JÜLICH
FORSCHUNGSZENTRUM

What is OpenACC?

16.10.2017 | A. Kreuzer

Outline

- Basic information
- Accelerator Model
- Execution Model
- Memory Model
- Programming Model

Agenda

Basic information

- Accelerator Model
- Execution Model
- Memory Model
- Programming Model

Balance between Productivity & Flexibility

- Accelerated **libraries**:
  - Small code changes
  - Limited by what libraries are available
  - High performance

- Compiler **directives**
  - Simple and familiar
  - Directives not always expose low level details

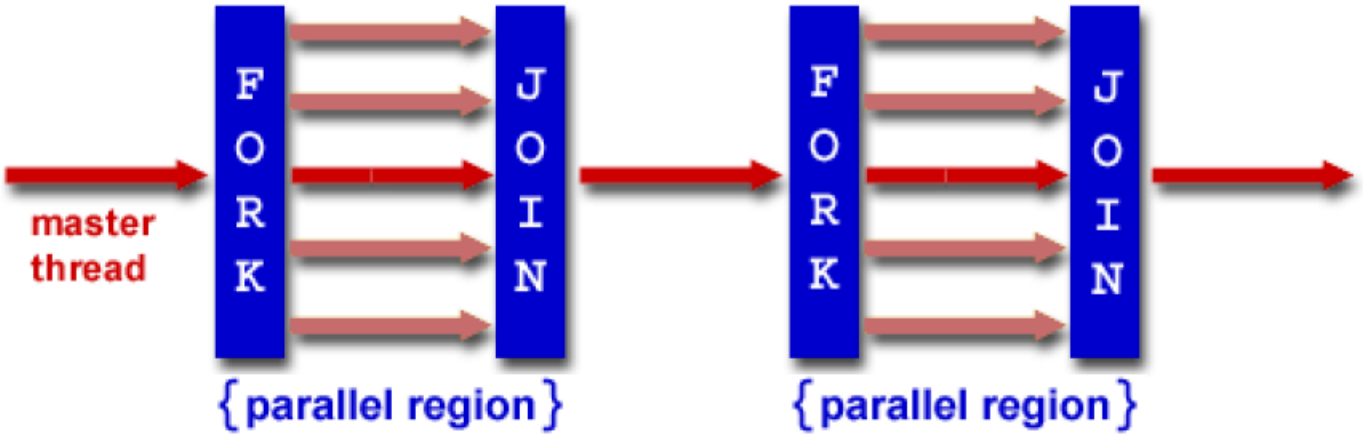
- Parallel programming **languages**
  - Expose low level details for maximum performance
  - Often difficult to learn

What is OpenMP?

- Based on a fork/join modell
- The program starts with one (master) thread. A team of threads for parallel execution is created when reaching a pragma and set free at the end of the pragma.
- A team consists of a fixed number of threads.
- At the end of the pragma region there is a synchronisation of all threads.
- After the pragma region the master thread continues (alone).

What is OpenMP?

- Within one program there can be several pragma regions.



What is OpenMP?

- What is a pragma?
  - Compiler directives
  - In case of OpenMP to specify parallel regions
  - `C: #pragma omp parallel`
  - `Fortran: !$OMP PARALLEL`

OpenACC is an accelerator programming model

- **Pragma based** accelerator programming model
  - **Compiler directives** to specify parallel regions in C, C++, Fortran
  - Programming model allows programmers to **start simple**
  - Similar to the use of **OpenMP**
  - **High level programming** model for accelerator based architectures
  - Create **heterogeneous programs** without explicit accelerator initialization

OpenACC is interoperable and portable

- **Interoperable** with Accelerator Programming Languages and Libraries
- **Portable** across OSes, host CPUs, accelerators and compilers
- **Developed** by

The Portland Group

CAPS

CRAY
THE SUPERCOMPUTER COMPANY

NVIDIA

Where to get information about OpenACC?

- **OpenACC Website**:
  - Specification, quick reference card, getting started videos, Hands-on Labs
    - http://www.openacc.org
  - Best practice guide:
    - http://www.openacc.org/content/openacc-programming-best-practices-guide
- **Website from NVIDIA**:
  - OpenACC Toolkit
    - https://developer.nvidia.com/openacc
  - Getting started information
    - https://developer.nvidia.com/how-to-openacc
  - Online Course
    - https://developer.nvidia.com/openacc-courses

OpenACC has its benefits and limitations

Open	**Open standard** for GPU programming
Productive	Using only **compiler directives** to accelerate your application
Portable	Same code for a **wide range of architectures**
Powerful	Complete **access to massive parallel power** of accelerators

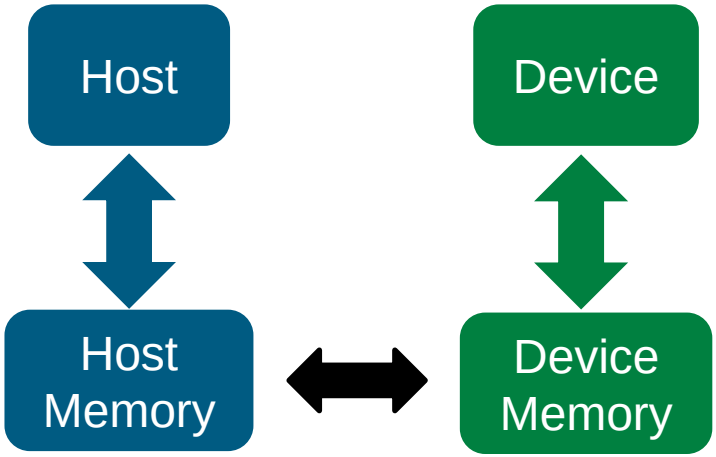
Performance	Simplicity and portability **sometimes** may limit the performance

Agenda

- Basic information
- **Accelerator Model**
- Execution Model
- Memory Model
- Programming Model

OpenACC has an abstract model

- To ensure portability OpenACC has an **abstract model** for accelerated computing
- **Offloading** from host to device
- Host and device could be the same
- Single or separate **memory spaces**

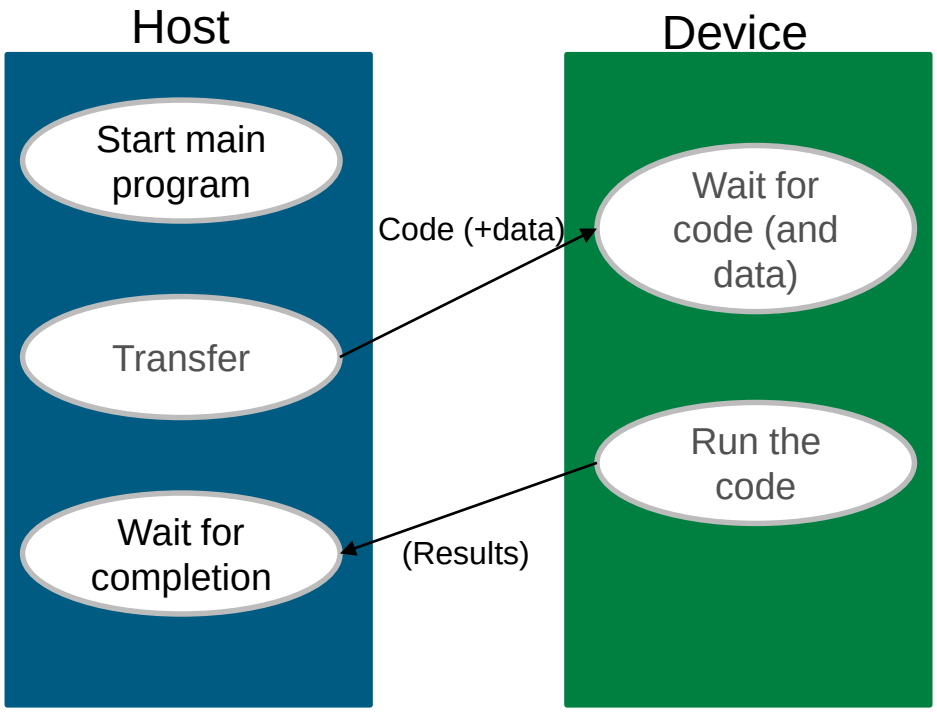


Agenda

- Basic information
- Accelerator Model
- **Execution Model**
- Memory Model
- Programming Model

OpenACC executes host-directed

- **Main** program runs on **host**
- Code is **transferred** to the accelerator
- **Execution** on accelerator is **started**
- **Wait** for completion

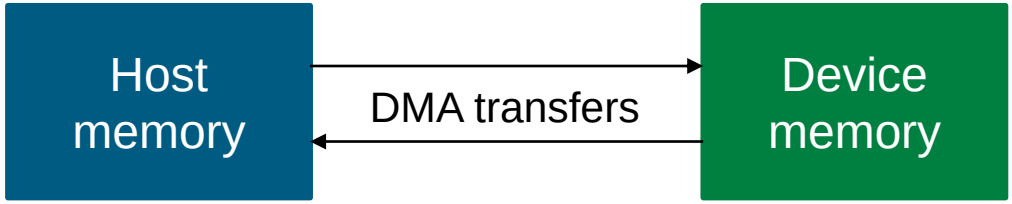


Agenda

- Basic information
- Accelerator Model
- Execution Model
- **Memory Model**
- Programming Model

Memory needs to be transferred

- There may be **two separate memory spaces**



- The **data** needed for the calculations has to be **transferred to the device**
- The **host** has to **read** the results back

Memory transfer is hidden, thus be cautious

- **Transfer hidden** from the programmer, so beware:
  - Latency
  - Bandwidth
  - Limited device memory size

- On the accelerator:
  - Device Memory and Host memory are **not coherent**
  - **Memory management** can be done by compiler

