

# How to use OpenACC 2.5?

10/21/16 | A. Severt

# Work flow when using OpenACC

Identify available parallelism



Parallelize loops with OpenACC



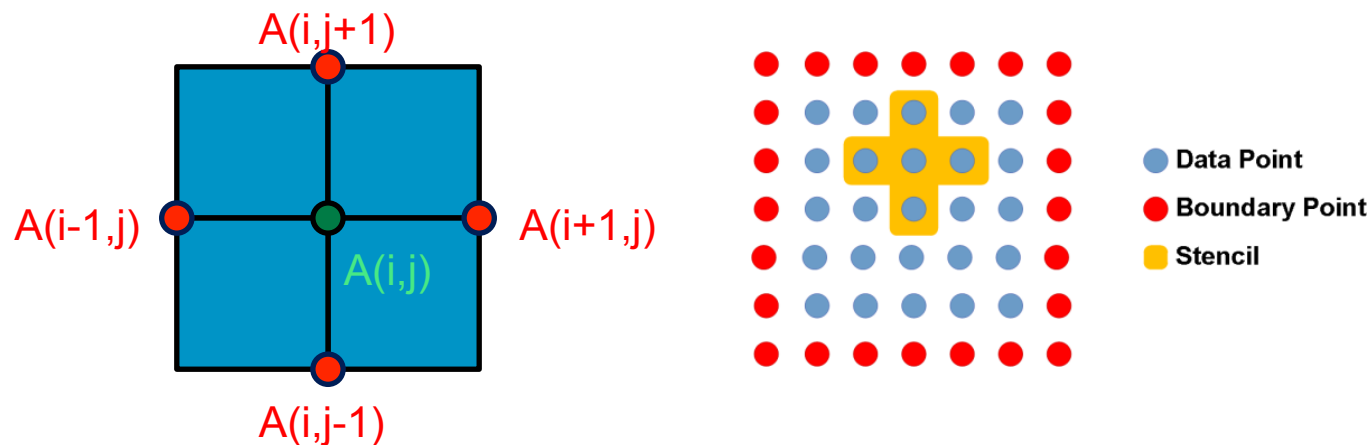
Optimize data locality



Optimize loop performance

# 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++;
}
```



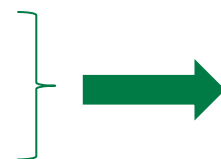
Swap input/output arrays

# 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

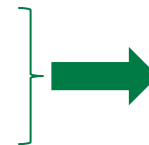
```
do while ((error > tol) .and. (iter < iter_max))
  error = 0.0
  do j = 2, M-1
    do i = 2, N-1
      Anew(i,j) = 0.25 * (A(i+1,j) + A(i-1,j)
                        + A(i,j-1) + A(i,j+1))
      error = max(error, abs(Anew(i,j) - A(i,j)))
    end do
  end do
  do j = 2, M-1
    do i = 2, N-1
  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



Swap input/output arrays

# 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



Parallelize loops with OpenACC



Optimize data locality



Optimize loop performance

# 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`

# 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**

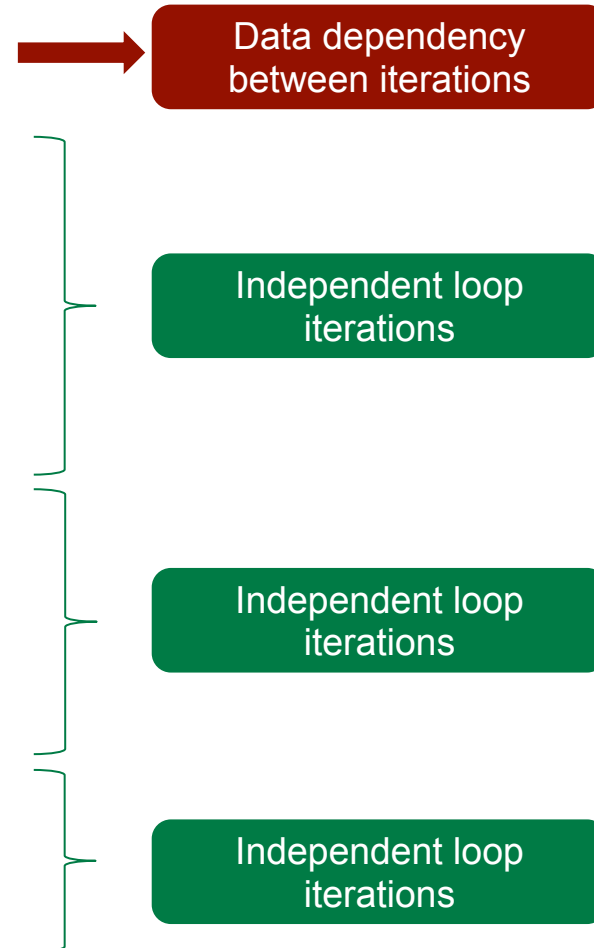
Most of the **time is spent in the loop for matrix update**

```
==== CPU profiling result (flat):
```

