and Fortran

- nvc -fast -gpu=target architecture -Minfo=accel -o laplace_2d laplace_2d.c

## Compiling OpenACC program using GCC

- gcc -fast -fopenacc -foffload=offload target -o laplace_2d laplace_2d.c

## Compiling OpenACC program using CRAY

- gcc -fast -f pragma=acc -h msgs -o laplace_2d laplace_2d.c

# Using OpenACC (Target Directive Support Since HPC SDK 23.1)

| Flags | Specification |
| --- | --- |
| -acc | enable OpenMP targeting device |
| -acc=host | to generate an executable that will run serially on the host CPU |
| -acc=multicore | parallelize for a multicore CPU |
| -acc=gpu –gpu=cc80 | map OpenACC parallelism to an NVIDIA GPU, compile targeting compute capability |
| –gpu=managed | place all allocatables in CUDA Unified Memory |
| –gpu=pinned | use CUDA pinned memory for all allocatables |
| -Minfo=acc?all | Compiler diagnostics for OpenACC |
| export NVCOMPILER_ACC_NOTIFY = 1\|2\|3 | Environment variable for NOTIFY |

```
$ nccx -acc -gpu=cc80,managed -Minfo=acc -o binary OpenACC_Code.c
```

# Using OpenACC (Target Directive Support Since HPC SDK 23.1)

| Flags | Specification |
| --- | --- |
| -mp | enable OpenMP targeting device |
| -mp=gpu | to generate an executable that will run serially on the host CPU |
| –gpu=cc80 | map OpenACC parallelism to an NVIDIA GPU, compile targeting compute capability |
| –gpu=managed | place all allocatables in CUDA Unified Memory |
| –gpu=pinned | use CUDA pinned memory for all allocatables |
| -Minfo=acc?all | Compiler diagnostics for OpenACC |
| export NVCOMPILER_ACC_NOTIFY = 1\|2\|3 | Environment variable for NOTIFY |

```
$ nccx -mp=gpu -gpu=cc80,managed -Minfo=mp -o binary OpenMP_Code.c
```

Checkpoint-1:  laplace2d_serial: parallelize with OpenACC

# Building and Running Enabled OpenACC Code

```
while ( error > tol && iter < iter_max ) {
  error=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]);
      error = max(error, 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++;
}
```

## Things to do

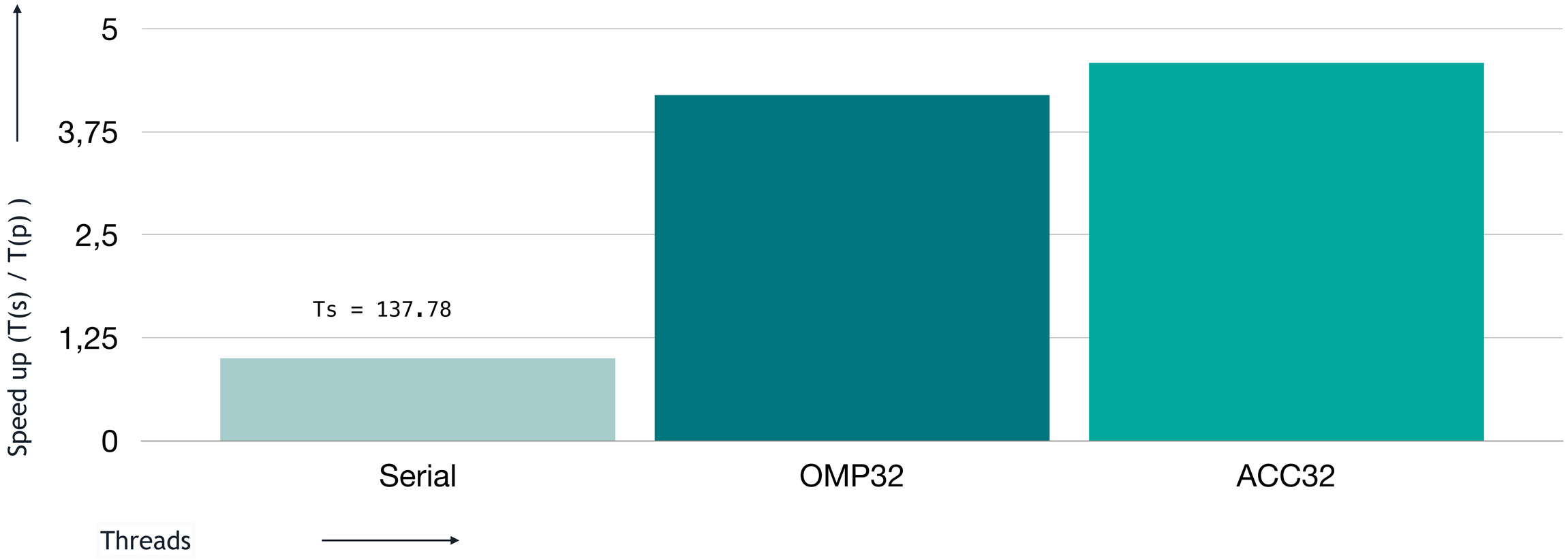
- Make these two loops parallel with OpenACC directives
- Add Parallel and Kernel construct
- Hint you might want to use reduction as well

# Compiling the same code sequentially and for multicore CPU

```
nshukla1 at lrdn2200 in /leonardo_scratch/large/userinternal/nshukla1/Sol/LaplaceC
$ nvc -fast -acc=multicore -Minfo=acc -o laplace_multi laplace2d.c
main:
     49, Generating Multicore code
         49, #pragma acc loop gang
     49, Generating reduction(+:error)
     51, Loop is parallelizable
     59, Generating Multicore code
         59, #pragma acc loop gang
```

# Performance Speed Up (Higher Is Better)

Simulation was performed 1000 Iterations on Leonardo



# Laplace2D_kernels_report

Set export PGI_ACC_TIME=1

```
main:
     34, Loop unrolled 8 times
     44, Loop not vectorized/parallelized: potential early exits
     49, Loop is parallelizable
         Generating implicit copyin(A[:][:]) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating implicit copyout(Anew[1:4094][1:4094]) [if not already present]
     50, Loop is parallelizable
         Generating Tesla code
         49, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating implicit reduction(max:error)
         50,   /* blockIdx.x threadIdx.x auto-collapsed */
     58, Loop is parallelizable
         Generating implicit copyin(Anew[1:4094][1:4094]) [if not already present]
         Generating implicit copyout(A[1:4094][1:4094]) [if not already present]
     59, Loop is parallelizable
         Generating Tesla code
         58, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         59,   /* blockIdx.x threadIdx.x auto-collapsed */
     69, FMA (fused multiply-add) instruction(s) generated
```

Implicit reduction

# Laplace2D_kernels_report

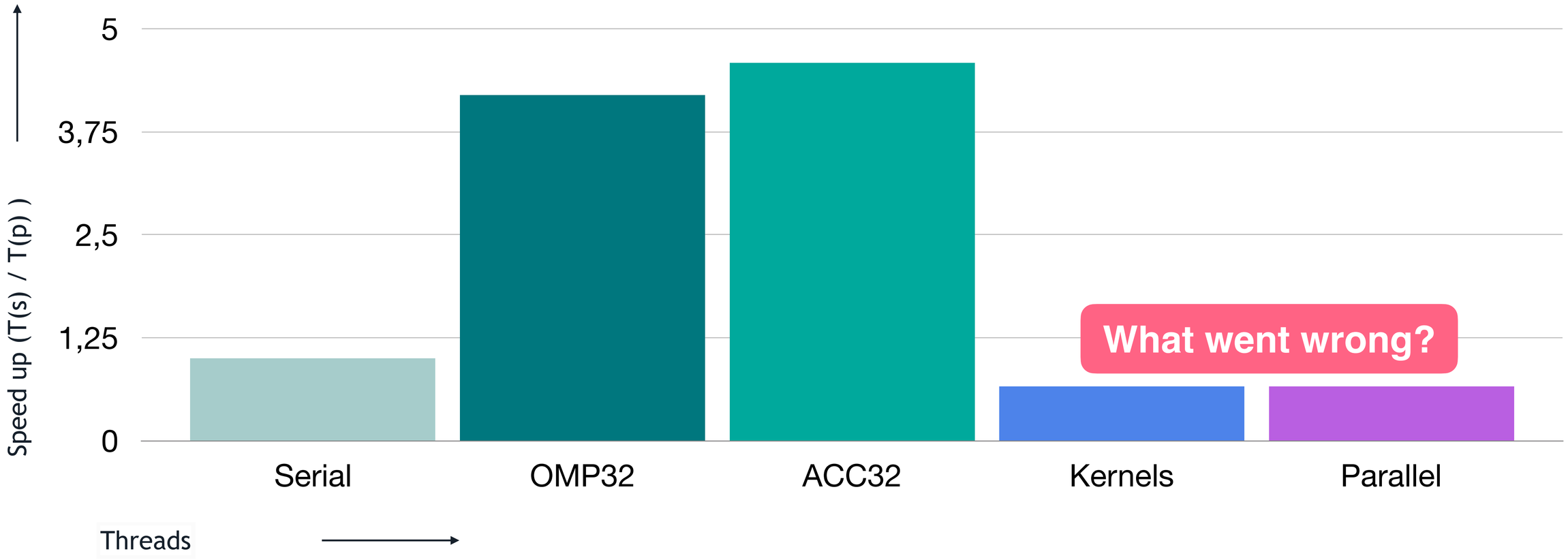
```
main:
     34, Loop unrolled 8 times
     44, Loop not vectorized/parallelized: potential early exits
     49, Loop is parallelizable
         Generating implicit copyin(A[:][:]) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating implicit copyout(Anew[1:4094][1:4094]) [if not already present]
     50, Loop is parallelizable
         Generating Tesla code
         49, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating reduction(max:error)
         50,   /* blockIdx.x threadIdx.x auto-collapsed */
     58, Loop is parallelizable
         Generating implicit copyin(Anew[1:4094][1:4094]) [if not already present]
         Generating implicit copyout(A[1:4094][1:4094]) [if not already present]
     59, Loop is parallelizable
         Generating Tesla code
         58, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         59,   /* blockIdx.x threadIdx.x auto-collapsed */
     69, FMA (fused multiply-add) instruction(s) generated
```

Compiler Generates reduction

Set NV_ACC_TIME=1 for lightweight profiler on time of data movements and kernels
NV_ACC_NOTIFY=1 gives a detailed breakdown of kernel launches and data transfers (bit field)

# Performance Speed Up (Higher Is Better)

Simulation was performed 1000 Iterations on Leonardo



Why do we have to care about data transfers?