Agenda

- Basic information
- Accelerator Model
- Execution Model
- Memory Model
- **Programming Model**

Programming by directives

- **Directive** based
  - Activated by **compiler options**
    - For example: $ pgcc -acc
  - **Ignored** if not supported by compiler
  - Allows productive, incremental and single-source **porting**
- **Features** through advanced directives or library functions
  - Explicit **data and device management**
  - **Asynchronous** kernel execution

Directives are simple and portable

- **Simple** compiler hints specified by using the **#pragma** mechanism (for C/C++) or **!$** (for Fortran)
- Should be familiar to **OpenMP** users
- Work on **GPUs**, **multicore** CPUs or other **accelerators** like Intel Xeon Phi
- Syntax (C/C++):

  #pragma acc *directive [clause [, clause] …] new-line*
- Syntax (Fortran):

  !$acc *directive [clause [, clause] …]*

  !$acc end *directive*

Example

C code

```
#pragma acc data
        copy(x[0:N],y[0:N])
#pragma acc parallel loop
{
  for (int i=0; i<N; i++){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  for (int i=0; i<N; i++){
    y[i] = i*x[i]+y[i];
  }
}
```

Fortran code

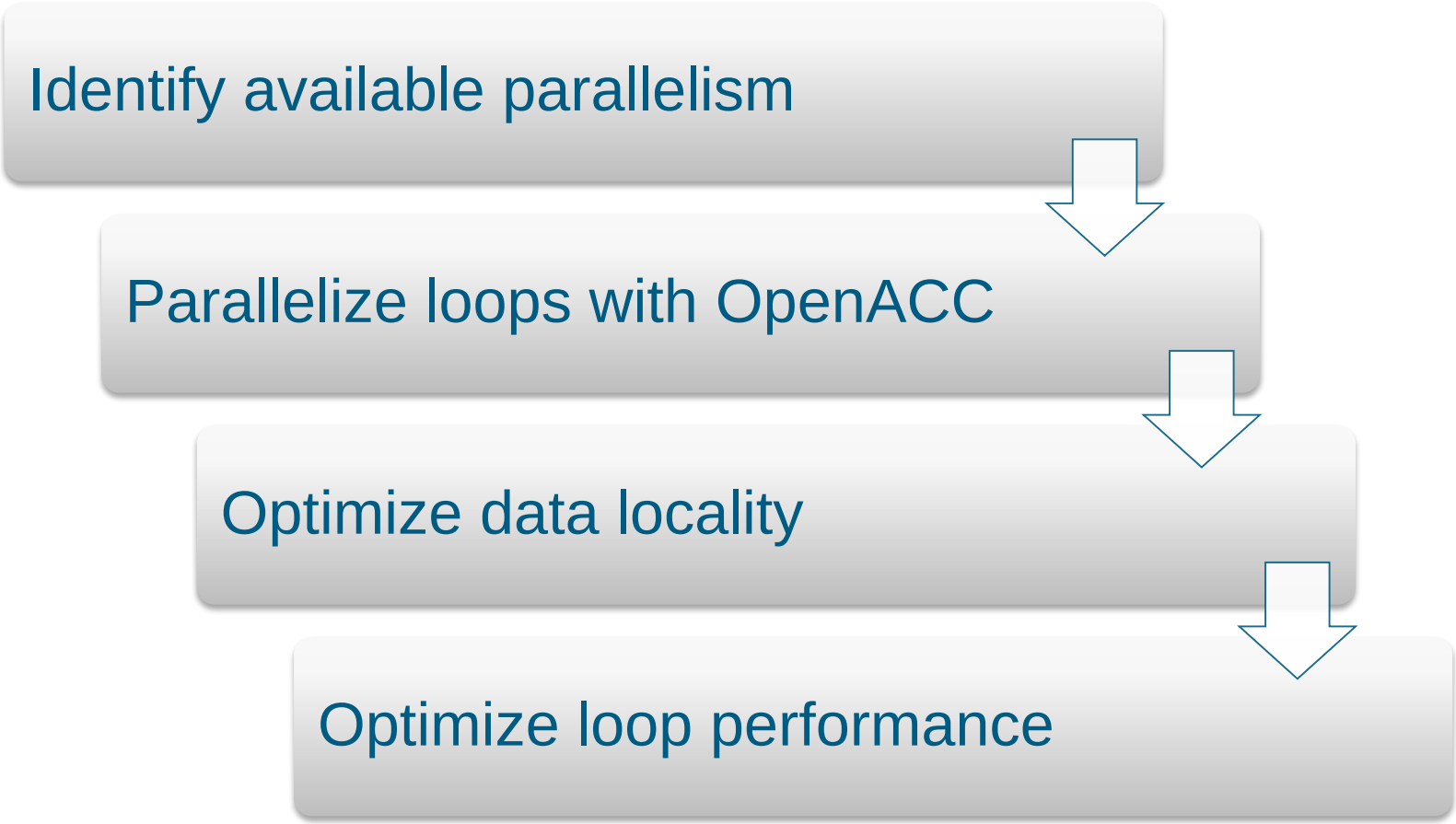
```
!$acc data copy(x(1:N),y(1:N))
!$acc parallel loop

  do i=1,N
    x(i) = 1.0
    y(i) = 2.0
  end do
  do i=1,N
    y(i) = i*x(i)+y(i)
  end do
!$acc end parallel loop
!$acc end data
```

JÜLICH
FORSCHUNGSZENTRUM

How to use OpenACC 2.5?

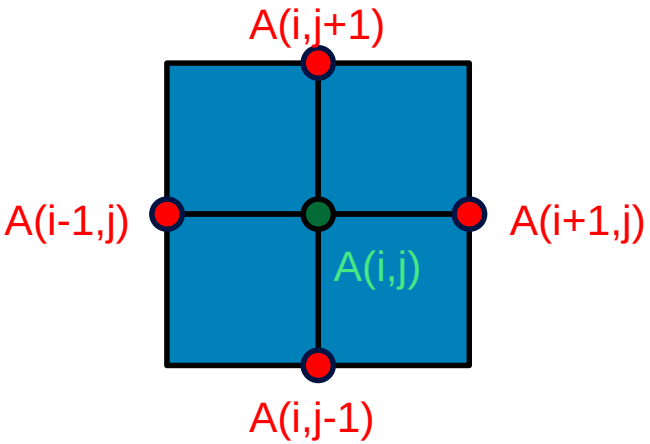
16.10.2017 | A. Kreuzer

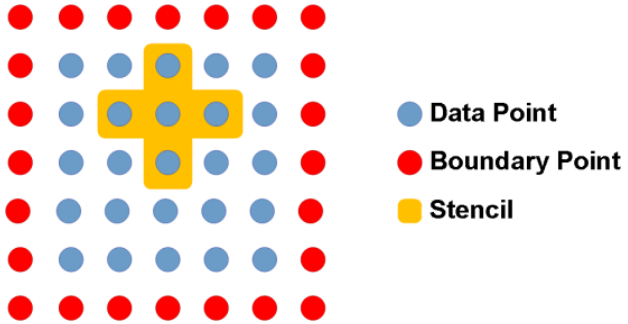
Work flow when using OpenACC



Pursue workflow with the Jacobi example

- **Jacobi** iteration:
  - Iteratively **converges** to correct value
  - Computing new values at each point from the **average** of neighboring points
  - **Example**: Solve Laplace equation in 2D: $\nabla^2 A(x,y) = 0$





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

Jacobi iteration: C 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 = 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];
    }
  }
  for( int i = 1; i < M-1; i++){
    A[0][i] = A[N-2][i];
    A[N-1][i] = A[1][i];
  }
  iter++;
}
```

Iterate until converged

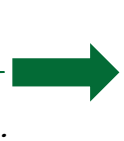
Jacobi iteration: C code

```
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 = 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];
    }
  }
  for( int i = 1; i < M-1; i++){
    A[0][i] = A[N-2][i];
    A[N-1][i] = A[1][i];
  }
  iter++;
}
```

Iterate across matrix elements

Jacobi iteration: C 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 = 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];
    }
  }
  for( int i = 1; i < M-1; i++){
    A[0][i] = A[N-2][i];
    A[N-1][i] = A[1][i];
  }
  iter++;
}
```



Calculate new value from neighbours

Jacobi iteration: C 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 = 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];
    }
  }
  for( int i = 1; i < M-1; i++){
    A[0][i] = A[N-2][i];
    A[N-1][i] = A[1][i];
  }
  iter++;
}
```

Swap input/output arrays

Jacobi iteration: C 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 = 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];
    }
  }
  for( int i = 1; i < M-1; i++){
    A[0][i] = A[N-2][i];
    A[N-1][i] = A[1][i];
  }
  iter++;
}
```

Set periodic boundary conditions

Jacobi iteration: Fortran code

```
do while ((error > tol) .and. (iter < iter_max))
  error = 0.0
  do j = 2, M-1
    do i = 2, N-1
      Anew(i,j) = 0.25 *(A(i+1,j) + A(i-1,j)
                         + A(i,j-1) + A(i,j+1))
      error = max(error, abs(Anew(i,j) - A(i,j)))
    end do
  end do
  do j = 2, M-1
    do i = 2, N-1
      A(i,j) = Anew(i,j)
    end do
  end do
  do i = 2, N-1
    A(i,1) = A(i,N-1)
    A(i,N) = A(I,2)
  end do
  iter = iter +1
end do
```

Iterate until converged

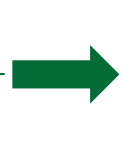
Jacobi iteration: Fortran code

```
do while ((error > tol) .and. (iter < iter_max))
  error = 0.0
  do j = 2, M-1
    do i = 2, N-1
      Anew(i,j) = 0.25 *(A(i+1,j) + A(i-1,j)
                         + A(i,j-1) + A(i,j+1))
      error = max(error, abs(Anew(i,j) - A(i,j)))
    end do
  end do
  do j = 2, M-1
    do i = 2, N-1
      A(i,j) = Anew(i,j)
    end do
  end do
  do i = 2, N-1
    A(i,1) = A(i,N-1)
    A(i,N) = A(I,2)
  end do
  iter = iter +1
end do
```