Time(%)	Time	Name
53.22%	105.35s	MAIN_ (./laplace2d.f90:81)
10.10%	19.99s	MAIN_ (./laplace2d.f90:90)

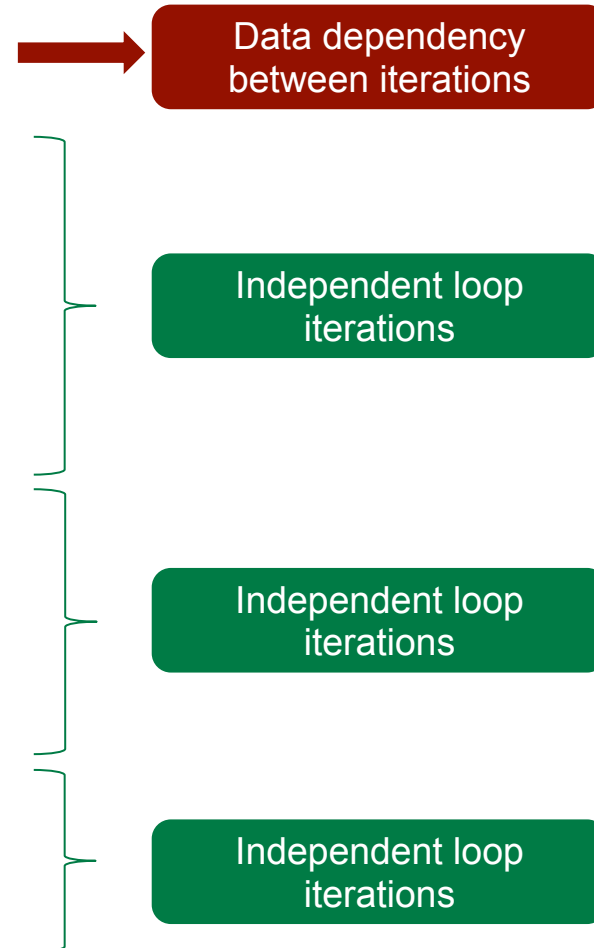
# For-loops are independent (C Code)

```
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
```



# Work flow when using OpenACC

Identify available parallelism



Parallelize loops with OpenACC



Optimize data locality



Optimize loop performance

# 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 <b>reduction</b> is performed on the listed variables. Supports +, *, max, min and various logical operations.
private (list)	A <b>copy</b> of the listed variables is made for each gang.
firstprivate (list)	Same as private but the <b>copy</b> will be <b>initialized</b> with the value from the host.
if (condition)	When condition is true, the parallel region will <b>execute on the accelerator</b> ; otherwise, it will execute on the host.
async [(int)]	There will be <b>no implicit barrier</b> 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 <b>data-independent</b> . Could only be used within a kernels region.
collapse (int)	Specifies number of <b>tightly nested loops</b> .
seq	Specifies that the loop will be executed <b>sequentially</b> 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;
}}
```

Kernel 1

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

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 kernels
```

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

- 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**

### Parallel

- 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

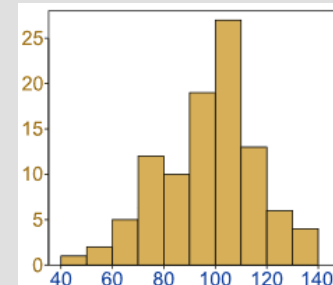
- Subsequent block of code is performed **atomically** with respect to other threads on the accelerator
- Prevents** simultaneous, **conflicting reading** and **writing** threads, and thus **prevent indeterminate results** and **race conditions**