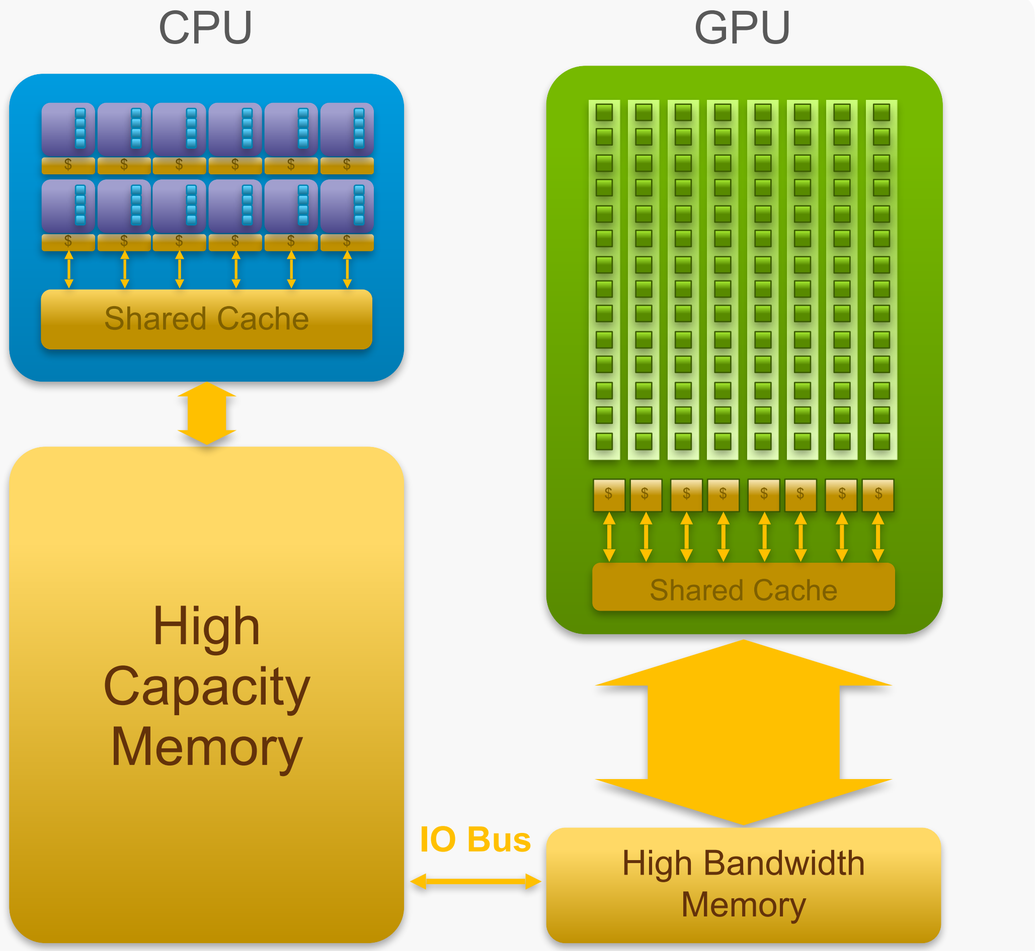
# Data Movement Is a Bottleneck in GPU Programming

## Between the host and device

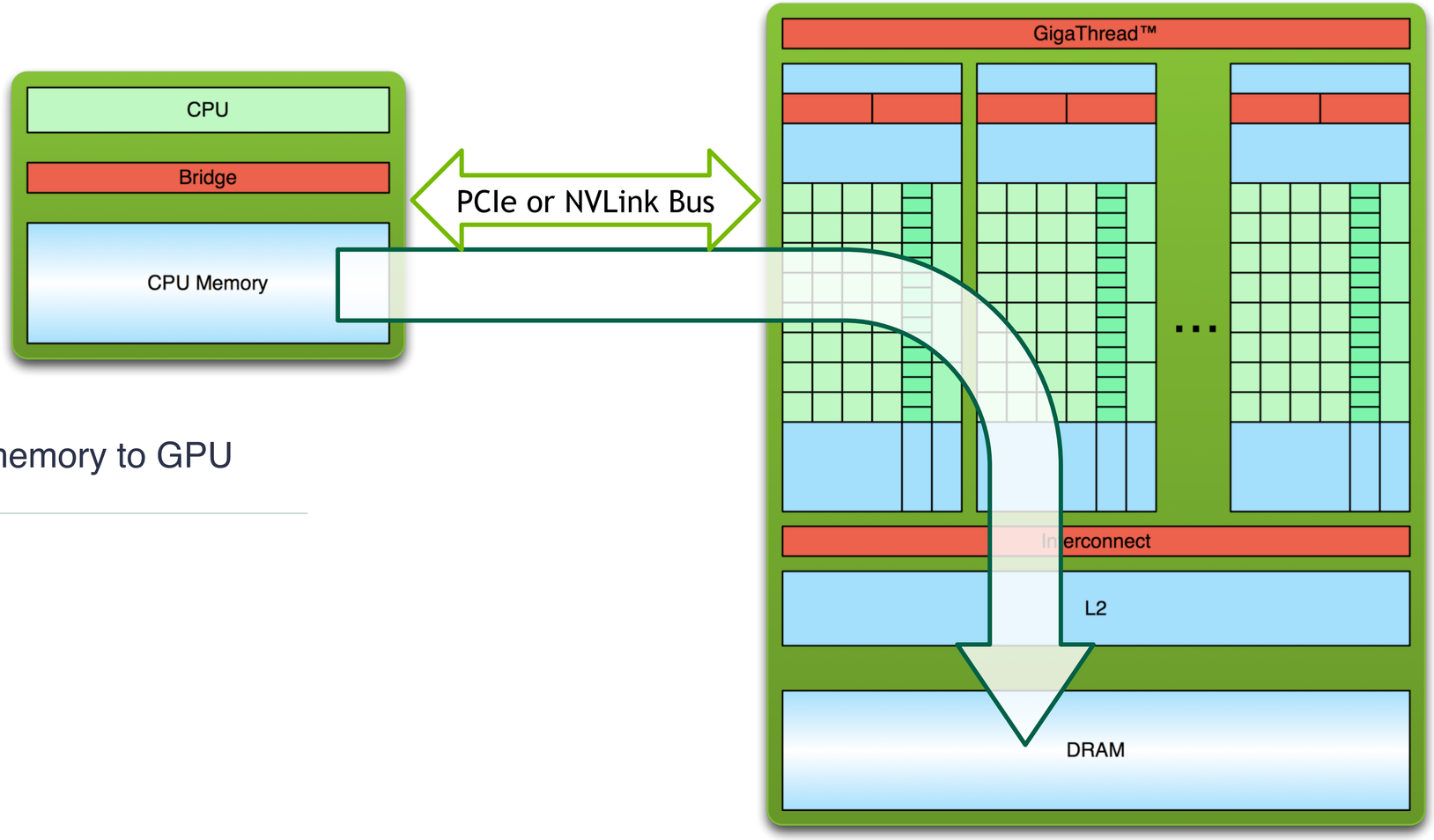
- GPU to its internal memory (HBM2): 900 GB/s
- GPU to CPU via PCIe: 16 GB/s
- GPU to GPU via NVLink: 25 GB/s
- CPU to RAM (DDR4): 128 GB/s

## Explicit memory management

- Data must be visible on the device when the kernel is launched
- To maximize performance, data movement needs to be minimise