Iterate across matrix elements

Jacobi iteration: Fortran code

```
do while ((error > tol) .and. (iter < iter_max))
  error = 0.0
  do j = 2, M-1
    do i = 2, N-1
      Anew(i,j) = 0.25 *(A(i+1,j) + A(i-1,j)
                         + A(i,j-1) + A(i,j+1))
      error = max(error, abs(Anew(i,j) - A(i,j)))
    end do
  end do
  do j = 2, M-1
    do i = 2, N-1
      A(i,j) = Anew(i,j)
    end do
  end do
  do i = 2, N-1
    A(i,1) = A(i,N-1)
    A(i,N) = A(I,2)
  end do
  iter = iter +1
end do
```



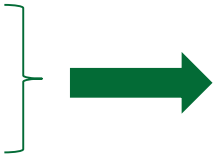
Calculate new value from neighbours

Jacobi iteration: Fortran code

```
do while ((error > tol) .and. (iter < iter_max))
  error = 0.0
  do j = 2, M-1
    do i = 2, N-1
      Anew(i,j) = 0.25 *(A(i+1,j) + A(i-1,j)
                         + A(i,j-1) + A(i,j+1))
      error = max(error, abs(Anew(i,j) - A(i,j)))
    end do
  end do
  do j = 2, M-1
    do i = 2, N-1
      A(i,j) = Anew(i,j)
    end do
  end do
  do i = 2, N-1
    A(i,1) = A(i,N-1)
    A(i,N) = A(I,2)
  end do
  iter = iter +1
end do
```

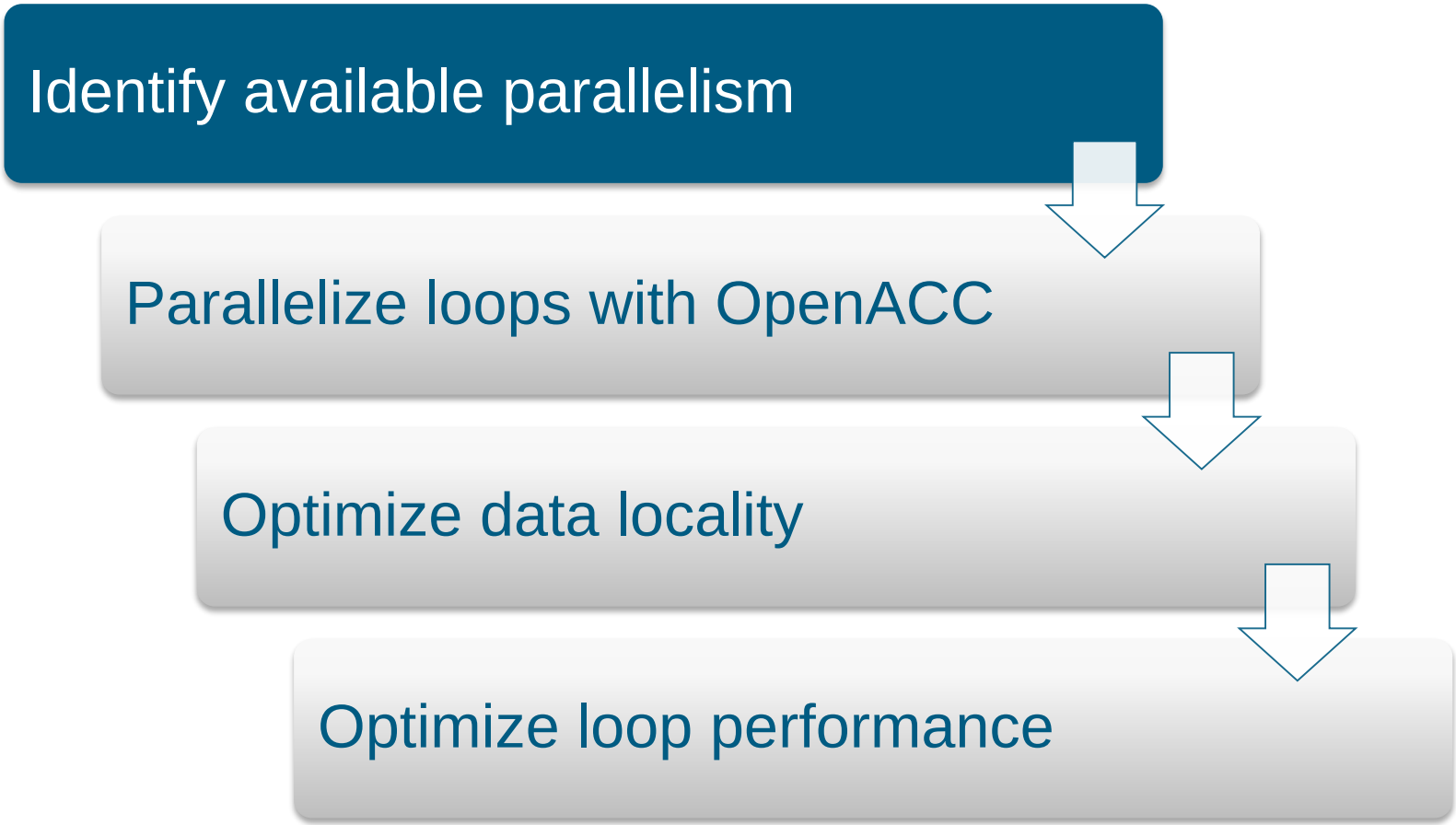
Jacobi iteration: Fortran code

```
do while ((error > tol) .and. (iter < iter_max))
  error = 0.0
  do j = 2, M-1
    do i = 2, N-1
      Anew(i,j) = 0.25 *(A(i+1,j) + A(i-1,j)
                         + A(i,j-1) + A(i,j+1))
      error = max(error, abs(Anew(i,j) - A(i,j)))
    end do
  end do
  do j = 2, M-1
    do i = 2, N-1
      A(i,j) = Anew(i,j)
    end do
  end do
  do i = 2, N-1
    A(i,1) = A(i,N-1)
    A(i,N) = A(I,2)
  end do
  iter = iter +1
end do
```



Set periodic boundary conditions

Work flow when using OpenACC



Identify available parallelism Preparations

- Open 2 terminals (1 for compiling, 1 for executing)

1 Terminal to compile

- $ module load PGI
- for C programmers:
  $ cd OpenACC/Programming-Model-OpenACC/exercises/C or
- for FORTRAN programmers
  $ cd OpenACC/Programming-Model-OpenACC/exercises/FORTRAN

2 Terminal to execute

- $ module load PGI
- $ cd your_directory
- Get a job via slurm
  - $ salloc --reservation=openacc --time=1:30:00 --cpus-per-task=4 --partition=gpus --gres=gpu:1
  - $ srun --pty /bin/bash

Identify available parallelism Using pgprof – Exercise 1

1 Terminal to compile

- $ cd task1/
- Use a profiling tool to obtain an application profile and identify the hotspots of the Jacobi example:
  - $ make profile

2 Terminal to execute

- $ cd task1/
- Use a profiling tool to obtain an application profile of the Jacobi example:
  - $ pgprof --cpu-profiling-scope instruction --cpu-profiling-mode flat ./laplace2d_profile
- Where is the **hotspot**?
- Which **parts** of the code can be parallelized?

Solution: Loop for matrix update is most time consumable

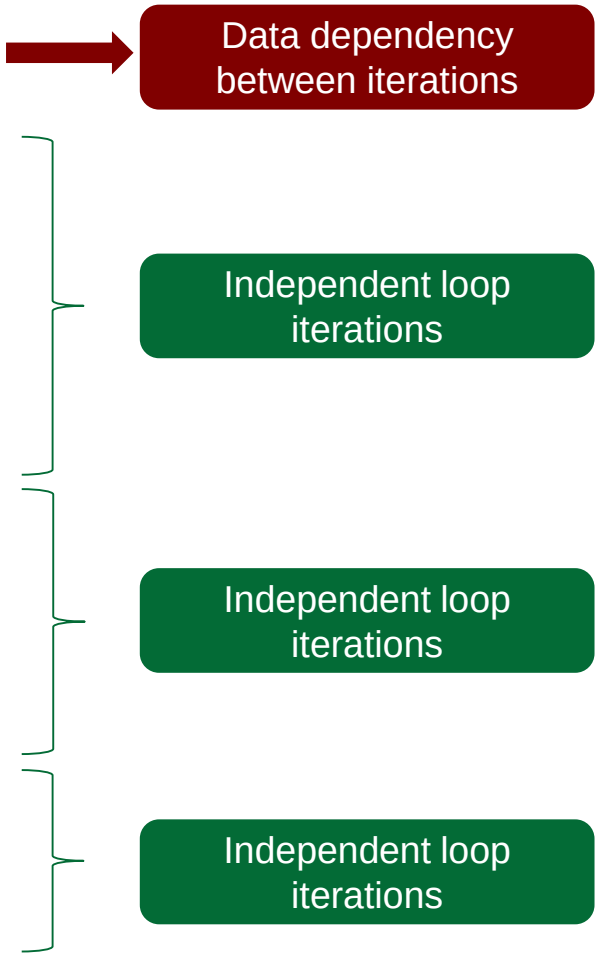
```
pgcc -fast -Minfo=all,intensity -Mprof=ccff laplace2d.c
main:
     45, Intensity = 4.00
         Loop not fused: function call before adjacent
loop
         Generated vector sse code for the loop
     75, Intensity = 0.0
     78, Intensity = 1.00
     80, Intensity = 1.00
         Generated vector sse code for the loop
         Generated 3 prefetch instructions for the loop
    …

======== CPU profiling result (flat):
Time(%)      Time  Name
 22.84%    17.04s    c mcopy8 (0xdae5516b)
 19.54%    14.58s  main (./laplace2d.c:80 0x426)
  9.69%  7.22998s  main (./laplace2d.c:80 0x435)
  9.56%  7.12998s  main (./laplace2d.c:80 0x44b)
  6.00%  4.47999s  main (./laplace2d.c:80 0x447)
  5.82%  4.33999s  main (./laplace2d.c:80 0x456)
  5.75%  4.28999s  main (./laplace2d.c:80 0x43e)
  5.44%  4.05999s  main (./laplace2d.c:80 0x46b)
…
```