### 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`

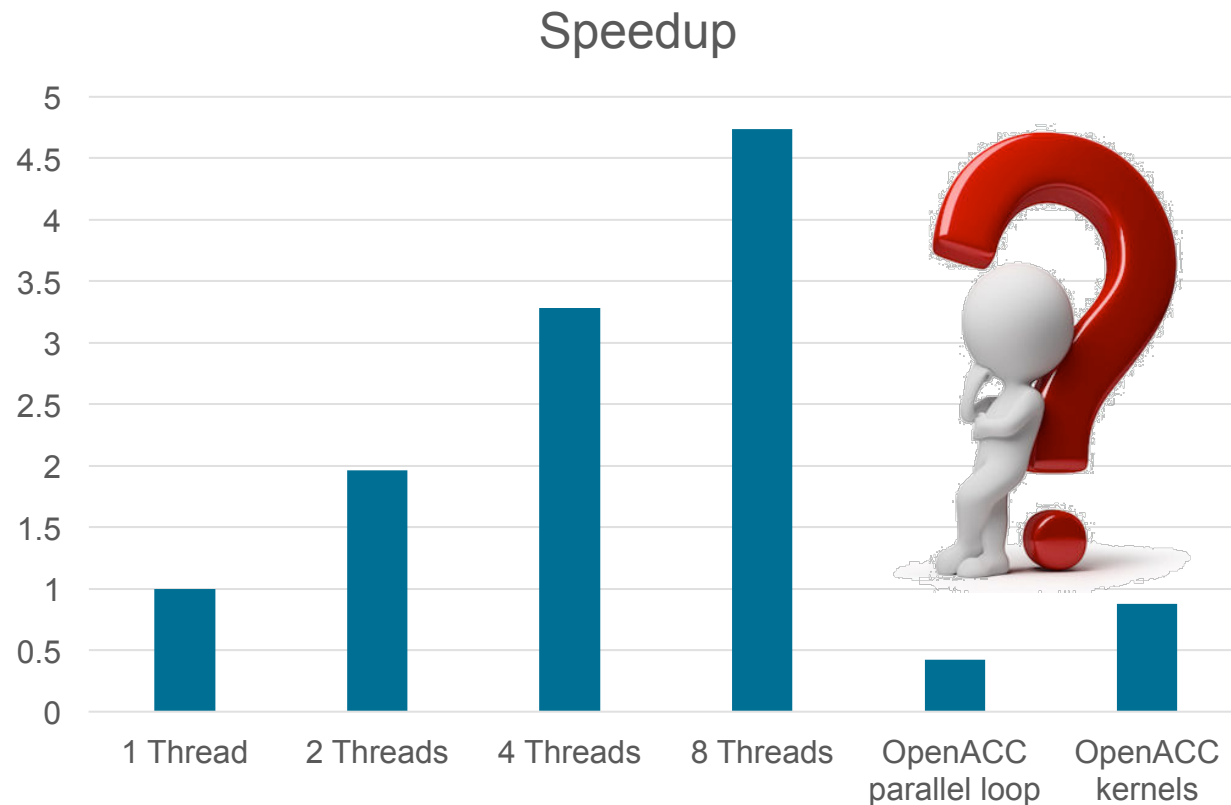
Reference for  
speedup

### 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

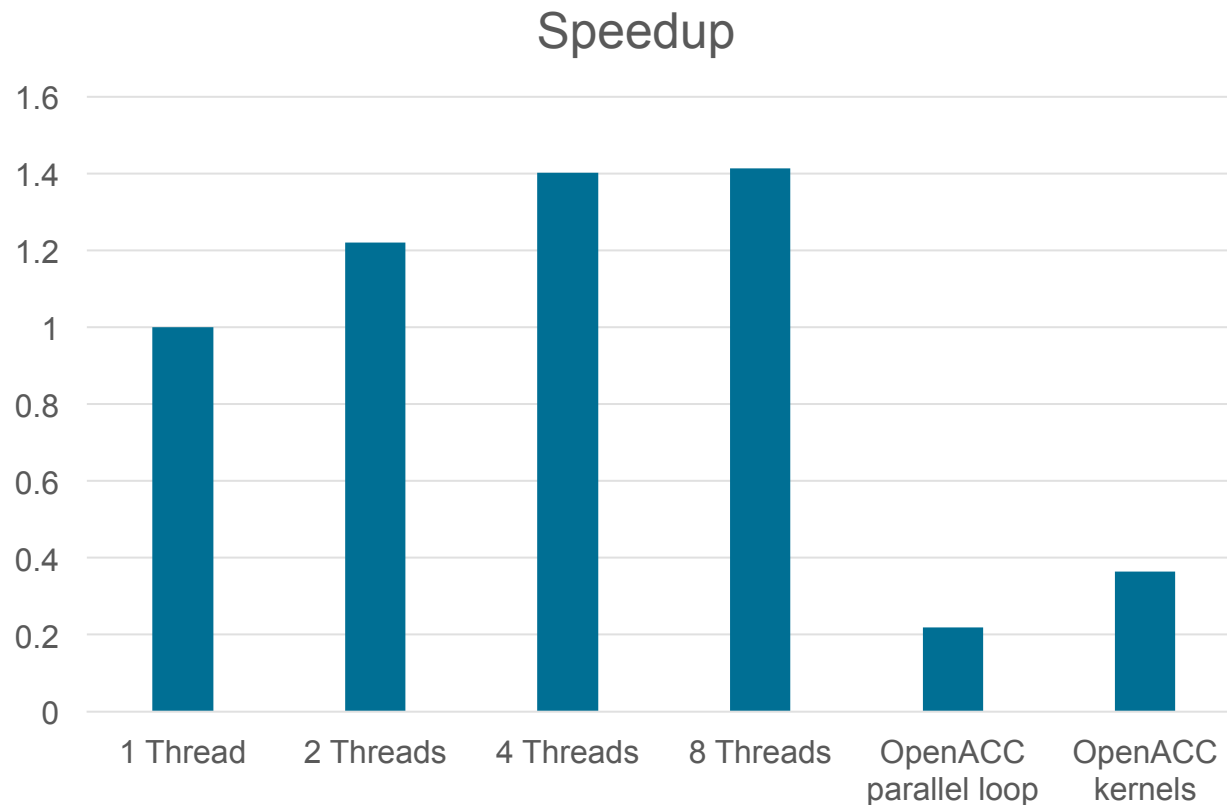
## Comparison with OpenMP shows slow-down