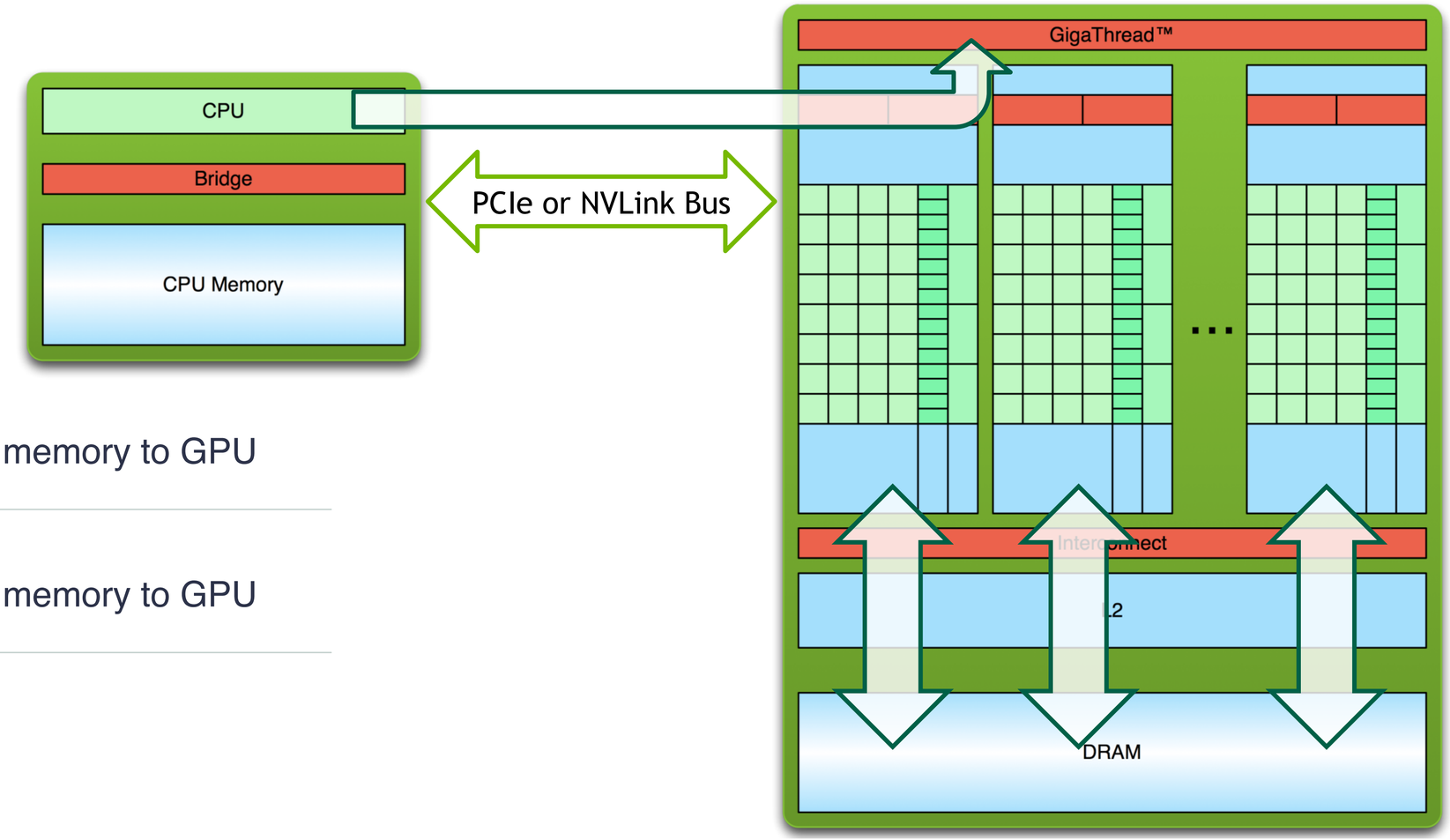


# Three Simple Processing Steps




1 Copy input data from CPU memory to GPU

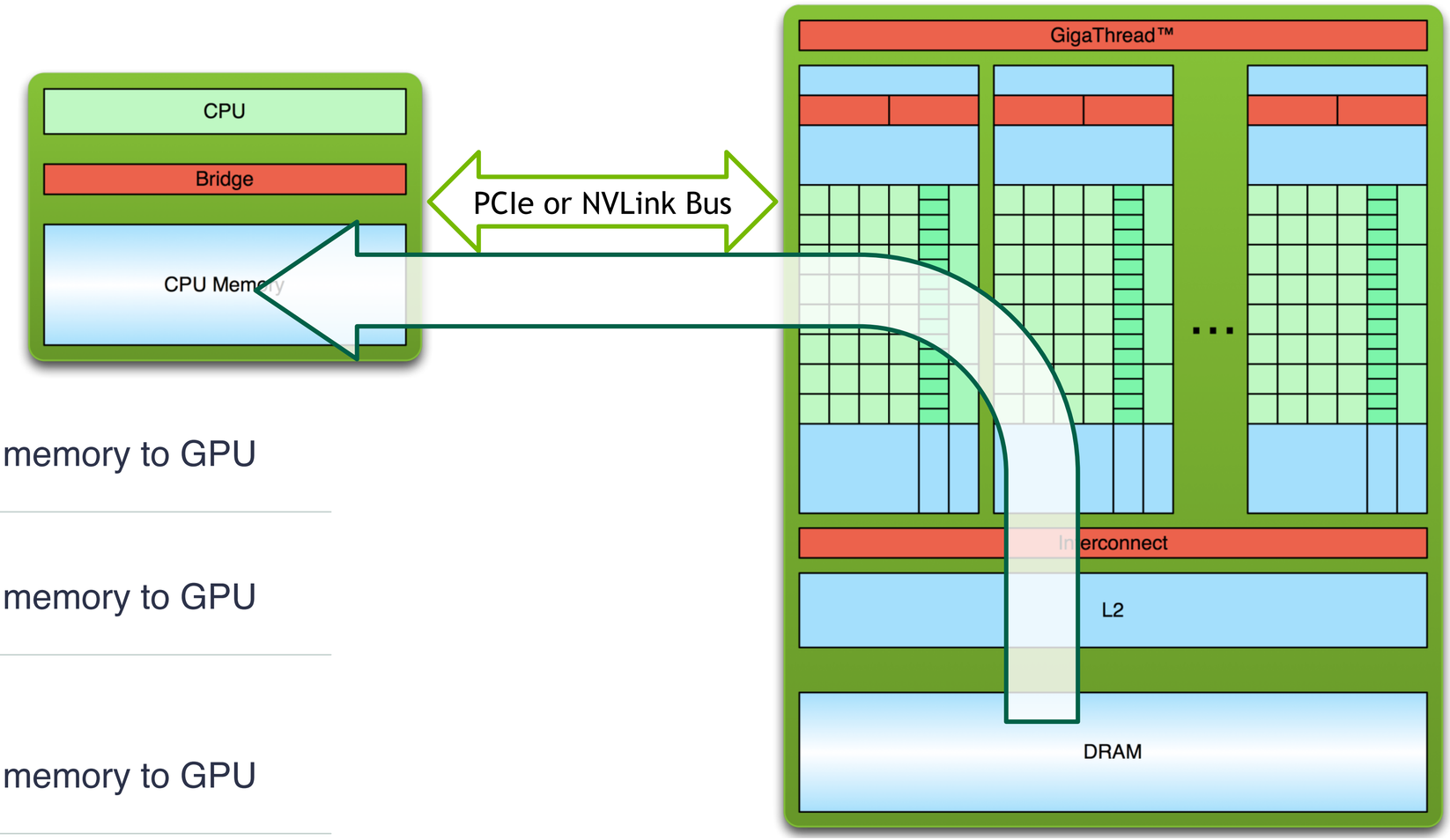
# Three Simple Processing Steps




1. Copy input data from CPU memory to GPU

2. Copy input data from CPU memory to GPU

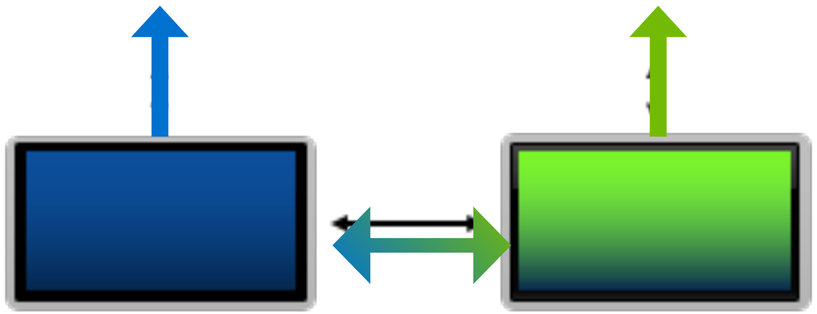
# Three Simple Processing Steps




1. Copy input data from CPU memory to GPU
2. Copy input data from CPU memory to GPU
3. Copy input data from CPU memory to GPU

# Managed Memory

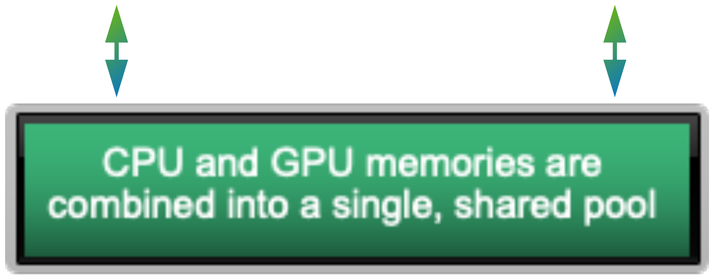
Without Managed memory



System memory GPU memory

$ nccx -gpu=cc80 -Minfo=acc -o binary Code.c

With Managed memory




Managed memory

$ nccx -gpu=cc80,managed -Minfo=acc -o binary Code.c

# OpenACC With Managed Memory

```
while ( error > tol && iter < iter_max )
{
  error = 0.0;
#pragma acc kernels
  {
    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]);
        error = fmax( error, fabs(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];
      }
    }
  }
}
```

Without Managed Memory the compiler must determine the size of A and Anew and copy their data to and from the GPU each iteration to ensure correctness

With Managed Memory the underlying runtime will move the data only when needed

GPU programming can be thought of as a two-step process

Compute Bound

- Exposing to the GPU environment

**Data Management**

- Data management directives

Explicit Data Transfers

# Data Management With OpenACC

1. Allocate ‘A’ on GPU
2. Copy From CPU to GPU
3. Copy back to GPU and deallocate A from GPU

```
int *A = (int*) malloc(N * sizeof(int));
#pragma acc parallel loop   copy(A[0:N])
{
  for (int i=0; i<N; i++) A[i] = 0;
}
```

```
for (int i=0; i<N; I++) A[i] = 0;
#pragma acc parallel loop copy(A[0:N])
{
    for (int i=0; i<N; I++) A[i] = 1;
}
```

```
#pragma acc parallel loop copyin(A[0:N]) copyout (B[0:N])
{
    for (int i=0; i<N; I++) A[i] = B[I];
}
```

# Managing the Data on the Device

**Data region constructs**

- A data region is the dynamic scope of a structured block associated with an implicit or explicit data construct.
- Facilities the sharing of data between multiple parallel regions (kernels, parallel, loop *etc*)
- Must start and end in the scope of the same function or subroutine – it's a structure construct

The data directive defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

Syntax:

**C:**

```
#pragma acc data [clause]
{
    code region ...
    including compute related pragmas
}
```

**Fortran:**

```
!$acc data
    code region ...
    including compute related pragmas
!$acc end data
```

# Managing the Data on the Device

provides the programmer additional control over how and when data is created on and copied to or from the device

### Clause: copyin / map(to)

- Copy the variable data to the device at the beginning of the region, and
- release the space on the device when done without copying the data back to the host

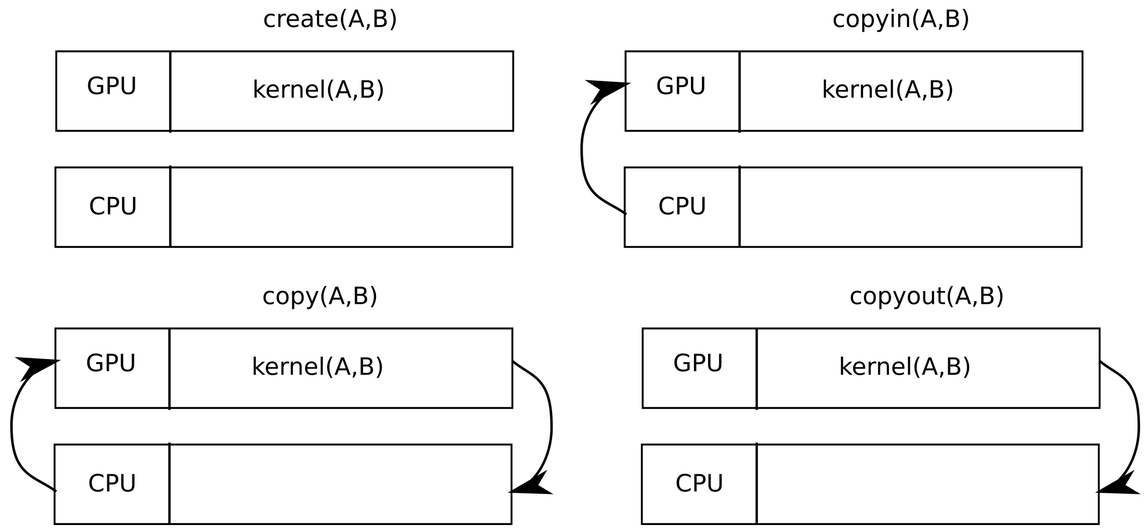
### Clause: copyout / map(from)

- Create space for the listed variables on the device but do not initialize them
- At the end of the region, copy the results back to the host and release the space on the device

### Clause: copy / map(from)

- Create space for the listed variables on the device, initialize the variable by copying data to the device at the beginning of the region
- copy the results back to the host at the end of the region, and release the space on the device when done

and create(list), delete(list), present(list), present_or_copy[inlout], present_or_create, device_ptr




**C and C++:**

```
#pragma omp target data [clause [[,] clause]…]
#pragma acc data [clause [[,] clause]…]
```

**Fortran:**

```
!$omp target data [clause [[,] clause]…]
!$acc data [clause [[,] clause]…]
```

# Data Clause: Moving Data Between Host to Device

used with the kernels, parallel, data and declare constructs

```
#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 |
| --- |
| [illegible] |


# Array Shaping

provides the programmer additional control over how and when data is created on and copied to or from the device

Compiler sometimes cannot determine size of arrays

- Must specify explicitly start/end point
- Memory only exits within the data region
- Must be within a single function

```
/* C/C++ code to offload on the device*/

    #pragma acc data copyin(a[0:N]) copyout(b[s/4:3*s/4])

/* Fortran code to offload on the device*/

    !$acc data copyin(a[1:N]) copyout(b(s/4:3*s/4))
```

- Fortran uses *start:end* and C uses *start:count*
- Data clauses can be used on data, kernels or parallel

Note: data clauses can be used on data, parallel, or kernels

# Encompassing Multiple Compute Regions

provides the programmer additional control over how and when data is created on and copied to or from the device

```
#pragma acc data copyin(A[0:N], B[0:N]) create(C[0:N])
{

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

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

}
```

```
void copy(int *A, int *B, int N)
{
#pragma acc parallel loop copyout(A[0:N]) copyin(B[0:N])
    for( int i = 0; i < N; i++ )
    {
        A[i] = B[i];
    }
}
```

```
#pragma acc data copyout(A[0:N],B[0:N]) copyin(C[0:N])
{
    copy(A, C, N);

    copy(A, B, N);
}
```

# Encompassing Multiple Compute Regions

provides the programmer additional control over how and when data is created on and copied to or from the device

## DAXPY in C

```
#pragma acc data create( D[0:ARRAY_SIZE], Y[0:ARRAY_SIZE] ) copyin(A) copyout(D[0:ARRAY_SIZE])
 {
#pragma acc parallel loop
 for (size_t i=0; i<ARRAY_SIZE; i++)
 {
        D[i] = 0.0;
        X[i] = 1.0;
        Y[i] = 2.0;
 }