- pgprof informs us that the **computational intensity** (calculations/data movement) is high enough to use
- This shows how the code is **currently optimized**
- Most of the **time is spent** in the loop for matrix update

```
======== CPU profiling result (flat):
Time(%)      Time  Name
 53.22%  105.35s  MAIN_ (./laplace2d.f90:81
 10.10%   19.99s  MAIN_ (./laplace2d.f90:90
```

For-loops are independent (C 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 = fmax(error, fabs(Anew[j][i] - A[j][i]));
    }
  }
  for (int j = 1; j < N-1; j++) {
    for (int i = 0; i < M; i++) {
      A[j][i] = Anew[j][i];
    }
  }
  for( int i = 1; i < M-1; i++){
    A[0][i] = A[N-2][i];
    A[N-1][i] = A[1][i];
  }
  iter++;
}
```



For-loops are independent (Fortran code)

```
do while ((error > tol) .and. (iter < iter_max))
  error = 0.0
  do j = 2, M-1
    do i = 2, N-1
      Anew(i,j) = 0.25 *(A(i+1,j) + A(i-1,j)
                         + A(i,j-1) + A(i,j+1))
      error = max(error, abs(Anew(i,j) - A(i,j)))
    end do
  end do
  do j = 2, M-1
    do i = 2, N-1
      A(i,j) = Anew(i,j)
    end do
  end do
  do i = 2, N-1
    A(i,1) = A(i,N-1)
    A(i,N) = A(I,2)
  end do
  iter = iter +1
end do
```

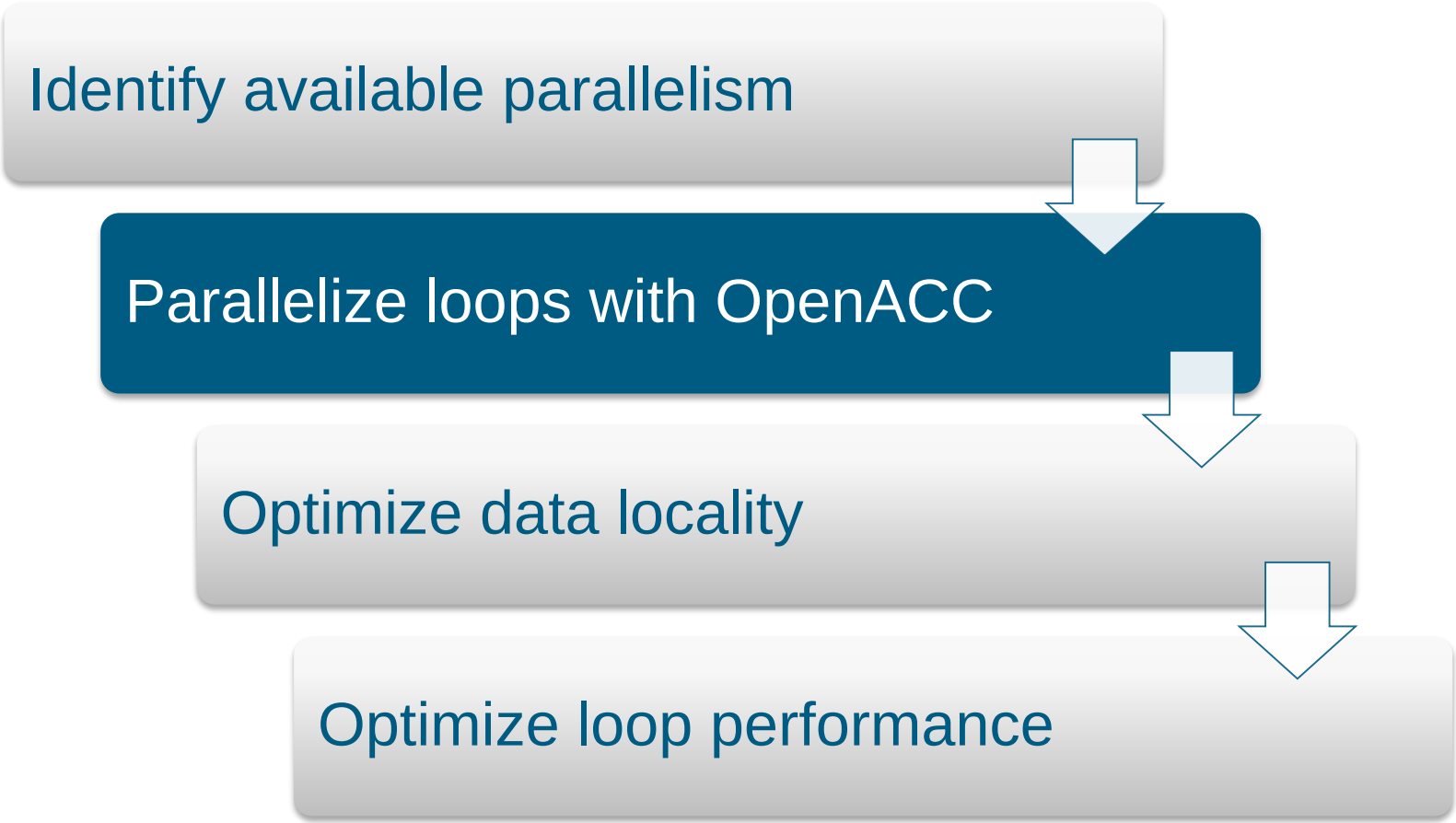
Data dependency between iterations

Independent loop iterations

Independent loop iterations

Independent loop iterations

Work flow when using OpenACC



Parallelize loops with OpenACC
Parallel directive - Usage

- Programmer identifies a block of code containing parallelism, **compiler generates kernel**
- Starts a number of **gangs**
- Implicit **barrier** at the end of a parallel region
- Each gang executes the same code **sequentially**

C/C++:

```
#pragma acc parallel [clause[[,] clause] …] new-line
{structured block}
```

Fortran:

```
!$acc parallel [clause[[,] clause] …]
structured block
!$acc end parallel
```

Parallelize loops with OpenACC
Parallel directive - Clauses

reduction (operator:list)	A **reduction** is performed on the listed variables. Supports +, *, max, min and various logical operations.
private (list)	A **copy** of the listed variables is made for each gang.
firstprivate (list)	Same as private but the **copy** will be **initialized** with the value from the host.
if (condition)	When condition is true, the parallel region will **execute on the accelerator**; otherwise, it will execute on the host.
async [(int)]	There will be **no implicit barrier** at the end of the parallel region.

Parallelize loops with OpenACC
Loop directive - Usage

- Programmer identifies a loop that can be **parallelized**
- Must be **directly before** a loop
- Can describe what **type of parallelism** to use to execute the loop

C/C++:

```
#pragma acc loop[clause[[,] clause] …] new-line
{structured block}
```

Fortran:

```
!$acc loop[clause[[,] clause] …]
structured block
!$acc end loop
```

Parallelize loops with OpenACC
Loop directive - Clauses

independent	Iterations of the loop are **data-independent**. Could only be used within a kernels region.
collapse (int)	Specifies number of **tightly nested loops**.
seq	Specifies that the loop will be executed **sequentially** by the accelerator.

Parallelize loops with OpenACC
Parallel directive - Example

C code

```
double sum = 0.0;
#pragma acc parallel
{
#pragma acc loop
  for (int i=0; i<N; i++){
    x[i] = 1.0;
    y[i] = 2.0;
  }
#pragma acc loop
  for (int i=0; i<N; i++){
    y[i] = i*x[i]+y[i];
    sum+=y[i];
  }
}
```

Kernel 1

Kernel 2

Fortran code

```
sum = 0.0
!$acc parallel
!$acc loop
  do i=1,N
    x(i) = 1.0
    y(i) = 2.0
  end do
!$acc end loop
!$acc loop
  do i=1,N
    y(i) = i*x(i)+y(i)
    sum  = sum + y(i)
  end do
!$acc end loop
!$acc end parallel
```

Kernel 1

Kernel 2

Parallelize loops with OpenACC
Parallel Loop directive - Usage

- **Combined** directive is a shortcut and is used instead of two separated directives (*parallel* and *loop*)
- Any clause that is allowed on a *parallel* or *loop* directive is allowed here
- **Restrictions**:
  - The combined directive may **not appear within the body** of another parallel region
  - The restrictions from the parallel directive apply

C/C++: `#pragma acc parallel loop[clause[[,] clause] …]`

Fortran: `!$acc parallel loop[clause[[,] clause] …]`

Parallelize loops with OpenACC
Parallel Loop directive example

C code

```
double sum = 0.0;
#pragma acc parallel loop
{for (int i=0; i<N; i++){
  x[i] = 1.0;
  y[i] = 2.0;
}}

#pragma acc parallel loop
reduction(+:sum)
{for (int i=0; i<N; i++){
  y[i] = i*x[i]+y[i];
  sum+=y[i];
}}
```

Kernel 1

Kernel 2

Fortran code