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

# Parallelize loops with OpenACC

## Comparison with OpenMP shows slow-down

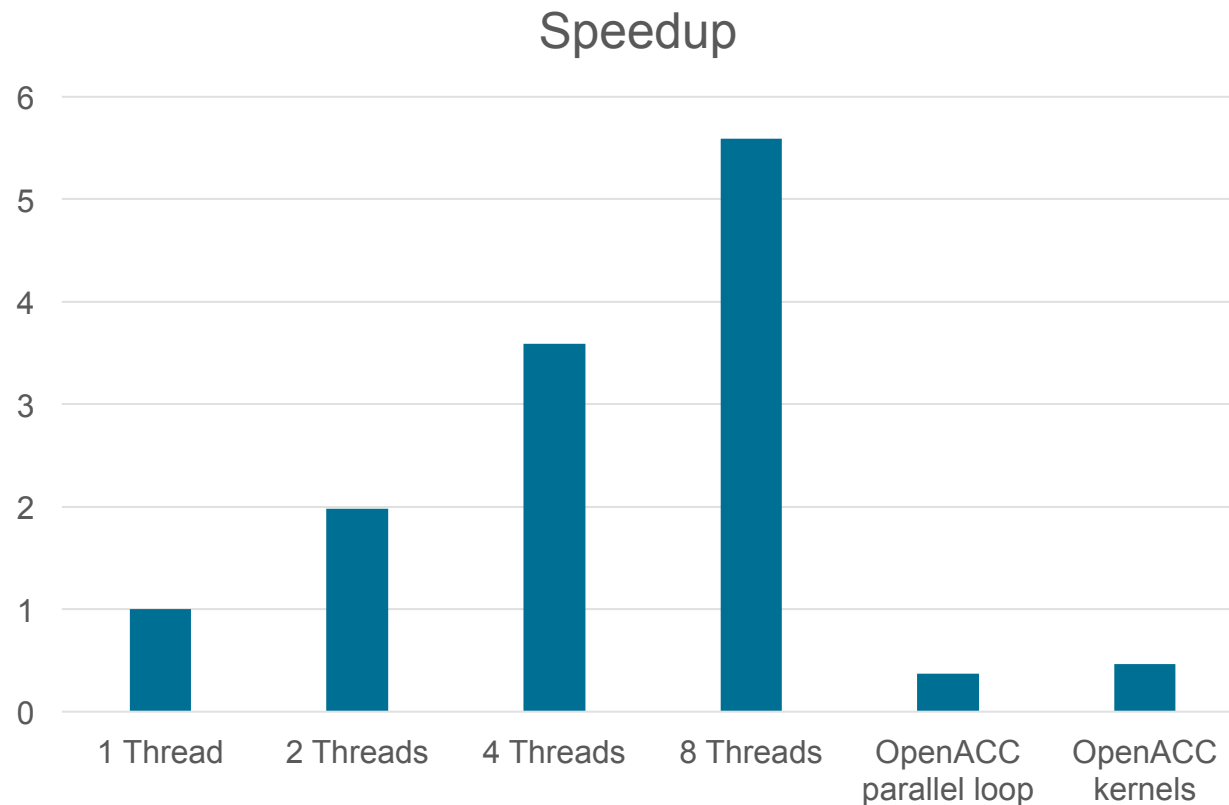


Measured on Judge (Intel Xeon CPU (X5650 Westmere) and NVIDIA Tesla M2070 GPU)



# Parallelize loops with OpenACC

## Comparison with OpenMP shows slow-down



Measured on Jug1 (Intel Xeon CPU (E5-2650) and NVIDIA Tesla K40 GPU)