 start_time = omp_get_wtime();
#pragma acc parallel loop
 for ( size_t i=0; i<ARRAY_SIZE; i++ )
        D[i] = A*X[i] + Y[i];
 end_time = omp_get_wtime();
 }
```

Compute region

Data region

Checkpoint-2:  laplace2d_parallel: data clause

# Data Clause

provides the programmer additional control over how and when data is created on and copied to or from the device

## Analyze the code and refactor the code by following these steps

- **Step1:** **Parallelize the code with**
  - Add parallel loop
- **Step2:** **Including data clause in our Laplace code**
  - Use acc data to minimize transfers
  - Add a **structured data directive** to properly handle the arrays **A** and **Anew**
    Run the Code (With Managed Memory)

## Try to understand the compiler report to be sure about what the compiler is doing

- nsys profile -t nvtx,openacc --stats=true --force-overwrite true -o laplace ./laplace

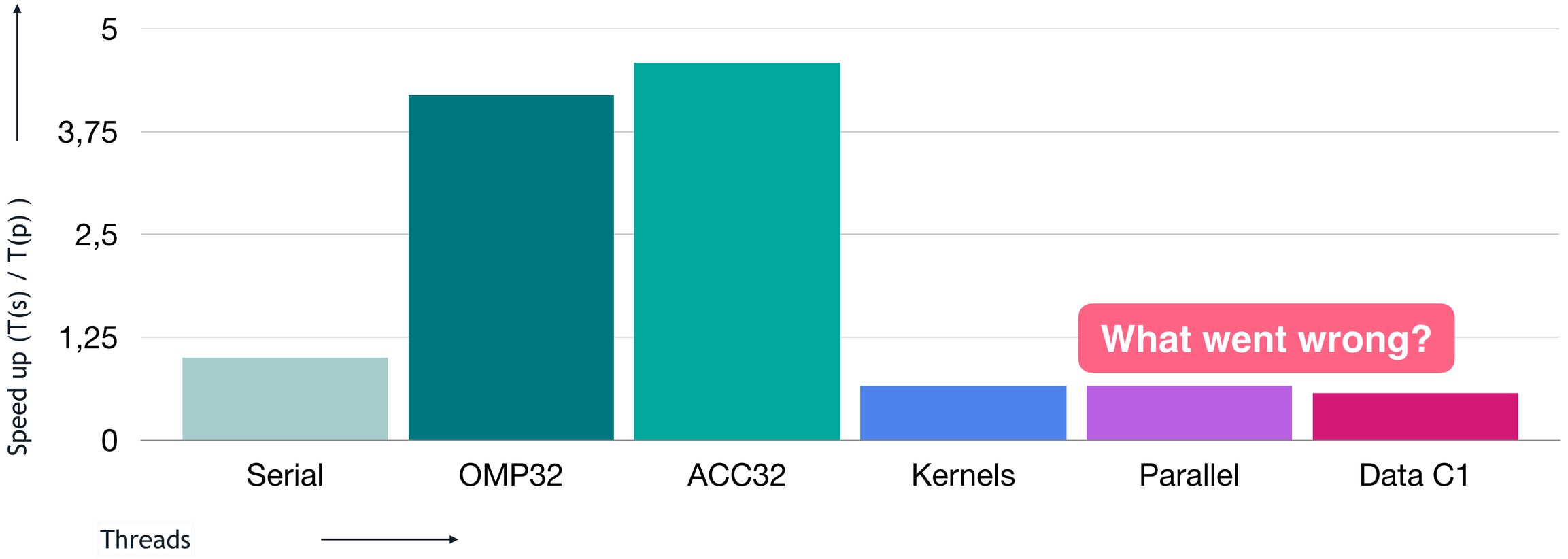
# 02-Laplace2D: Data Clause With OpenACC

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

#pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
  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 copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

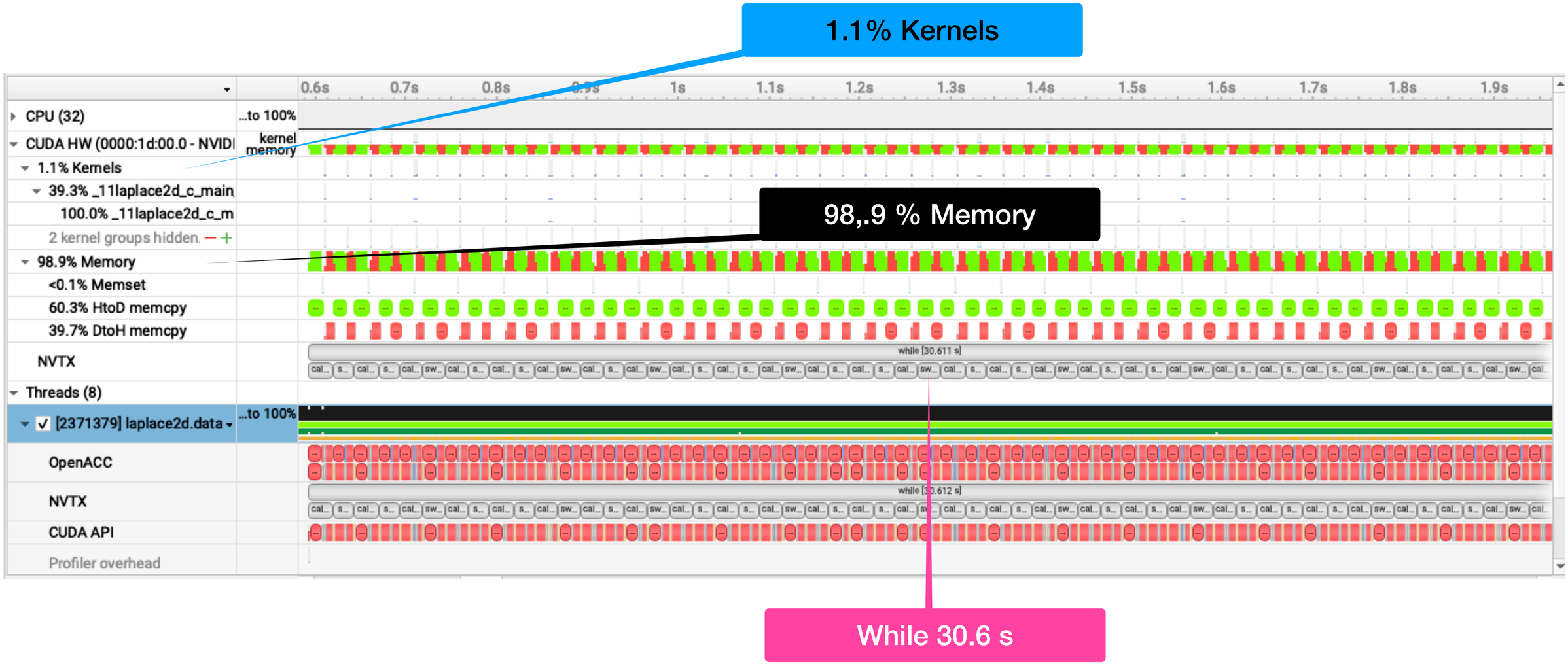
Data clauses provide necessary “shape” to the arrays.

# Performance ~~Speed Up~~ (Higher Is Better)

Simulation was performed 1000 Iterations on Leonardo



# NSIGHT: Recording an Application Timeline



# Excessive Data Transfers

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

A, Anew resident on host

COPY
HOST / DEVICE


```
#pragma acc parallel loop reduction(max:error)
```

A, Anew resident on accelerator

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

```
    ...
}
```

A, Anew resident on host

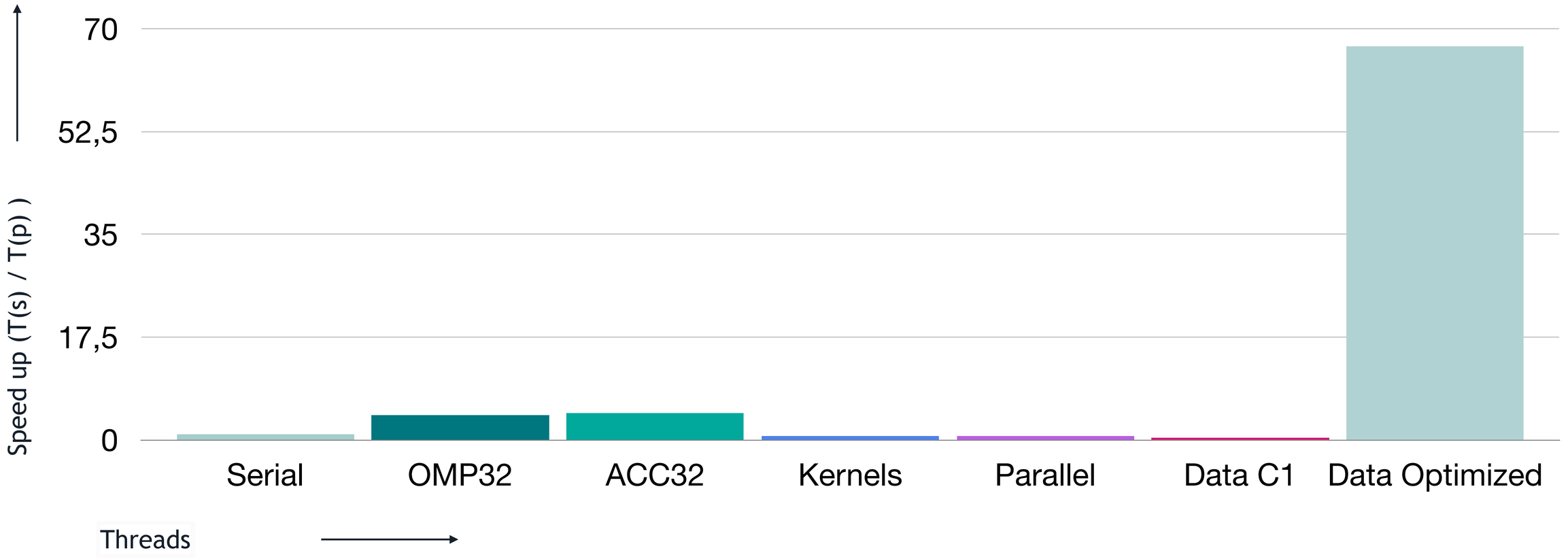

COPY
DEVICE / HOST

A, Anew resident on accelerator

These copies are performed every iteration of the while loop.
NB: in case of two #pragma acc parallel there are 4 copies per while loop iteration.

# Performance ~~Speed Up~~ (Higher Is Better)

Simulation was performed 1000 Iterations on Leonardo



# Advanced OpenACC

# Unstructured Data Directives

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));

    return ptr;
}

void deallocate(int *ptr)
{

    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

Data lifetime is not always restricted to a single routine

# Unstructured Data Directives

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));

    return ptr;
}

void deallocate(int *ptr)
{

    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

Data lifetime is not always restricted to a single routine

data is created

# Unstructured Data Directives

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));

    return ptr;
}

void deallocate(int *ptr)
{

    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

Data lifetime is not always restricted to a single routine

data is used

# Unstructured Data Directives

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));

    return ptr;
}

void deallocate(int *ptr)
{

    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

Data lifetime is not always restricted to a single routine

data is deallocated

# Unstructured Data Directives

## → UNSTRUCTURED DATA DIRECTIVES

```
#pragma acc enter data clauses

        < Sequential and/or Parallel code >

#pragma acc exit data clauses
```

- ✳ **`enter data`** to upload / create
- ✳ **`exit data`** to download / delete

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));

    return ptr;
}

void deallocate(int *ptr)
{

    free(ptr);
}

int main()
{
    int *ptr = allocate(100);


    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

# Unstructured Data Directives

## → UNSTRUCTURED DATA DIRECTIVES

```
#pragma acc enter data clauses

        < Sequential and/or Parallel code >

#pragma acc exit data clauses
```

- ✳ **`enter data`** to upload / create
- ✳ **`exit data`** to download / delete
- ✳ Can branch across multiple functions
- ✳ Do not open a data region

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));
    #pragma acc enter data create(ptr[0:size])
    return ptr;
}

void deallocate(int *ptr)
{
    #pragma acc exit data delete(ptr)
    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    #pragma acc parallel loop
    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

# Unstructured Data Directives

## CLAUSES

- `copyin ( list )` Allocates memory on device and copies data from host to device on enter data
- `copyout ( list)` Deallocate memory on device and copies data from device to host on exit data
- `create ( list )` Allocates memory on device without data transfer on enter data
- `delete ( list )` Deallocates memory on device without data transfer on exit data

```
#pragma acc enter data copyin(a[0:N],b[0:N]) create(c[0:N])

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

#pragma acc exit data copyout(c[0:N]) delete(a,b)
```

# Unstructured Data Directives

## Unstructured

- Can have multiple starting/ending points
- Can branch across multiple functions
- Memory exists until explicitly deallocated

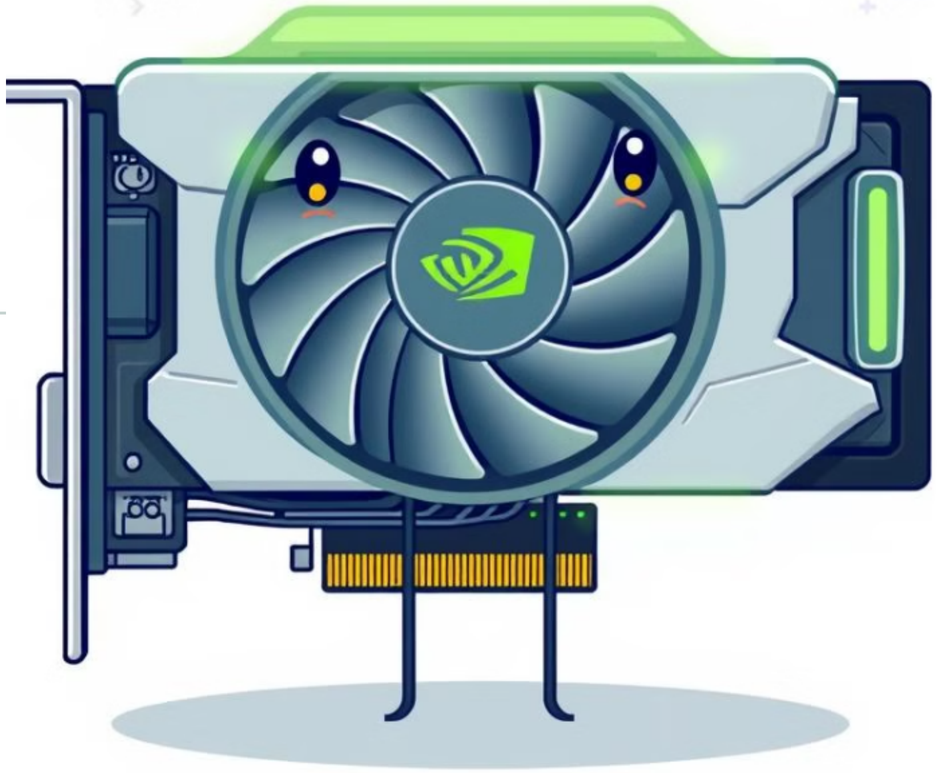
```
#pragma acc enter data copyin(a[0:N],b[0:N]) \
          create(c[0:N])