```
sum = 0.0
!$acc parallel loop
  do i=1,N
    x(i) = 1.0
    y(i) = 2.0
  end do
!$acc end parallel loop
!$acc parallel loop reduction(+:sum)
  do i=1,N
    y(i) = i*x(i)+y(i)
    sum  = sum + y(i)
  end do
!$acc end parallel loop
```

Kernel 1

Kernel 2

Parallelize loops with OpenACC
Kernels directive – Usage & Clauses

- Express that a region **may contain parallelism**
- Compiler determines what can safely be parallelized → **kernel**
- Kernels are **launched** on accelerator
- Clauses are mainly the same as for the parallel directive (if, async,..)

C/C++:

```
#pragma acc kernels [clause[[,] clause] …] new-line
{structured block}
```

Fortran:

```
!$acc kernels [clause[[,] clause] …]
structured block
!$acc end kernels
```

Parallelize loops with OpenACC Kernels directive - Example

C code

```
double sum = 0.0;
#pragma acc kernels
{
  for (int i=0; i<N; i++){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  for (int i=0; i<N; i++){
    y[i] = i*x[i]+y[i];
    sum+=y[i];
  }
}
```

Kernel 1

Kernel 2

Fortran code

```
sum = 0.0
!$acc kernels
  do i=1,N
    x(i) = 1.0
    y(i) = 2.0
  end do
  do i=1,N
    y(i) = i*x(i)+y(i)
    sum  = sum + y(i)
  end do
!$acc end kernels
```

Kernel 1

Kernel 2

Parallelize loops with OpenACC
Kernels vs. Parallel

- Both approaches are **equally valid** and can **perform equally well**

Kernel	Parallel
▪ **Compiler performs** parallel **analysis** and parallelizes what it believes safe ▪ Can cover **larger area** of code with single directive ▪ Gives compiler additional leeway to **optimize**	▪ Requires **analysis by programmer** to ensure safe parallelism ▪ Will parallelize what a **compiler may miss** ▪ **Straightforward** path from OpenMP

Parallelize loops with OpenACC Restrictions

- Parallel/kernel regions **may not contain other parallel** and/or kernel regions
- **No branching** into or out of an parallel/kernel construct
- Program **must not depend on the order** of evaluation of the clauses
- At most **one if clause** may appear

Parallelize loops with OpenACC
Atomic directive - Usage

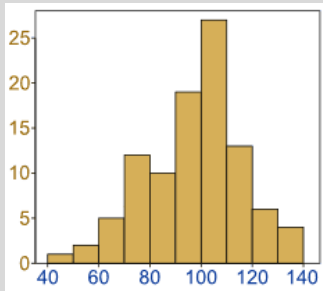
- Subsequent block of code is performed **atomically** with respect to other threads on the accelerator
- **Prevents** simultaneous, **conflicting reading** and **writing** threads, and thus **prevent indeterminate results** and **race conditions**

C code

```
#pragma acc parallel loop
   for(int i=0; i<N; i++){
      #pragma acc atomic
      a[i%100]++;
   }
```

Fortran code

```
!$acc parallel loop
   do i=1, N
      !$acc atomic
      a[i%100]++;
   end do
!$acc end parallel
```



Parallelize loops with OpenACC
Atomic directive - Example

C code

```
for(int it=0;it<ITERS;it++){
    #pragma acc parallel loop
    for(int i=0;i<HN;i++){
        h[i]=0;
    }

    #pragma acc parallel loop
    for(int i=0;i<N;i++) {
        #pragma acc atomic
         h[a[i]]+=1;
    }

}
```

h can only be accessed by 1 thread at a time

Fortran code

```
do it=1, ITERS
    !$acc parallel loop
    do i=1, HN
        h(i)=0
    end do
    !$acc end parallel
    !$acc parallel loop
    do i=1, N
        !$acc atomic
         h(a(i))=h(a(i)) + 1
    end do
    !$acc end parallel
end do
```

h can only be accessed by 1 thread at a time

Parallelize loops with OpenACC
Parallel Loop directive – Exercise 2

- Use the **parallel loop** directive to accelerate the example:

1 Terminal to compile

- $ cd ../task2/
- $ cp laplace2d.c laplace2d_parallelloop.c
- Open laplace2d_parallelloop.c
- Uncomment lines 61-66 and 109-114 to include reference solution and timing
- Add the parallel loop directive(s) and possible options and save
- $ make parallelloop

2 Terminal to execute

- $ cd ../task2/
- Execute:
  $ ./laplace2d_parallelloop
- What is the total **runtime**?

Parallelize loops with OpenACC
Parallel Loop directive - Solution

- Use the **parallel loop** directive to accelerate the example:

```
#pragma acc parallel loop reduction(max:error)
        for (int j = jstart; j < jend; 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]));
            }
        }

#pragma acc parallel loop
        for (int j = jstart; j < jend; j++)
        {
            for( int i = 0; i < M; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        //Periodic boundary conditions
#pragma acc parallel loop
        for( int i = 1; i < M-1; i++ )
        {
                A[0][i]     = A[(N-2)][i];
                A[(N-1)][i] = A[1][i];
        }
```

Parallelize loops with OpenACC Kernels directive – Exercise 3

- Use the **kernels** directive to accelerate the example:

1 Terminal to compile

- $ cd ../task3/
- $ cp laplace2d.c laplace2d_kernels.c
- Open laplace2d_kernels.c
- Uncomment lines 61-66 and 109-114 to include reference solution and timing
- kernels directive(s) and save your changes
- $ make kernels

2 Terminal to execute

- $ cd ../task3/
- Execute:
  $ ./laplace2d_kernels
- What is the total **runtime**?

Parallelize loops with OpenACC Kernels directive - Solution

- Use the **kernels** directive to accelerate the example:

```
#pragma acc kernels
{
        for (int j = jstart; j < jend; 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 = jstart; j < jend; j++)
        {
            for( int i = 0; i < M; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        //Periodic boundary conditions
        for( int i = 1; i < M-1; i++ )
        {
                A[0][i]     = A[(N-2)][i];
                A[(N-1)][i] = A[1][i];
        }
}
```

Parallelize loops with OpenACC
Comparison with OpenMP – Exercise 4

- **Compare the runtime** of your OpenACC with OpenMP apps:

1 Terminal to compile

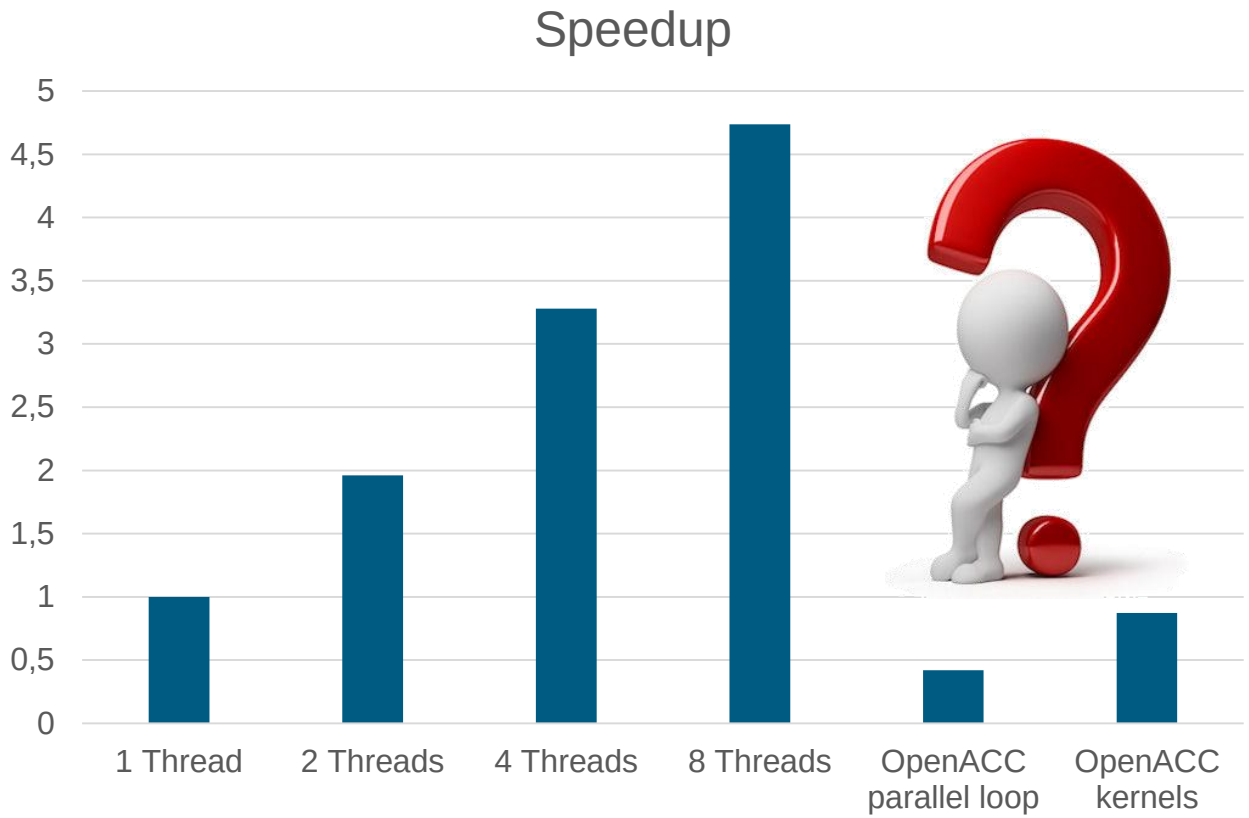
- $ cd ../task4/
- $ make omp
- $ make parallelloop
- $ make kernels

Reference for speedup

2 Terminal to execute

- $ cd ../task4/
- Execute:
  - $OMP_NUM_THREADS=1 ./laplace2d_omp
  - $OMP_NUM_THREADS=2 ./laplace2d_omp
  - $OMP_NUM_THREADS=4 ./laplace2d_omp
  - $ ./laplace2d_parallelloop
  - $./laplace2d_kernels
- What is the respective **speedup** $S = T_s/T_p$?

Parallelize loops with OpenACC
Comparison with OpenMP shows slow-down



Measured on Jureca (Intel Xeon E5-2680 v3 Haswell and NVIDIA K80 GPU)

Parallelize loops with OpenACC
Processing flow discovers data issue

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

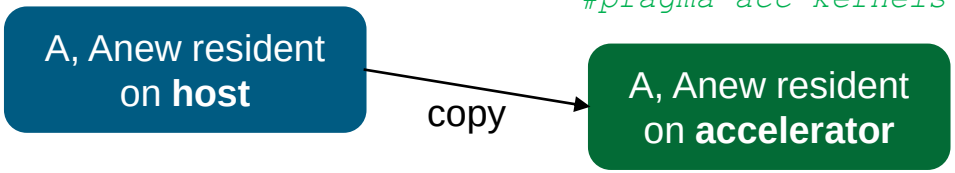
A, Anew resident on **host**