# Parallelize loops with OpenACC

## Processing flow discovers data issue

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

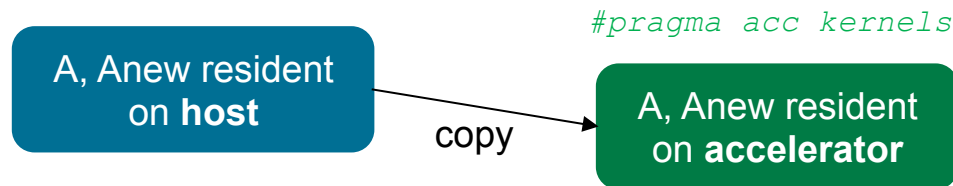
A, Anew resident  
on host

```
...  
    iter++;  
}
```

# Parallelize loops with OpenACC

## Processing flow discovers data issue

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



```
...  
    iter++;  
}
```

# Parallelize loops with OpenACC

## Processing flow discovers data issue

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

A, Anew resident  
on host

copy

*#pragma acc kernels*

A, Anew resident  
on accelerator

```
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;
```

A, Anew resident  
on host

copy

*#pragma acc kernels*

A, Anew resident  
on accelerator

```
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]));
    }
}
```

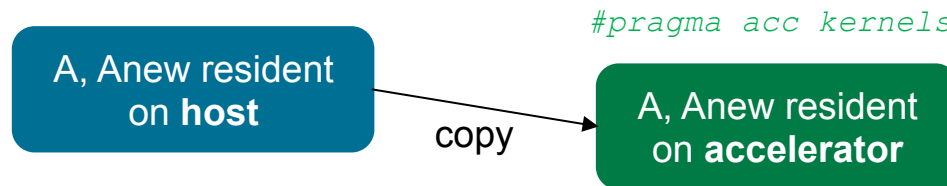
A, Anew resident  
on accelerator

```
...
    iter++;
}
```

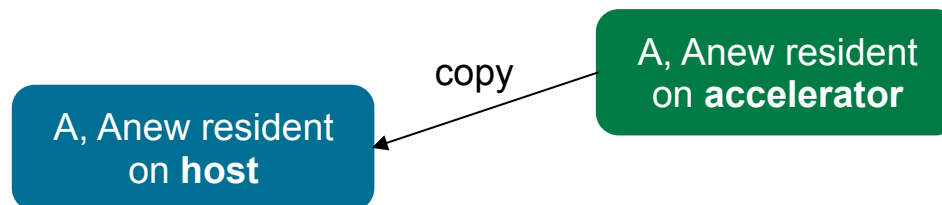
# Parallelize loops with OpenACC

## Processing flow discovers data issue

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



```
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;
```

A, Anew resident  
on host

copy

*#pragma acc kernels*

A, Anew resident  
on accelerator

Copies are done in **each** iteration!

```
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 host

copy

A, Anew resident  
on accelerator