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

#pragma acc exit data copyout(c[0:N]) \
          delete(a,b)
```

## Structured

- Must have explicit start/end points
- Must be within a single function
- Memory only exists within the data region

```
#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];
          }

}
```

# 4 OpenACC Loop Optimization



# SEQ Clause

SEQ Clause: short for sequential

- the compiler will tell to run the loop
- Compiler will parallelise the outer loop across the parallel threads, but each thread will run the inner-most loop sequentially
- It may automatically apply the seq clause to loops that have too any dimensions

```
#pragma acc parallel loop
    for (int i = 0; i < size; I++)

      #pragma acc loop
         for (int j = 0; j < size; j++)

           #pragma acc loop seq
              for (int k = 0; k < size; k++)

                 c[i][j] += a[I][j] * b[I][j];
```

# SEQ Clause

The **`seq`** clause in the **`routine`** directive for OpenACC is used to indicate that the specified routine should be executed **sequentially in one device thread** (GPU).

At call the compiler needs to know

- Function will be available on the GPU ( `!$acc routine` )
- It is a sequential routine, executed by one device thread ( `seq` )

At call site

- Function is called in a parallel loop region ( `parallel loop` )
- Each thread in the loop will call it and execute its own instance

```
module b1
 contains
  real function sqab(a)
     !$acc routine seq
     real :: a
     sqab = sqrt(abs(a))
  end function
 end module


subroutine test( x, n )
     use b1
     real, dimension(*) :: x
     integer :: n
     integer :: i
     !$acc parallel loop copy(x(1:n))
     do i = 1, n
        x(i) = sqab(x(i))
     enddo
 end subroutine
```

# Private and Firstprivate Clause

The Private Clause: allows the programmer to define a list of variables as “thread-private”

- Each thread will be given a private copy of every variable in comma-separated list

The Firstprivate Clause: like private except

- The private values are initialised to the same value used on the host. Private values are uninitizlised

```
double tmp[3]
 #pragma acc kernels loop privarte(tmp[0:3]
    for (int i = 0; i < size; i++)
       tmp[0] += <value>;
       tmp[1] += <value>;
       tmp[2] += <value>;
#pragma acc parallel loop
    for (int j = 0; j < size; I++)
          array[I][j] = tmp[0] + tmp[1] + tmp[2]
```

tmp array is private to each iteration of the outer loop

Shared among the threads in the inner loop

# Scalars and Private Clause

By default, scalars are firstprivate when used in a parallel region and private when used in a kernels region

Except in some cases, scalars do not need to be added to a private clause. These cases may include but are not limited to:

- Scalars with global storage such as global variables in C/C++, Module variables in Fortran
- When the scalar is passed by reference to a device subroutine
- When the scalar used as an rvalue after the compute region, aka “live-out”

Note: putting scalars in a private clause may actually hurt performance !

# Collapse Clause

- Collapse(N)
- Combine the next N tightly nested loops
- Can turn a multidimensional loop nest into a single-dimension loop
- Extremely useful for increasing memory locality, as well as creating larger loop to expose more parallelism

```
#pragma acc parallel loop collapse(2)
for ( int i = 0; i < N; i++)
    for ( int j = 0; j < N; j++)
        structured-block
```

# The Tile Clause

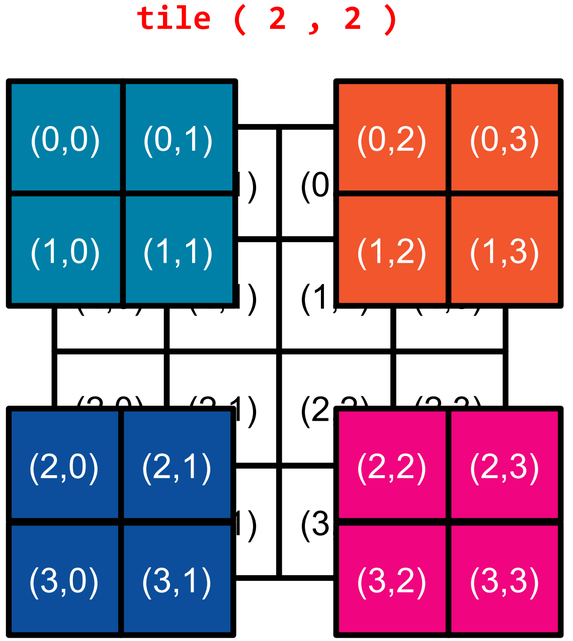
**tile( x, y, z . . . .)**

- Breaks multidimensional loop into “blocks ”
- Can increase data locality in some codes
- Will be able to execute multiple tiles simultaneously

```
#pragma acc kernels loop tile(32,32)
for ( int i = 0; i < 128; i++)
    for ( int i = 0; i < 128; i++)
     structured-block
```

Similar to the `collapse` clause, the inner loops should not have the `loop` directive.

```
#pragma acc kernels loop tile(32,32)
for ( int i = 0; i < 128; I++)
     #pragma acc loop
    for ( int i = 0; i < 128; i++)
```



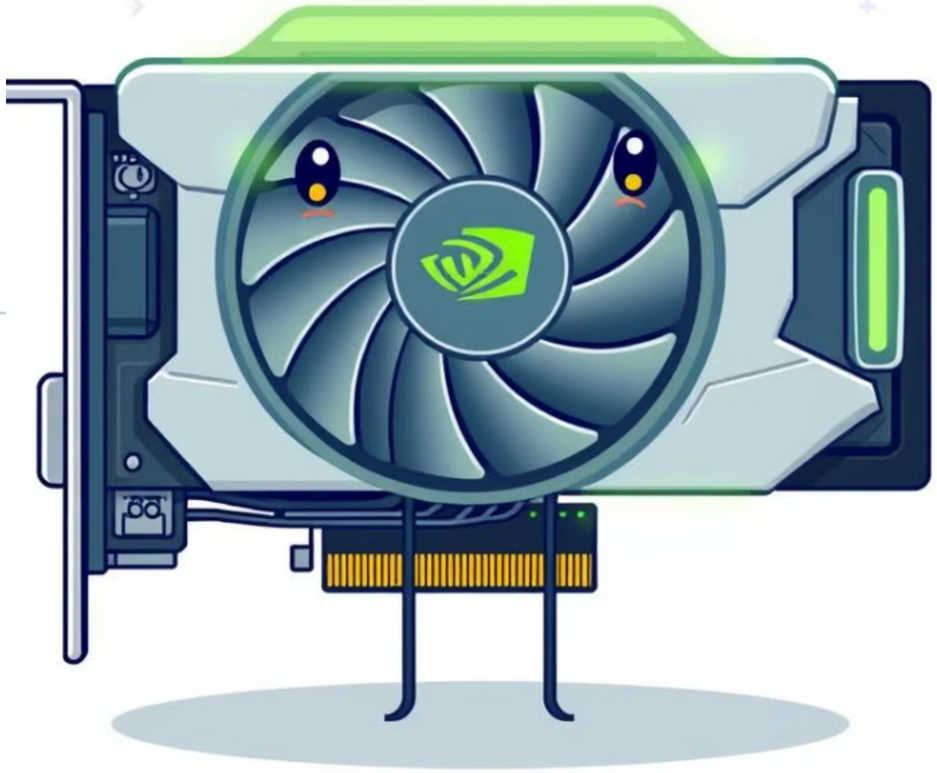
# Reduction Clause

- Takes many values and “reduces” to a single value
- Each thread calculates its part
- Perform a partial reduction on the loop iterations the they compute
- After the loop, the compiler will perform a final reduction to produce a single result using the specified operation

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

| Operator | Example | Description |
|---|---|---|
| + | reduction(+:sum) | Mathematical summation |
| * | reduction(*:product) | Mathematical product |
| max | reduction(max:maximum) | Maximum value |
| min | reduction(min:minimum) | Minimum value |
| & | reduction(&:val) | Bitwise AND |
| \| | reduction(\|:val) | Bitwise OR |
| && | reduction(&&:bool) | Logical AND |
| \|\| | reduction(\|\|:bool) | Logical OR |

# 4 GPU hardware hierarchy



# Execution Model: Three Levels of Parallelism

| PLATFORM | GANG | WORKER | VECTOR |
|---|---|---|---|
| MULTICORE CPU | Entire CPU (NUMA domain) | Core | SIMD vector |
| MANYCORE CPU (e.g. Xeon Phi) | NUMA domain (whole chip) | Core | SIMD vector |
| NVIDIA GPU | Thread block | WARP | Thread |
| AMD GPU | Workgroup | Wavefront | Thread |

Number of threads in a gang $N_{threads} = L_{vector} \times N_{workers}$

# A Simplified Representation of a NVIDIA GPU-Architecture

## Peak performance NVIDIA A100

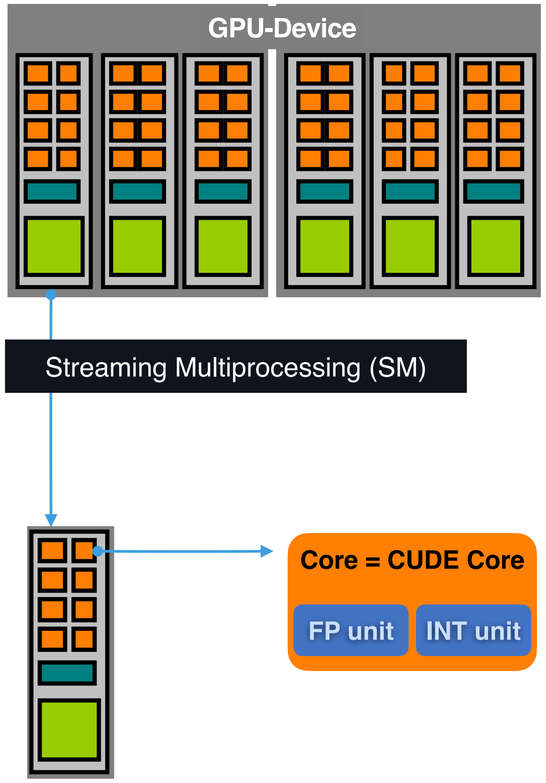
**FLOPS = Clock speed × cores × FLOP/cycle**

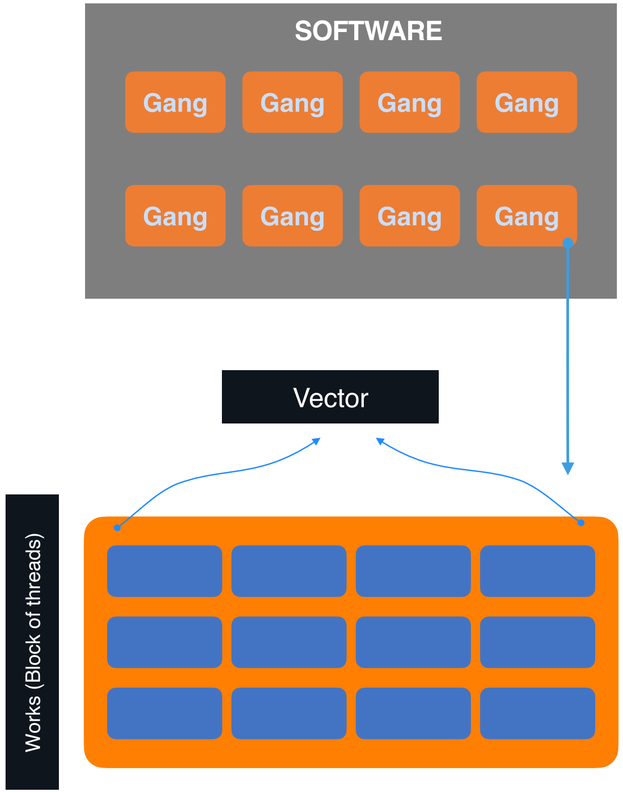
**FLOPS <=> FP64 or FP32 or ...**

**Clock speed (or GPU Boost Clock) = 1.41 GHz**

**Total FLOPS = 250.0 TFLOPS**

(Thread ∈ Block ∈ Grid)



# Gang Worker Vector Demystified

OpenACC
More Science, Less Programming

NVIDIA

# Gang Worker Vector Demystified

How much work 1 worker can do is limited by his speed.

A single worker can only move so fast.

# Gang Worker Vector Demystified

Even if we increase the size of his roller, he can only paint so fast.

We need more workers!

# Gang Worker Vector Demystified

OpenACC More Science, Less Programming

NVIDIA

# Gang Worker Vector Demystified

By organizing our workers into groups (gangs), they can effectively work together within a floor.

Groups (gangs) on different floors can operate independently.

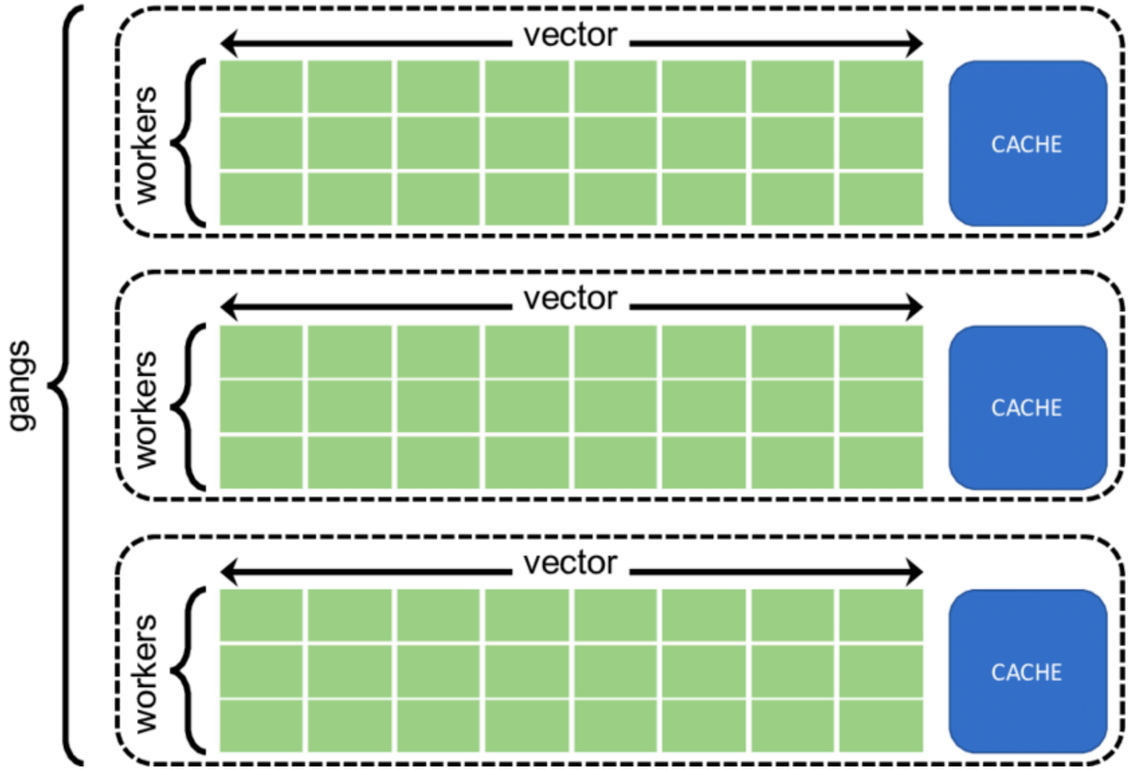
Since gangs operate independently, we can use as many or few as we need.

# Gang Worker Vector Demystified

Our painter is like an OpenACC ***worker***, he can only do so much.

His roller is like a ***vector***, he can move faster by covering more wall at once.

Eventually we need more workers, which can be organized into ***gangs*** to get more done.




OpenACC More Science, Less Programming

NVIDIA

# Gang Worker Vector Demystified

## Gang

- Multiple gangs will be generated, and loops iterations will be spread across the gangs
- Gangs are independent of each other
- There is no way to for the programmer to know exactly how many gangs are running at a given time

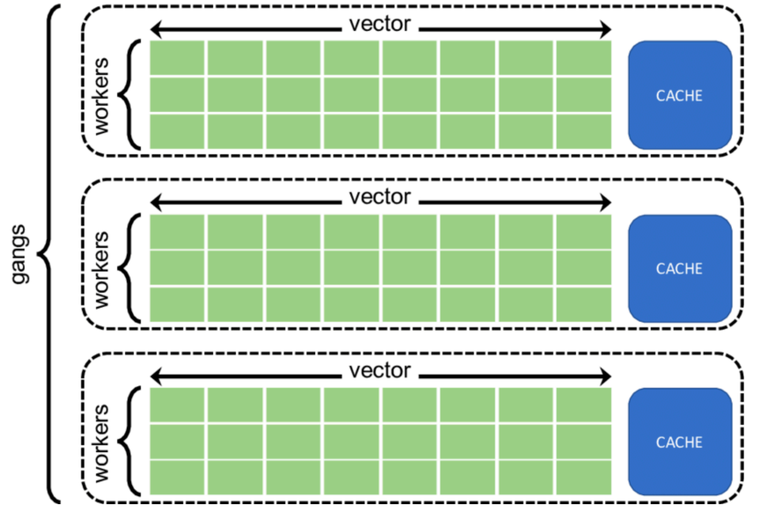
## Worker

- To have **multiple vectors** within a gang
- Splits up one large vector into multiple smaller vectors
- Intermediate level  between the low-level parallelism implemented in vector and group of threads
- useful when our inner parallel loops are very small, and will not benefit from having a large vector

## Vector parallelism:

- Lowest level of parallelism
- Every gang will have at least 1 vector
- Threads work in lockstep (SIMD/SIMT parallelism)