```
  …
  iter++;
}
```

Parallelize loops with OpenACC
Processing flow discovers data issue

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

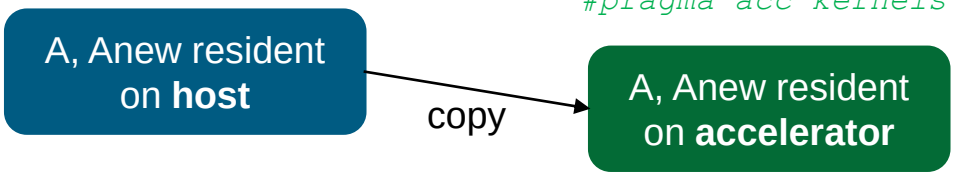
```
#pragma acc kernels
```




```
  …
  iter++;
}
```

Parallelize loops with OpenACC
Processing flow discovers data issue

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

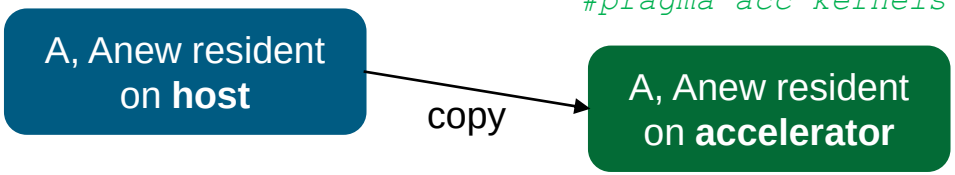



```
#pragma acc kernels
for (int j = jstart; j < jend; j++) {
  for (int i = 1; i < M-1; i++) {
    Anew[j][i] = 0.25 *(A[j][i+1] + A[j][i-1]
                        + A[j-1][i] + A[j+1][i]);
    error = fmax(error, fabs(Anew[j][i] - A[j][i]));
  }
}
```

```
  …
  iter++;
}
```

Parallelize loops with OpenACC
Processing flow discovers data issue

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



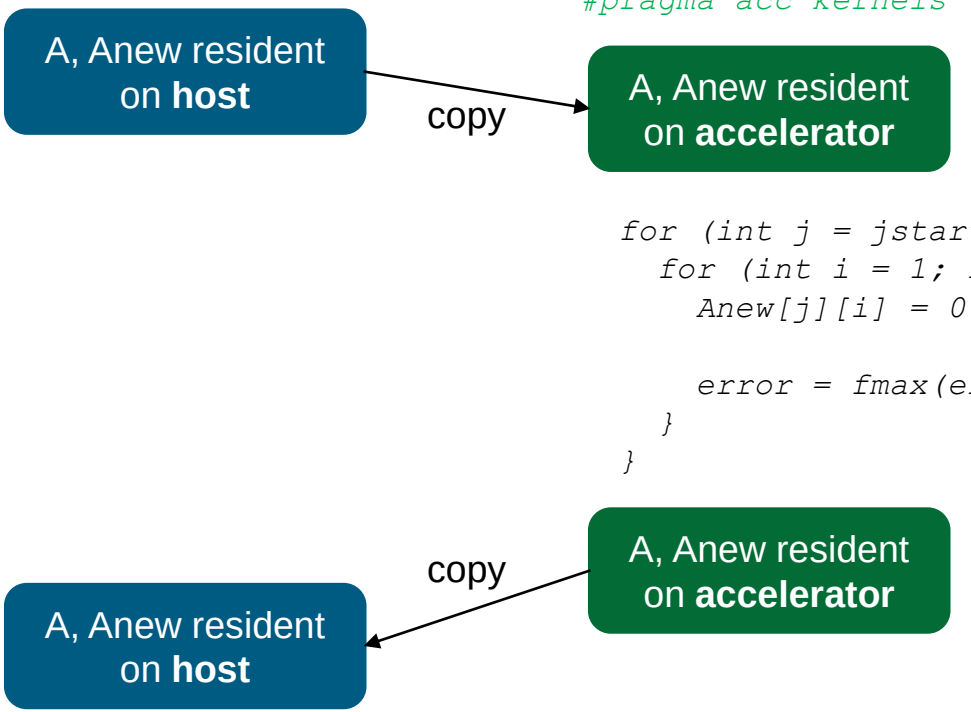

```
#pragma acc kernels
for (int j = jstart; j < jend; j++) {
  for (int i = 1; i < M-1; i++) {
    Anew[j][i] = 0.25 *(A[j][i+1] + A[j][i-1]
                        + A[j-1][i] + A[j+1][i]);
    error = fmax(error, fabs(Anew[j][i] - A[j][i]));
  }
}
```

A, Anew resident on **accelerator**

```
  …
  iter++;
}
```

Parallelize loops with OpenACC
Processing flow discovers data issue

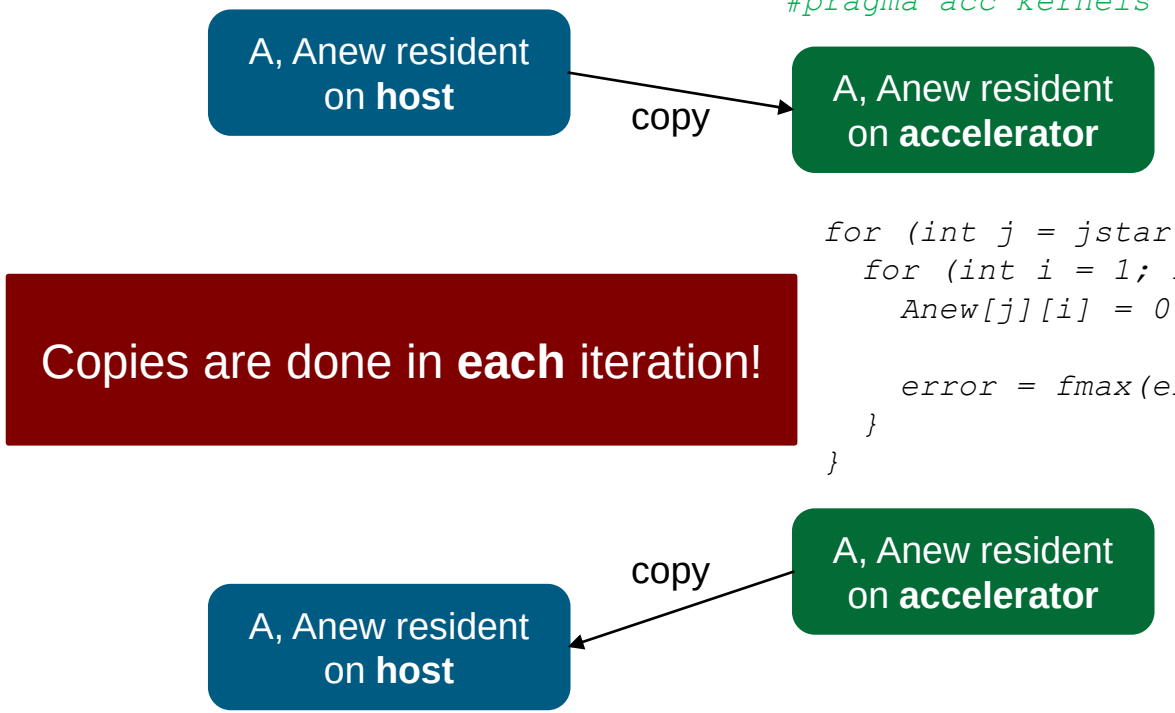
```
while (error > tol && iter < iter_max) {
  error=0.0;
```