```
...
    iter++;
}
```

# Work flow when using OpenACC

Identify available parallelism



Parallelize loops with OpenACC



Optimize data locality



Optimize loop performance



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

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:

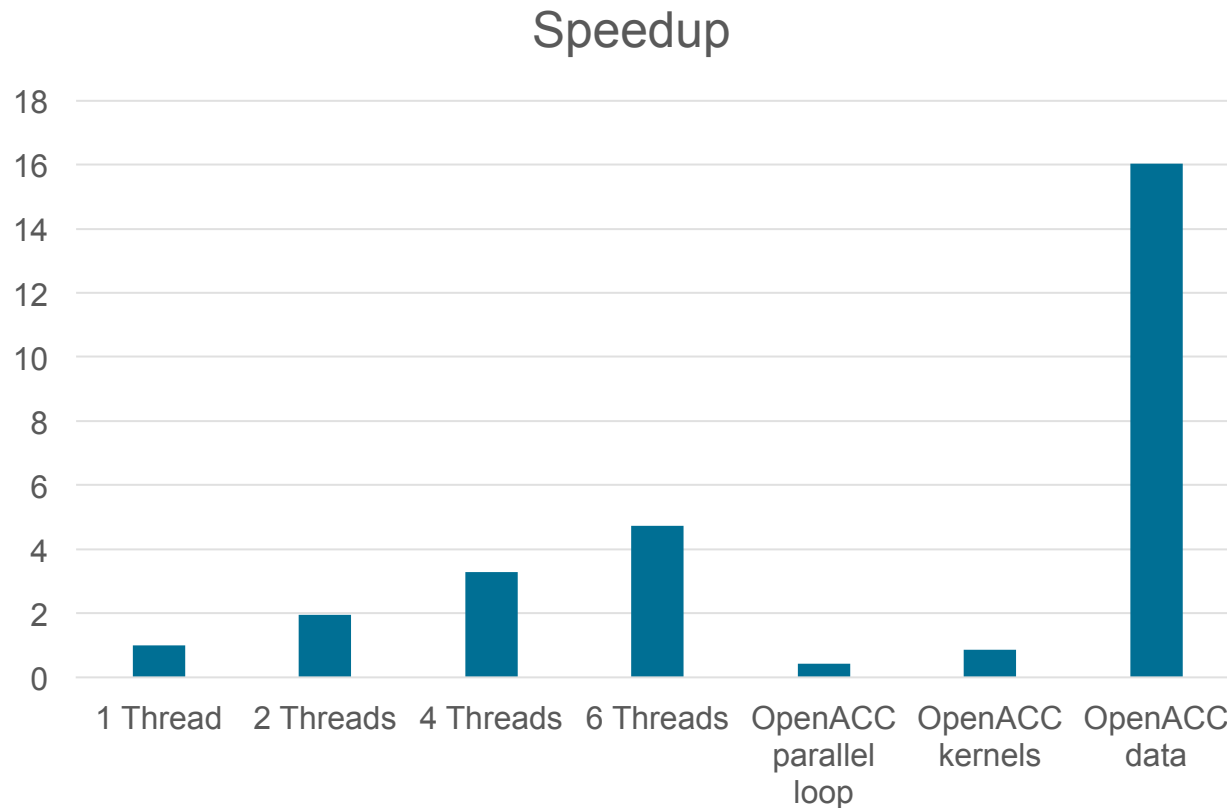
```
#pragma acc data copy(A, Anew)
while ( error > tol && iter < iter_max )
{
    error = 0.0;
#pragma acc kernels
{
    for (int j = jstart; j < jend; j++)
    {
        for( int i = 1; i < M-1; i++ )
        {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                +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

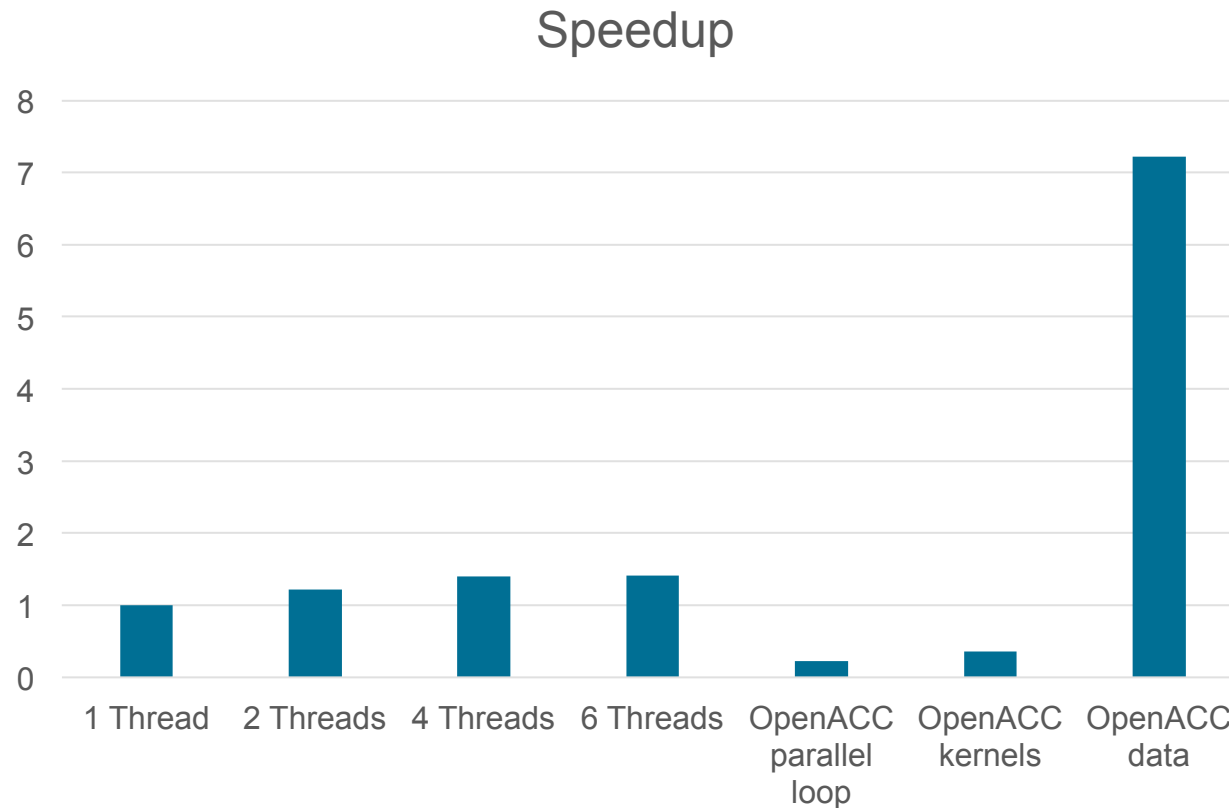


Measured on Jureca (Intel Xeon E5-2680 v3 Haswell and NVIDIA K80 GPU)