```
#pragma acc loop gang
for ( int i = 0; i < N; i++)
 #pragma acc loop worker
  for ( int j = 0; j < N; j++)
  #pragma acc loop vector
   for ( int k = 0; k < N; k++)
          structured-block
```



# Controlling the Size of Gang, Worker and Vectors

The compiler will choose a number of gangs, workers, and a vector length for you, but you can change it with clauses

num_gangs(N)

- Generate N gangs for this parallel region

num_workers(M)

- Generate N gangs for this parallel region

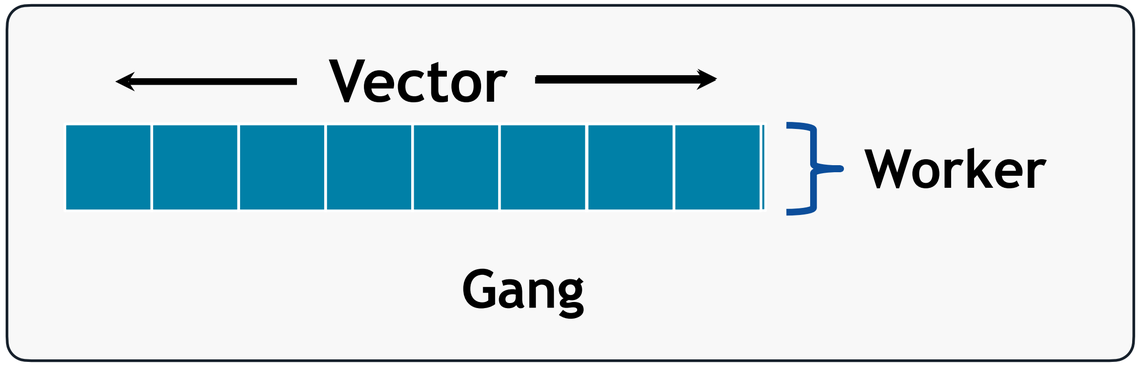
vector_length((P):

- Use a vector length of P for this parallel region

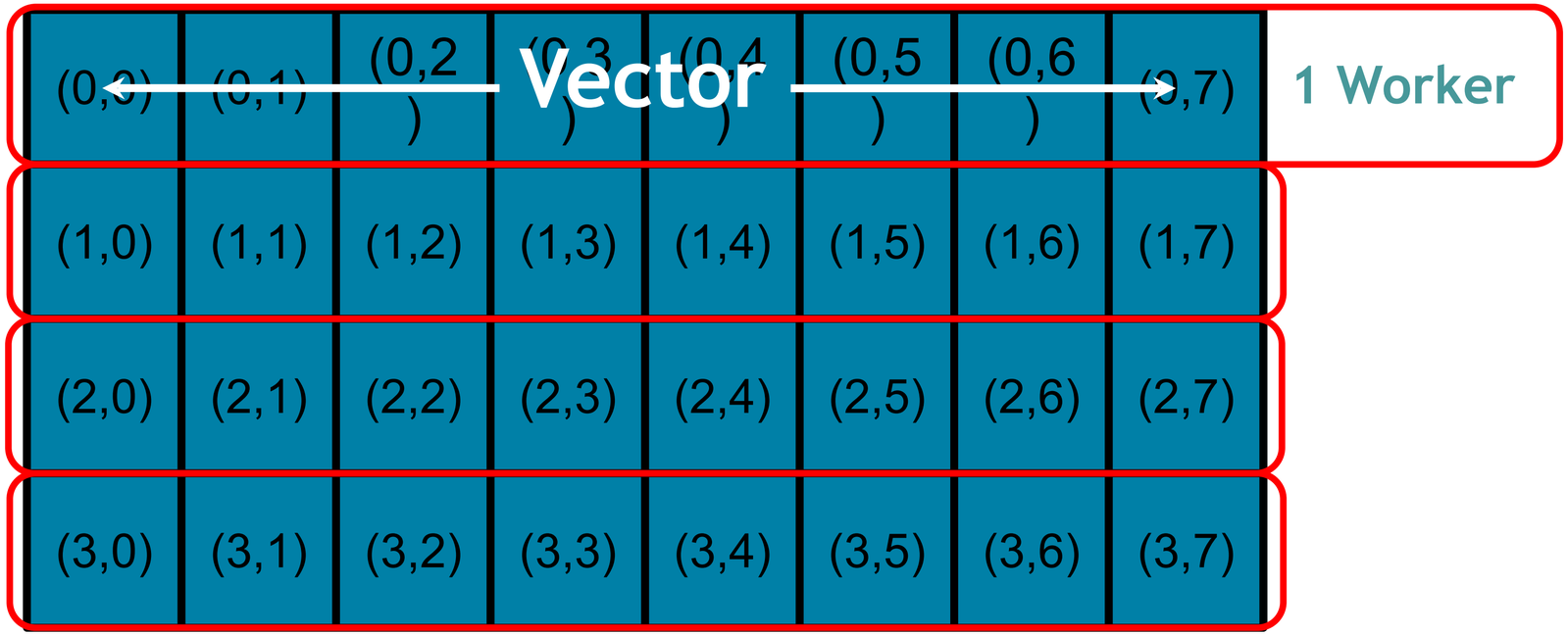
```
#pragma acc paralle nums_gangs(2) num_workers(4) vector_length(32)
 {
  #pragma acc loop worker
   for ( int i = 0; i < N; i++)
     #pragma acc loop vector
     for ( int j = 0; j < N; j++)
            structured-block
 }
```

Rule of 32: general rule of thumb for programming for NVIDIA GPUs is to always ensure that your vector length is a multiple of 32 (which means 32, 64, 96, 128, … 512, … 1024… etc.)

# Gang Worker Vector Demystified



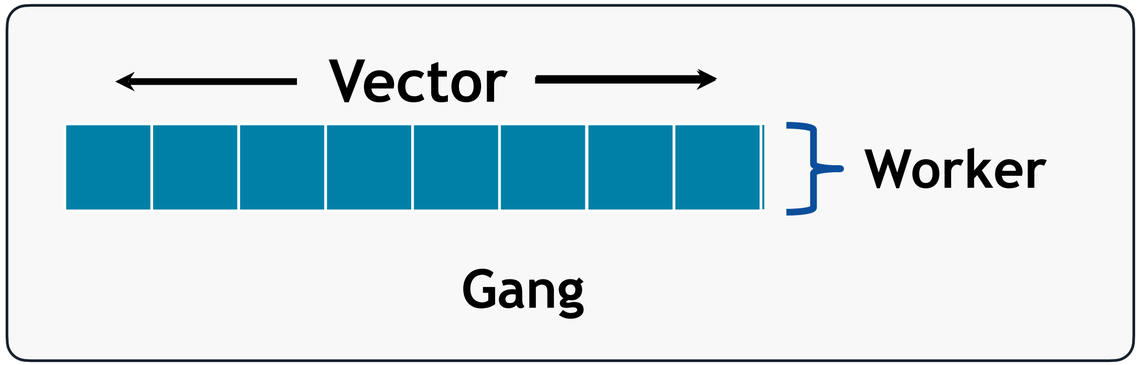