```
  …
  iter++;
}
```

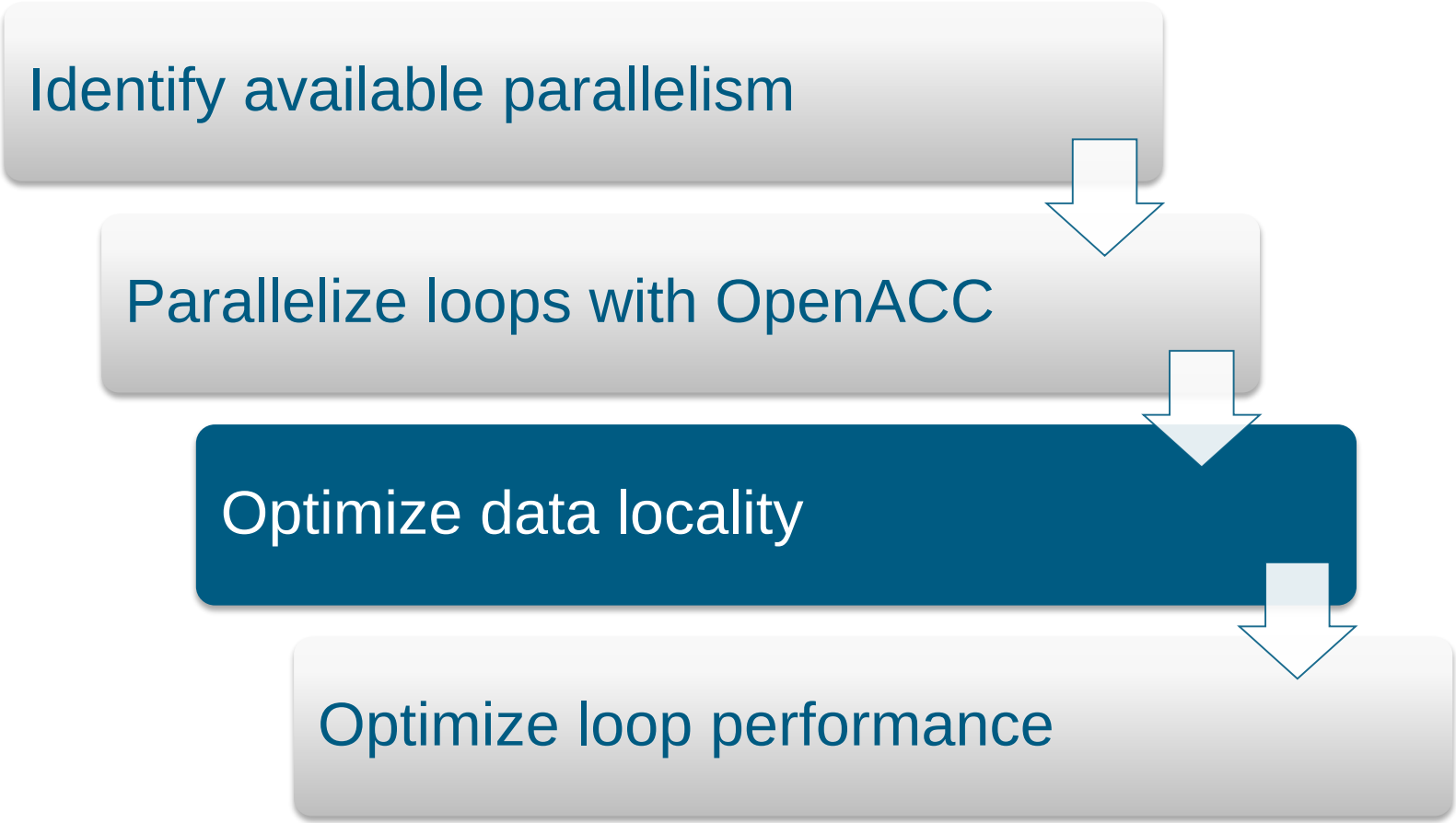
Parallelize loops with OpenACC
Processing flow discovers data issue

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




```
  …
  iter++;
}
```

Work flow when using OpenACC



Optimize data locality
Identify data locality

```
while (error > tol && iter < iter_max) {
  error=0.0;
#pragma acc kernels
  for (int j = jstart; j < jend; 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]));
    }
  }
#pragma acc kernels
  for (int j = jstart; j < jend; j++) {
    for (int i = 0; i < M; i++) {
      A[j][i] = Anew[j][i];
    }
  }
  …
  iter++;
}
```

Does the host **need the data back** between the nested loops or between the iterations of the while loop?

Optimize data locality
The data directive - Usage

- Defines a **region** of code in which accelerator arrays **remain on the device**
- The **arrays are shared** among all kernels in that region
- Data **transfer** is made **explicit** by the **user**

C/C++:

```
#pragma acc data[clause[[,] clause] …] new-line
{structured block}
```

Fortran:

```
!$acc data[clause[[,] clause] …]
structured block
!$acc end data
```

Optimize data locality
The data directive - Clauses

copy (list)	Allocates memory on GPU; copies data to the GPU when **entering** the region; copies data to host when **exiting** the region.
copyin (list)	Allocates memory on GPU; copies data to the **GPU** when **entering** the region.
copyout (list)	Allocates memory on GPU; copies data to **host** when **exiting** the region.
create (list)	Allocates memory on GPU but does **not copy**.
present (list)	Data must be **present on GPU.**

copy, copyin, copyout and create **check if data is already present**, increment the reference count and use that present copy.

Optimize data locality
The data directive - Array shaping

- Needed because:
  - Compiler sometimes **cannot determine size** of arrays
  - The programmer wants to use **subarrays**
- Must be specified explicitly using data clauses and array "shape"

C/C++:

```
#pragma acc data copy(a[lower bound:size]) new-
line
{structured block}
```

Fortran:

```
!$acc data copy(a(lower bound:upper bound))
structured block
!$acc end data
```

Optimize data locality
The data directive - Example

C code

```
#pragma acc data copyout(y[0:N])
                 create(x[0:N])
{
   #pragma acc parallel loop
   for (int i=0; i<N; i++){
     x[i] = 1.0, y[i] = 2.0;
   }

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

}
```

Fortran code

```
!$acc data copyout(y(1:N))
          create(x(1:N))

!$acc parallel loop
   do i=1, N
     x(i) = 1.0, y(i) = 2.0
   end do
   !$acc end parallel
   !$acc parallel loop
   do i=1, N
     y(i) = i*x(i)+y(i)
   end do
   !$acc end parallel
!$acc end data
```

Optimize data locality
The data directive – Exercise 5

- Use the **data** directive to manage data:

1 Terminal to compile

- $ cd ../task5/
- $ cp laplace2d_kernels.c laplace2d_data.c
- Open laplace2d_data.c
- Add the data construct
- Save your changes
- $ make data

2 Terminal to execute

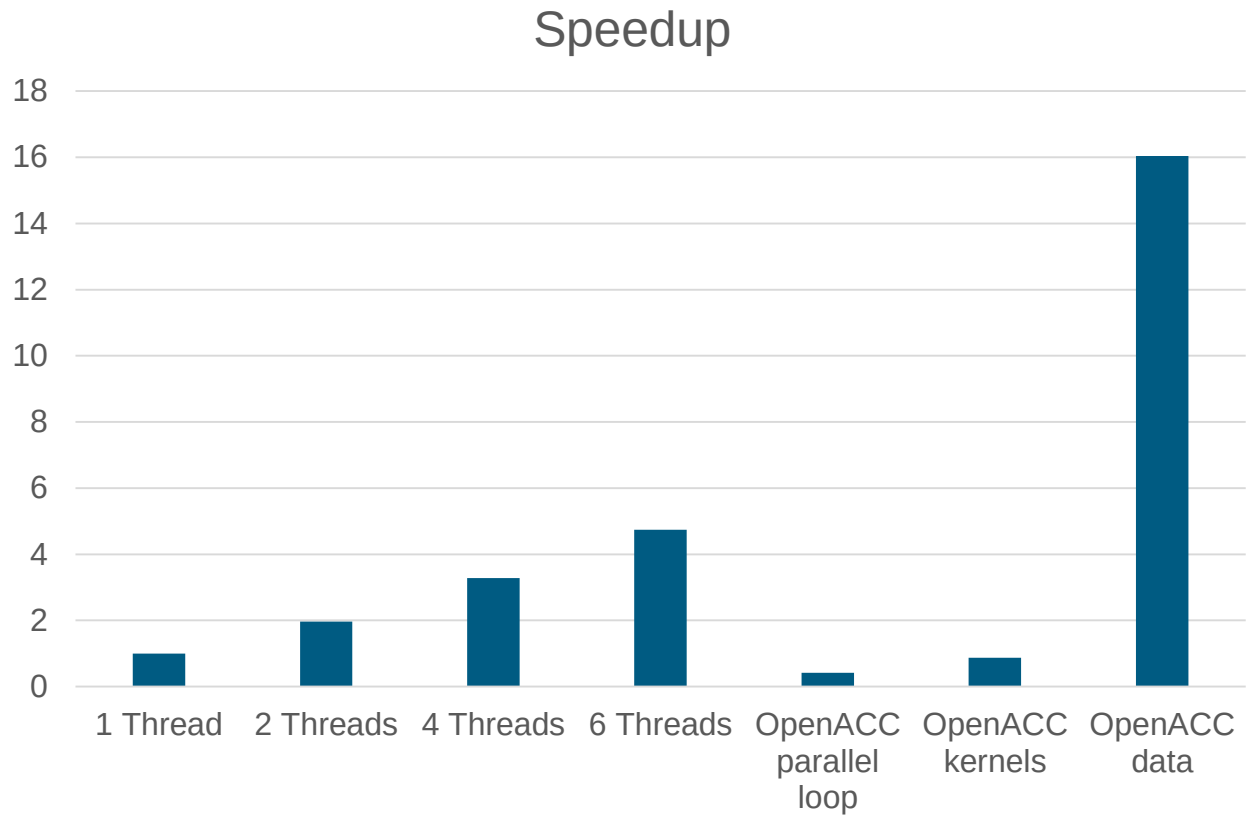
- $ cd ../task5/
- Execute: ./laplace2d_data
- What is the respective **speedup** $S = T_s/T_p$ now?

Optimize data locality
The data directive - Solution

- Use the **data** directive to manage data:

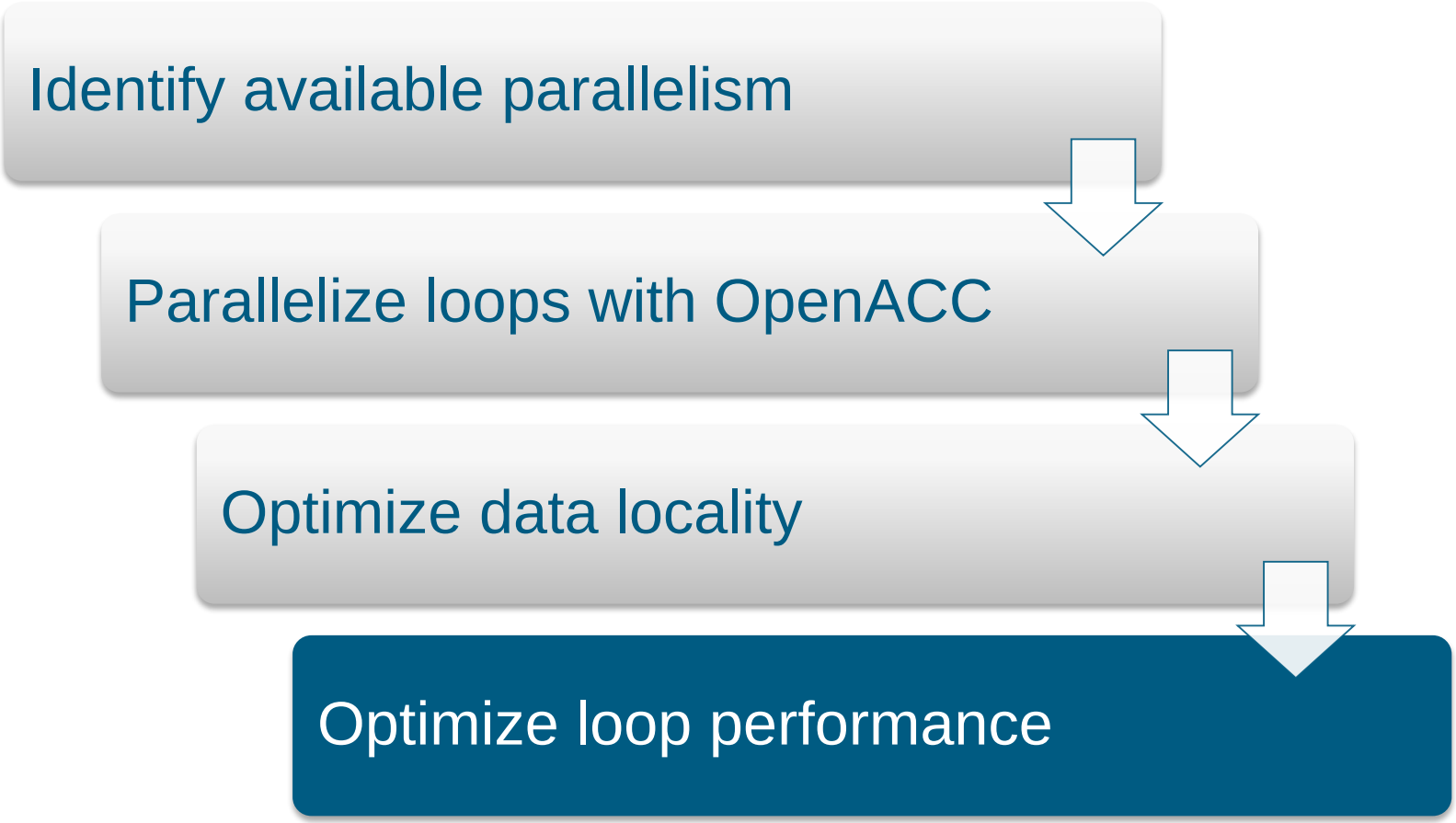
```
#pragma acc data copy(A, Anew)
    while ( error > tol && iter < iter_max )
    {
        error = 0.0;
#pragma acc kernels
{
        for (int j = jstart; j < jend; j++)
        {
            for( int i = 1; i < M-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i]-A[j][i]));
            }
        }