# Optimize data locality

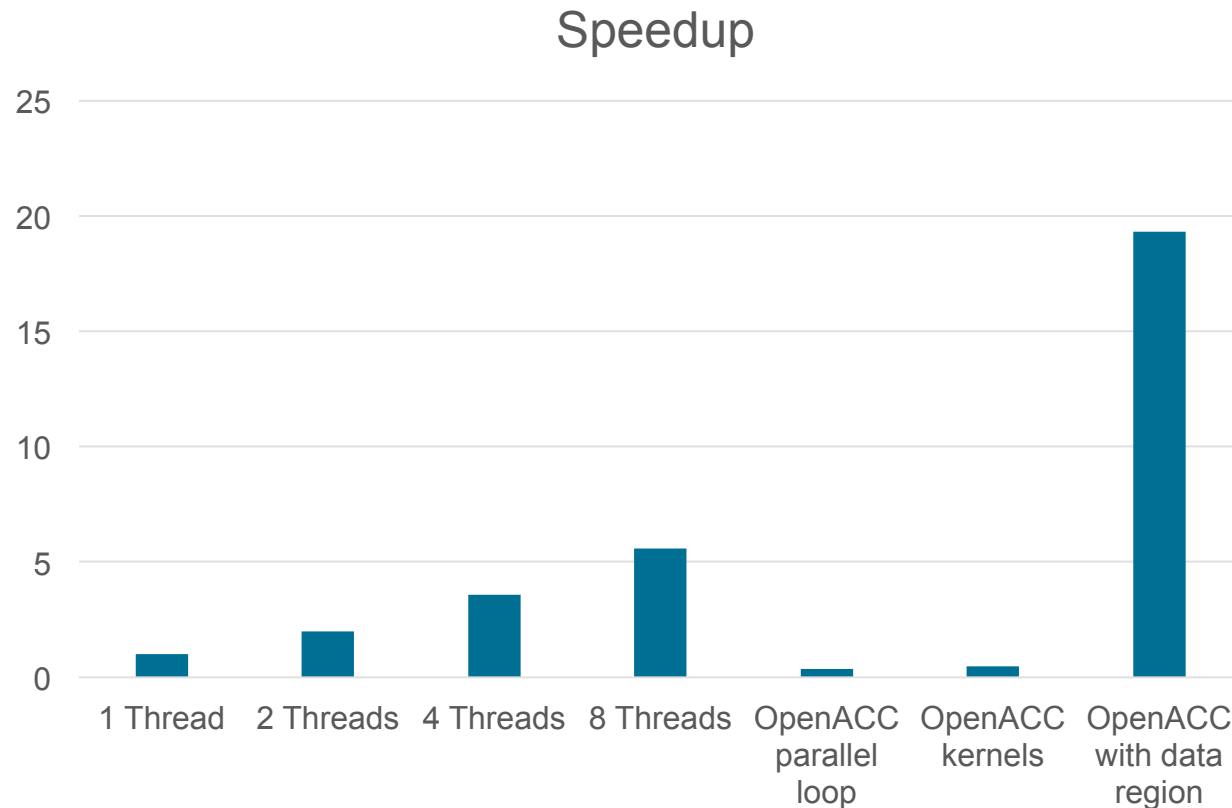
## Comparison with OpenMP shows speedup



Measured on Judge (Intel Xeon CPU (X5650 Westmere) and NVIDIA Tesla M2070 GPU)

# Optimize data locality

## Comparison with OpenMP shows speedup



Measured on Jug1 (Intel Xeon CPU (E5-2650) and NVIDIA Tesla K40 GPU)

# Work flow when using OpenACC

Identify available parallelism



Parallelize loops with OpenACC



Optimize data locality



Optimize loop performance

# Summary

## 1. Identify Available Parallelism:

- What important parts of the code have **available parallelism**?