```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(8)
    for(int y = 0; y < 8; y++){
        array[x][y]++;
     }
}
```



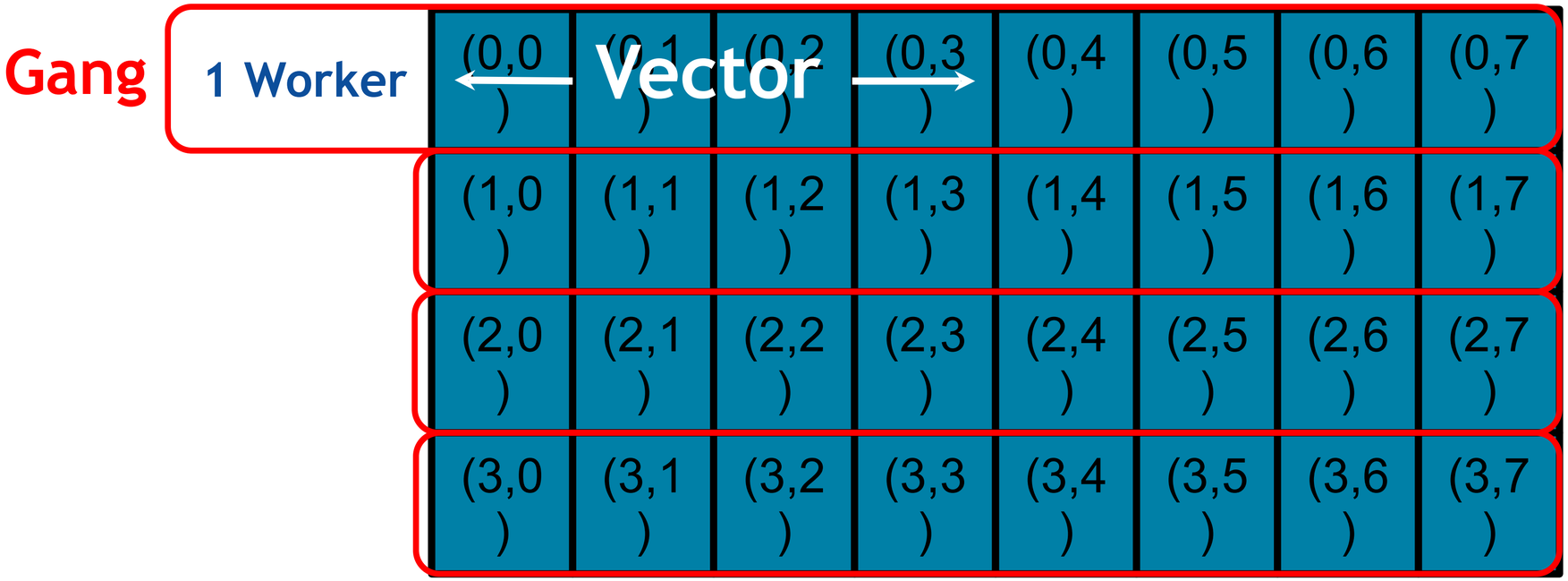

## Gang

- The vectors are colored, so that we can observe which loop iterations they are being applied to
- Based on the size of this loop nest, the compiler will (theoretically) generate **4 gangs**

# Gang Worker Vector Demystified

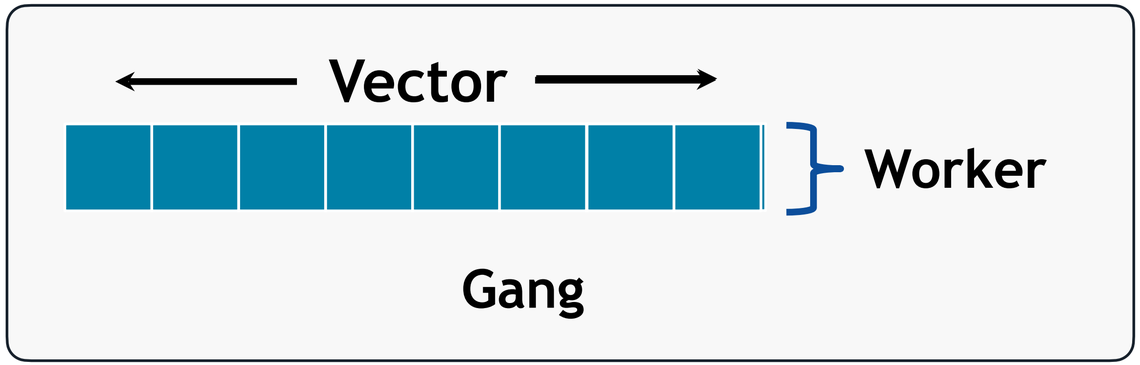



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(4)
    for(int y = 0; y < 8; y++){
        array[x][y]++;
     }
}
```

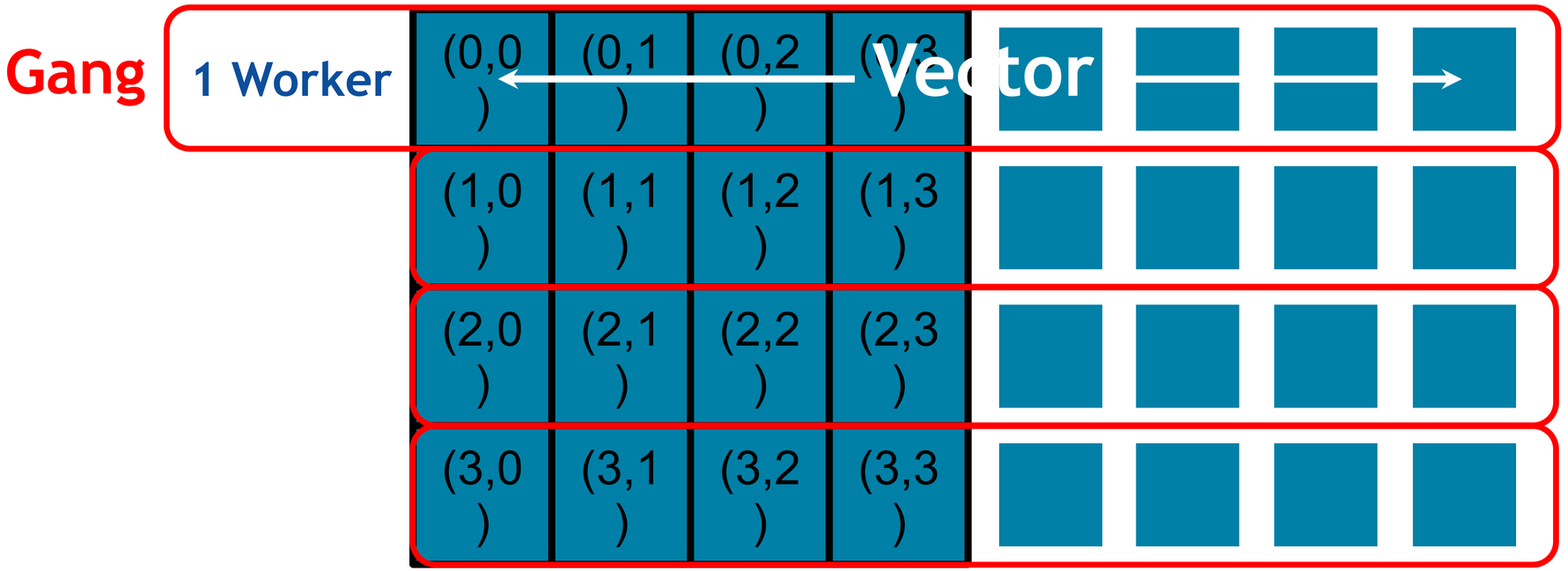



- We are still generating 4 gangs, but now each vector is computing two loop iterations
- If we wanted to generate **more gangs**, we would need to increase the size of the outer-loop

# Gang Worker Vector Demystified

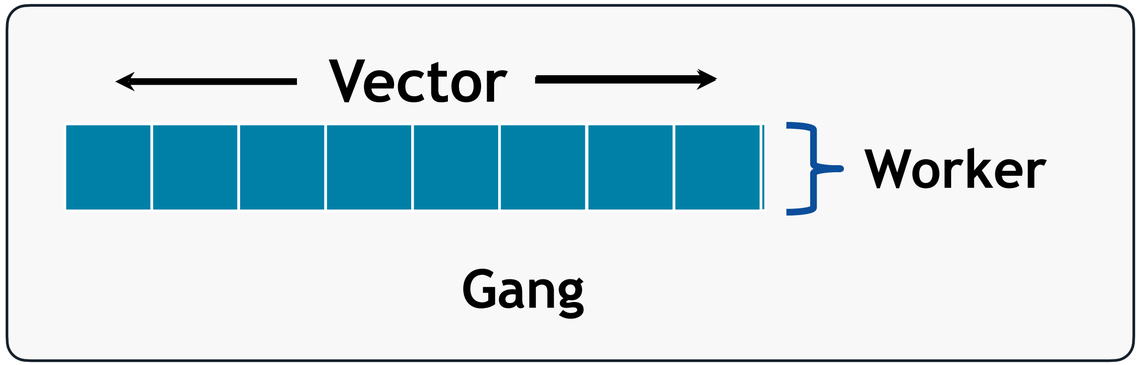



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(8)
    for(int y = 0; y < 4; y++){
        array[x][y]++;
     }
}
```



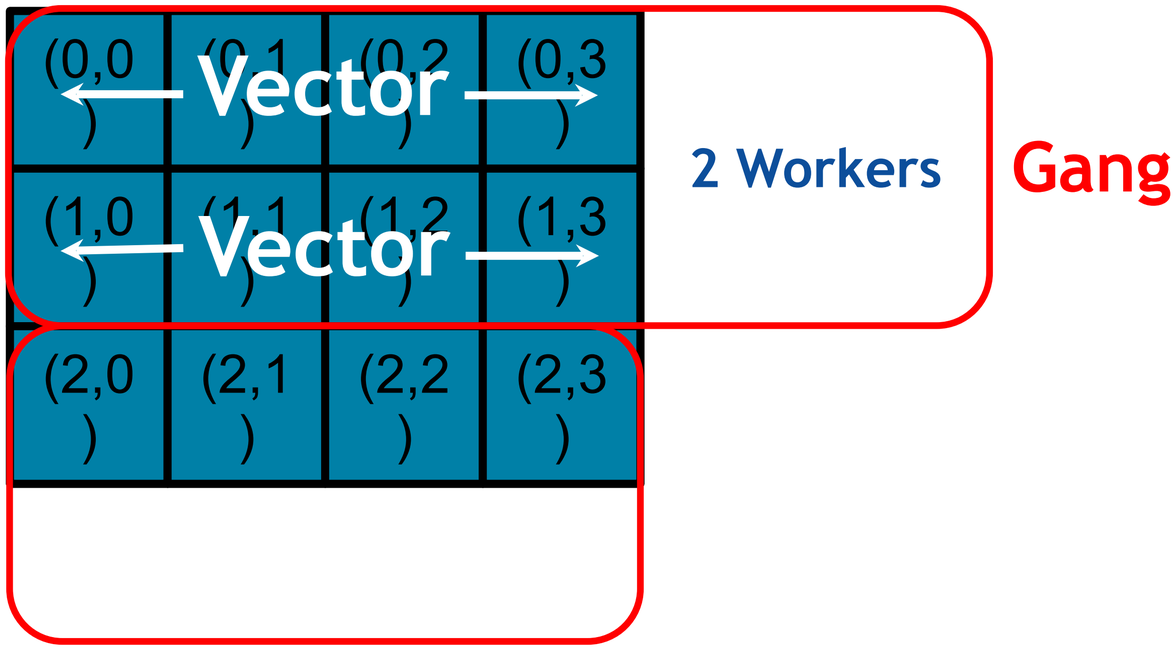

- We can see that our vector length is **much larger** than our inner-loop
- We are **wasting** half of our vector, meaning our code is performing half as well as it could

# Gang Worker Vector Demystified




We can fix this by **breaking our vector** up among **2 workers**