        for (int j = jstart; j < jend; j++)
        {
            for( int i = 0; i < M; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        //Periodic boundary conditions
        for( int i = 1; i < M-1; i++ )
        {
                A[0][i]     = A[(N-2)][i];
                A[(N-1)][i] = A[1][i];
        }
}
```

Optimize data locality
Comparison with OpenMP shows speedup



Measured on Jureca (Intel Xeon E5-2680 v3 Haswell and NVIDIA K80 GPU)

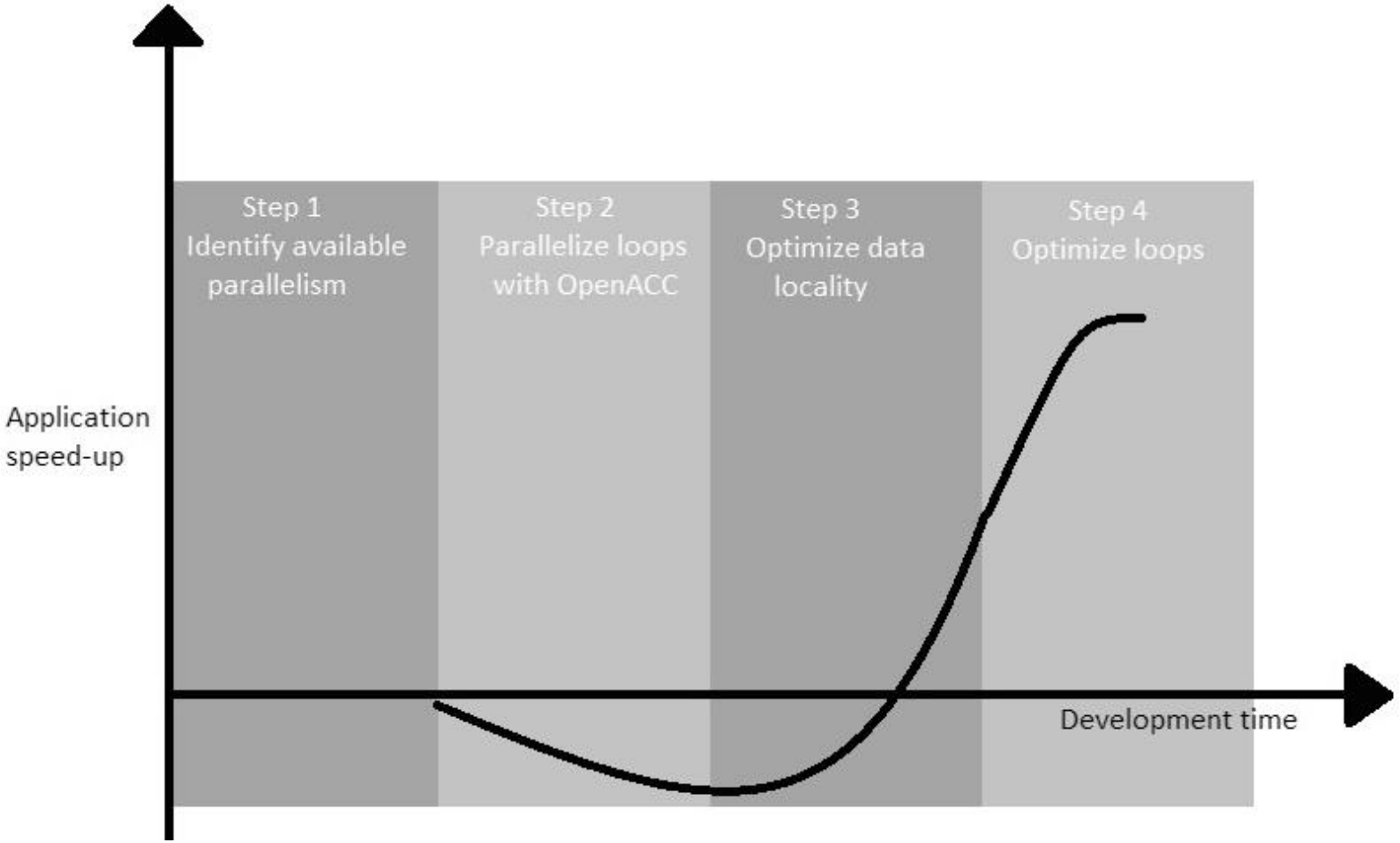
Work flow when using OpenACC



Summary

1. **Identify Available Parallelism**:
   - What important parts of the code have **available parallelism**?
2. **Parallelize Loops**
   - Express **as much parallelism** as possible and ensure you still get correct results.
   - Because the compiler must be **cautious about data movement**, the code might not perform optimally or even slow down.
3. **Optimize Data Locality**
   - The programmer will always know better than the compiler what **data movement is unnecessary**.
4. **Optimize Loop Performance**

Typical porting experience with OpenACC



JÜLICH
FORSCHUNGSZENTRUM

Differentiation from OpenMP 4.5

10/14/2017 | A. Severt

Philosophical Differences

OpenMP	OpenACC
More **explicit**	More **implicit**
User-**directed** parallelism	User-**guided** parallelism
Compiler has **less** performance responsibility	Compiler has **more** performance responsibility
User adds additional directives to **exclude parallelization issues**	Users write code actually **free of parallelization issues**
Different architectures require **different** directives	Higher-level directives allow **same-code** targeting of different architectures

Some OpenACC Directives/Clauses translate 1:1…

OpenACC	OpenMP
acc parallel	omp target teams, parallel
acc loop vector	omp simd
acc data	omp target data
acc update	omp target update
acc copy/copyin/copyout	map(tofrom/to/from:...)
...	...

…. some not!

- acc kernels
- acc loop
- omp parallel workshare

Main difference of OpenACC and OpenMP

- OpenACC is **descriptive**
  - Parallelism and data locality without specified mapping to the hardware
- OpenMP is **prescriptive**
  - The mapping has to be specified by the directive
- The descriptive approach is more **performance portable**

Conclusions

- OpenACC and OpenMP both provide **features aimed at accelerators**
- Both execute **host-directed**
- The two are **not equivalent** and have their own strengths and weaknesses
- Amount of work parallelizing the code is comparable