## 2. Parallelize Loops

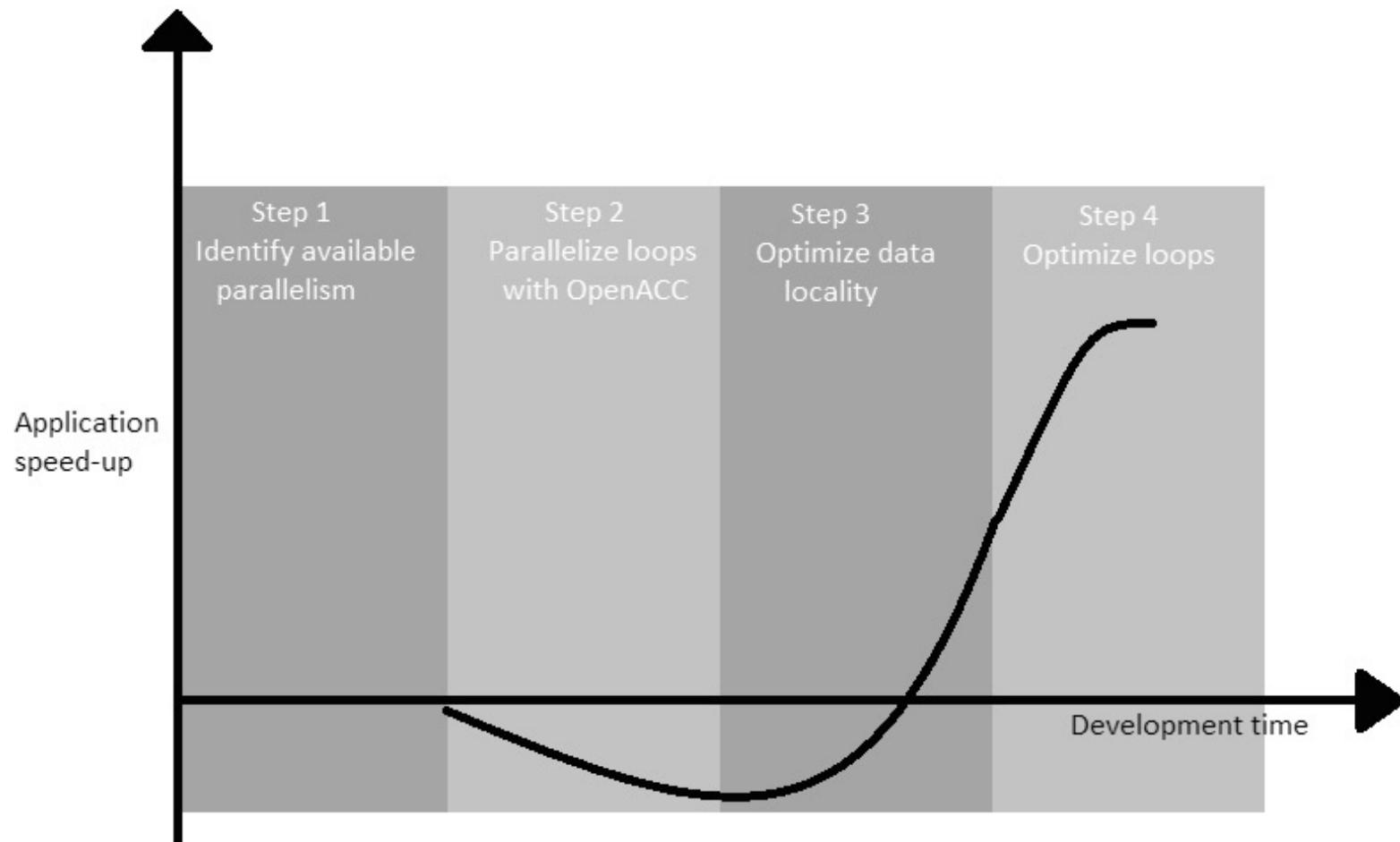
- Express **as much parallelism** as possible and ensure you still get correct results.
- Because the compiler must be **cautious about data movement**, the code might not perform optimally or even slow down.

## 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



# Differentiation from OpenMP 4.5

10/21/16 | A. Severt

# Philosophical Differences

## OpenMP

- More **explicit**
- User-**directed** parallelism
- Compiler has **less** performance responsibility
- User adds additional directives to **exclude parallelization issues**
- Different architectures require **different** directives

## OpenACC

Directives for Accelerators

- More **implicit**
- User-**guided** parallelism
- Compiler has **more** performance responsibility
- Users write code actually **free of parallelization issues**
- Higher-level directives allow **same-code** targeting of different architectures

# Some OpenACC Directives/Clauses translate 1:1...

## OpenMP

- `acc parallel`
- `acc loop vector`
- `acc data`
- `acc update`
- `acc copy/copyin/copyout`
- ...

## OpenACC

- `omp target teams, parallel`
- `omp simd`
- `omp target data`
- `omp target update`
- `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