```
#pragma acc kernels loop gang worker(2)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(4)
    for(int y = 0; y < 4; y++){
        array[x][y]++; }
}
```




- We are no longer wasting a portion of our vectors, since the smaller vector size now fits our loop properly
- We always need to consider the size of the loop when choosing the gang worker vector dimensions

# Final exercise:  laplace2d_parallel

# Optimize the laplace2d_OpenACC

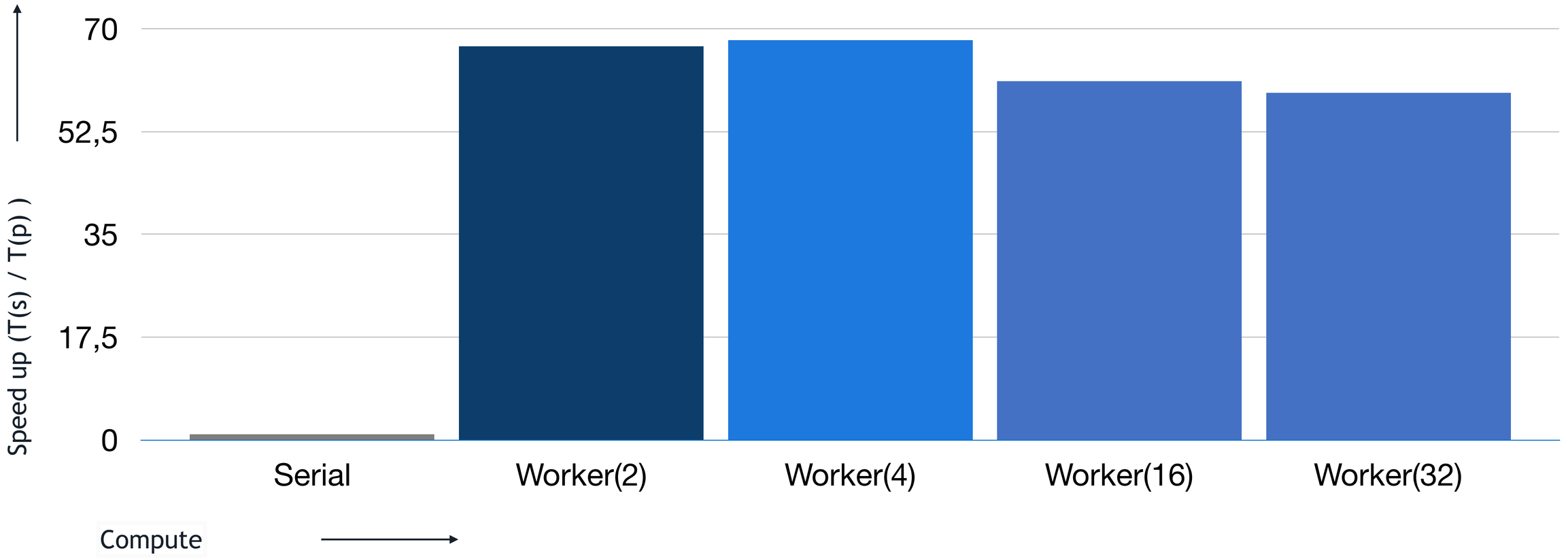
**Based on the knowledge that you had optimise the Laplace code**

**Beat this time = 0.60 s**

Try to understand the compiler report to be sure about what the compiler is doing

- nsys profile -t nvtx,openacc --stats=true --force-overwrite true -o laplace ./laplace

# Performance Speed Up (Higher Is Better)




# Transition towards to OpenMP offloading?

# OpenX (X = OMP, ACC) directives that are similar in nature

- OpenMP and OpenACC have similar coding paradigms under the hood
- The developer can instruct the compiler which levels of parallelism to use on given loops by adding clauses
- Applicable to a variety of hardware, but we fill focus a little bit on a GPU specific implementation

| OpenACC | CUDA | Mapping to NVIDIA GPU | OpenMP |
|---|---|---|---|
| Parallel/Kernel | Kernel | GPU | Parallel |
| Gang | Thread block | SMs | Team |
| Worker | Thread | SP or Compute unit | Thread |
| Vector | Warp (32 threads) | 32-wide thread | SIMD |

# OpenX (X = OMP, ACC) directives that are similar in nature

- In theory (according to the standards) the implementation of the levels adapt to the hardware, but in reality some compilers struggle with certain parallelisation levels

| OpenACC | OpenMP |
|---|---|
| acc parallel | omp target teams |
| acc loop gang | omp distribute |
| acc loop worker | omp parallel loop |
| acc loop vector | omp simd |
| acc declare | omp declare target |
| acc data | omp target data |
| acc update | omp target update |
| copy/copy_in/copy_out | map(tofrom/to/from:...) |

# Parallel: Similar but different

## OMP Parallel

- Creates a team of threads
- Very well-defined how the number of threads is chosen
- May synchronize within the team
- Data races are the the user's responsibility

## ACC Parallel

- Creates 1 or more gangs to workers
- Compiler free to choose number of gangs, workers, vector length
- May not synchronize between gangs
- Data races not allowed

# Loop: Similar but different

## OMP Loop (For/Do/

- Splits (“Workshares”) the iterations of the next loop to threads in the team, guarantees the user has managed any data races
- Loop will be run over threads and scheduling of loop iterations may restrict the compiler

## ACC Loop

- Declares the loop iterations as independent & race free (parallel) or interesting & should be analyzed (kernels)
- User able to declare independence w/o declaring scheduling
- Compiler free to schedule with gangs/workers/vector, unless overridden by user

# Distribute vs Loop

## OMP Loop (For/Do/

- Must live in a TEAMS region
- Distributes loop iterations over 1 or more thread teams
- Only master thread of each team runs iterations, until PARALLEL is encountered
- Loop iterations are implicitly independent, but some compiler optimizations still restricted

## ACC Loop

- Declares the loop iterations as independent & race free (parallel) or interesting & should be analyzed (kernels)
- Compiler free to schedule with gangs/workers/vector, unless overridden by user

# Distribute example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i++)
    for(j=0;j<m;j++)
      for(k=0;k<p;k++)
}
```

Generate a 1 or more thread teams

Distribute “i” over teams.

No information about “j” or “k” loops

```
#pragma acc parallel

#pragma acc loop
=0; i<n; i++)
#pragma acc loop
(j=0;j<m;j++)
#pragma acc loop
        for(k=0;k<p;k++)
}
```

# Distribute example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i
    for(j=0;j<m;j
      for(k=0;k<p;k++)
}
```

Generate a 1 or more gangs

These loops are independent, do the *right thing*

```
#pragma acc parallel
{
#pragma acc loop
  for(i=0; i<n; i++)
#pragma acc loop
    for(j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

# Synchronisation

## OpenMP

- Users may use barriers, critical regions, and/or locks to protect data races
- It’s possible to parallelize non-parallel code

## OpenACC

- Users expected to refactor code to remove data races.
- Code should be made truly parallel and scalable

# Synchronisation Example

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
#pragma omp critical
  A[i] = rand();
  A[i] *= 2;
}
```

```
parallelRand(A);
#pragma acc parallel loop
for (i=0; i<N; i++)
{
  A[i] *= 2;
}
```

# Laplace Heat: OpenMP offloadings

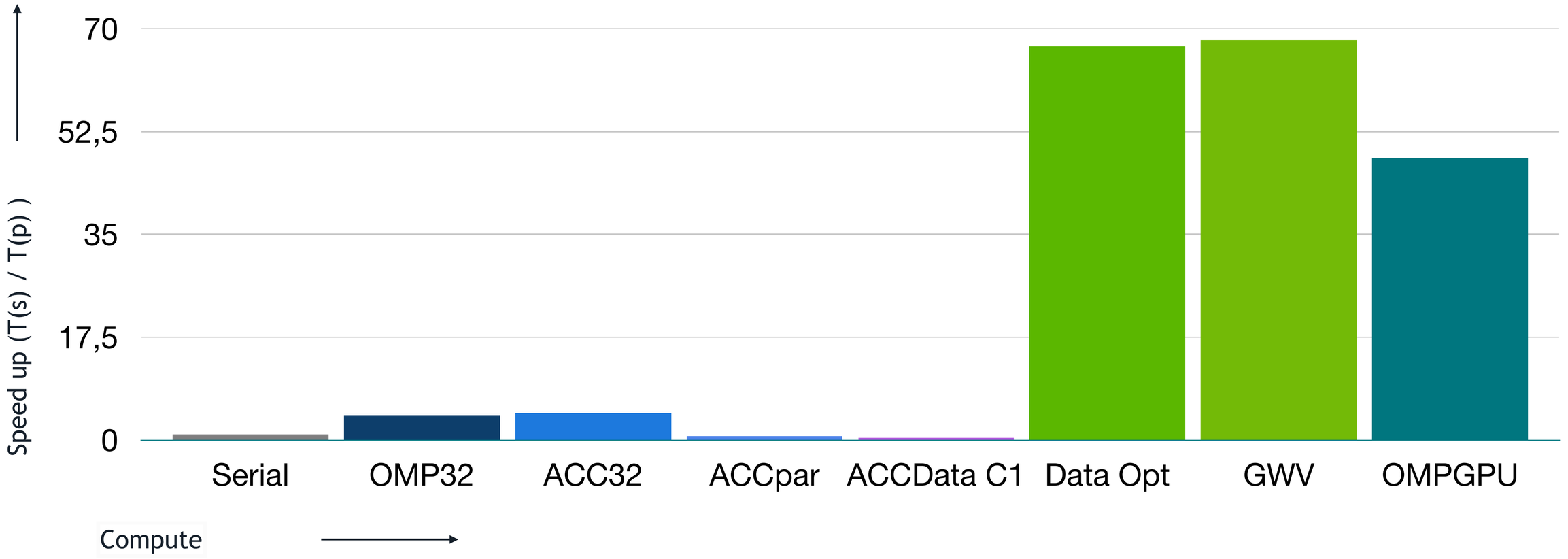
## Simulation was performed 1000 Iterations on Leonardo

```
#pragma omp target data map(alloc:Anew) map(A)
    while (error > tol && niter < niter_max)
    {
    error = 0.0;
     #pragma omp parallel for reduction(max:error)
       for (int j = 1; j < n-1; ++j) {
          for (int i = 1; i < m-1; ++i) {
             Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);
             error = fmax(error, fabs(Anew[idx] - A[idx])); }
             }

      #pragma omp target teams distributed parallel for collapse(2) schedule(static,1)
       for (int j = 1; j < n-1; ++j)
          for (int i = 1; i < m-1; ++i)
             A[j][i] = Anew[j][i]
    }
```

# Offload with OpenMP directives

# Performance Speed Up (Higher Is Better)

Simulation was performed 1000 Iterations on Leonardo



# Closing thoughts

## Accelerating C/C++/Fortran code with OpenX

Simple, Powerful, Portable

## Profile driven approach

Data dependencies and Data movement must be understood

## Optimization

Memory movement: Data regions can be used to make data movement coarse grain

Loop optimisation: use collapse, seq, independent etc