# Example (3/6)

```
1  program p_example
2    implicit none
3    integer, parameter :: dp = selected_real_
4    integer, parameter :: nel = 1000
5    real(dp), dimension(nel, nel) :: a
6    integer :: i, j
7
8
9    !$acc parallel loop
10   do i = 1, nel
11
12     do j = 1, nel
13       a(j, i) = 0.5_dp*(i*nel+j)
14     end do
15   end do
16   !$acc end parallel loop
17
18   write(*, *) a(10, 10)
19 end program p_example
```

```
1  NVCOMPILER_ACC_TIME=1 ./p_example
```

LOG

⇒ Works, but we can be more specific about distribution

# Example (4/6)

```
1  program p_example
2    implicit none
3    integer, parameter :: dp = selected_real_
4    integer, parameter :: nel = 1000
5    real(dp), dimension(nel, nel) :: a
6    integer :: i, j
7
8    !$acc parallel
9    !$acc loop gang
10   do i = 1, nel
11     !$acc loop vector
12     do j = 1, nel
13       a(j, i) = 0.5_dp*(i*nel+j)
14     end do
15   end do
16   !$acc end parallel
17
18   write(*, *) a(10, 10)
19 end program p_example
```

```
1  NVCOMPILER_ACC_TIME=1 ./p_example
```

LOG

⇒ All relevant do-loops annotated

# Example (5/6)

```
 1 program p_example
 2   implicit none
 3   integer, parameter :: dp = selected_real_
 4   integer, parameter :: nel = 1000
 5   real(dp), dimension(nel, nel) :: a
 6   integer :: i, j
 7
 8
 9   !$acc parallel loop gang vector
10   do i = 1, nel
11
12     do j = 1, nel
13       a(j, i) = 0.5_dp*(i*nel+j)
14     end do
15   end do
16   !$acc end parallel loop
17
18   write(*, *) a(10, 10)
19 end program p_example
```

```
1 NVCOMPILER_ACC_TIME=1 ./p_example
```

LOG

⇒ Works, but we can also specify type for each loop

# Example (6/6)

```
 1 program p_example
 2   implicit none
 3   integer, parameter :: dp = selected_real_
 4   integer, parameter :: nel = 1000
 5   real(dp), dimension(nel, nel) :: a
 6   integer :: i, j
 7
 8
 9   !$acc parallel loop gang vector collapse(
10   do i = 1, nel
11
12     do j = 1, nel
13       a(j, i) = 0.5_dp*(i*nel+j)
14     end do
15   end do
16   !$acc end parallel loop
17
18   write(*, *) a(10, 10)
19 end program p_example
```

```
1 NVCOMPILER_ACC_TIME=1 ./p_example
```

LOG

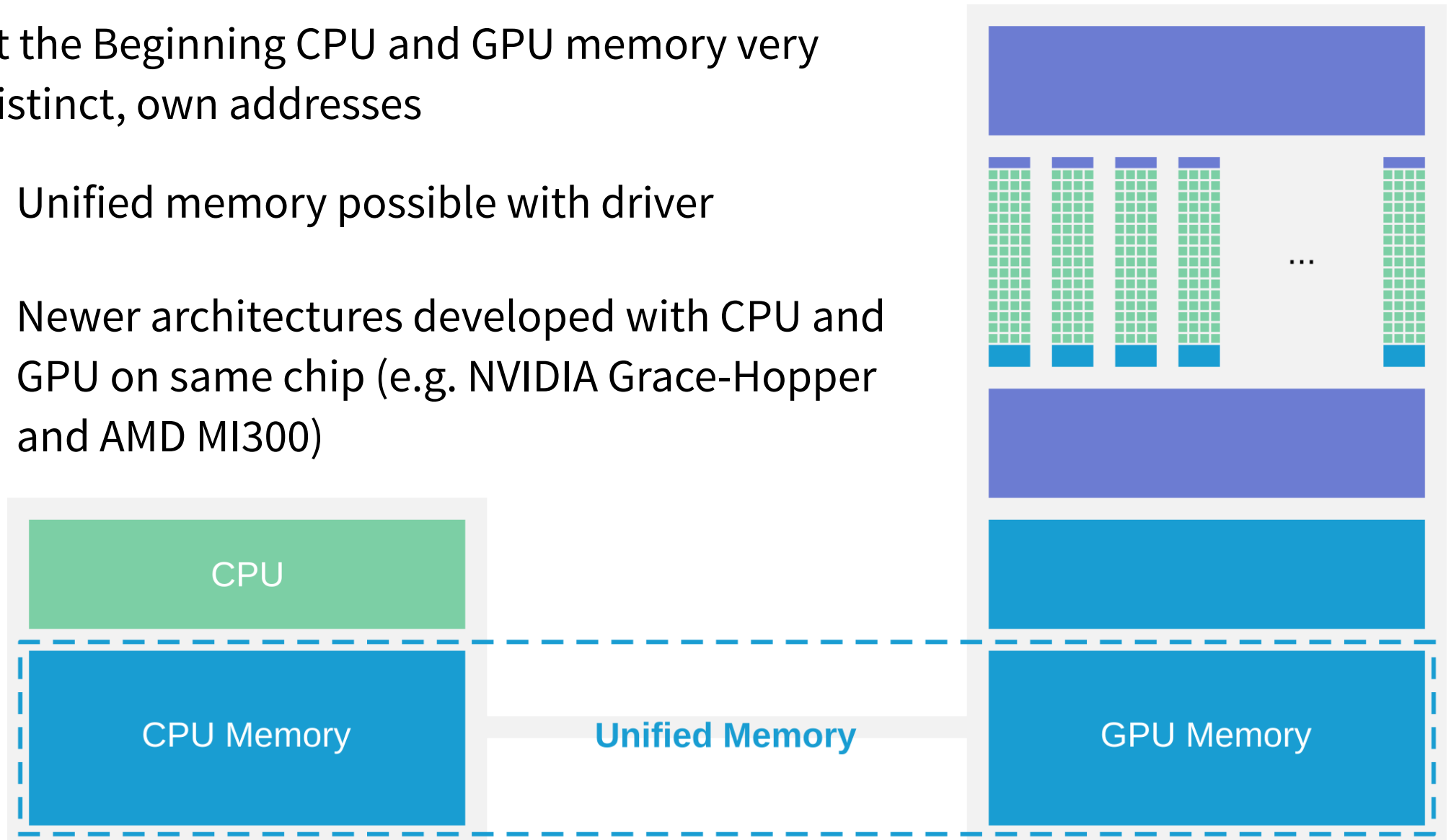
$\Rightarrow$ Collapse if possible

# OpenACC data movement

# CPU and GPU memory

At the Beginning CPU and GPU memory very distinct, own addresses

- Unified memory possible with driver
- Newer architectures developed with CPU and GPU on same chip (e.g. NVIDIA Grace-Hopper and AMD MI300)

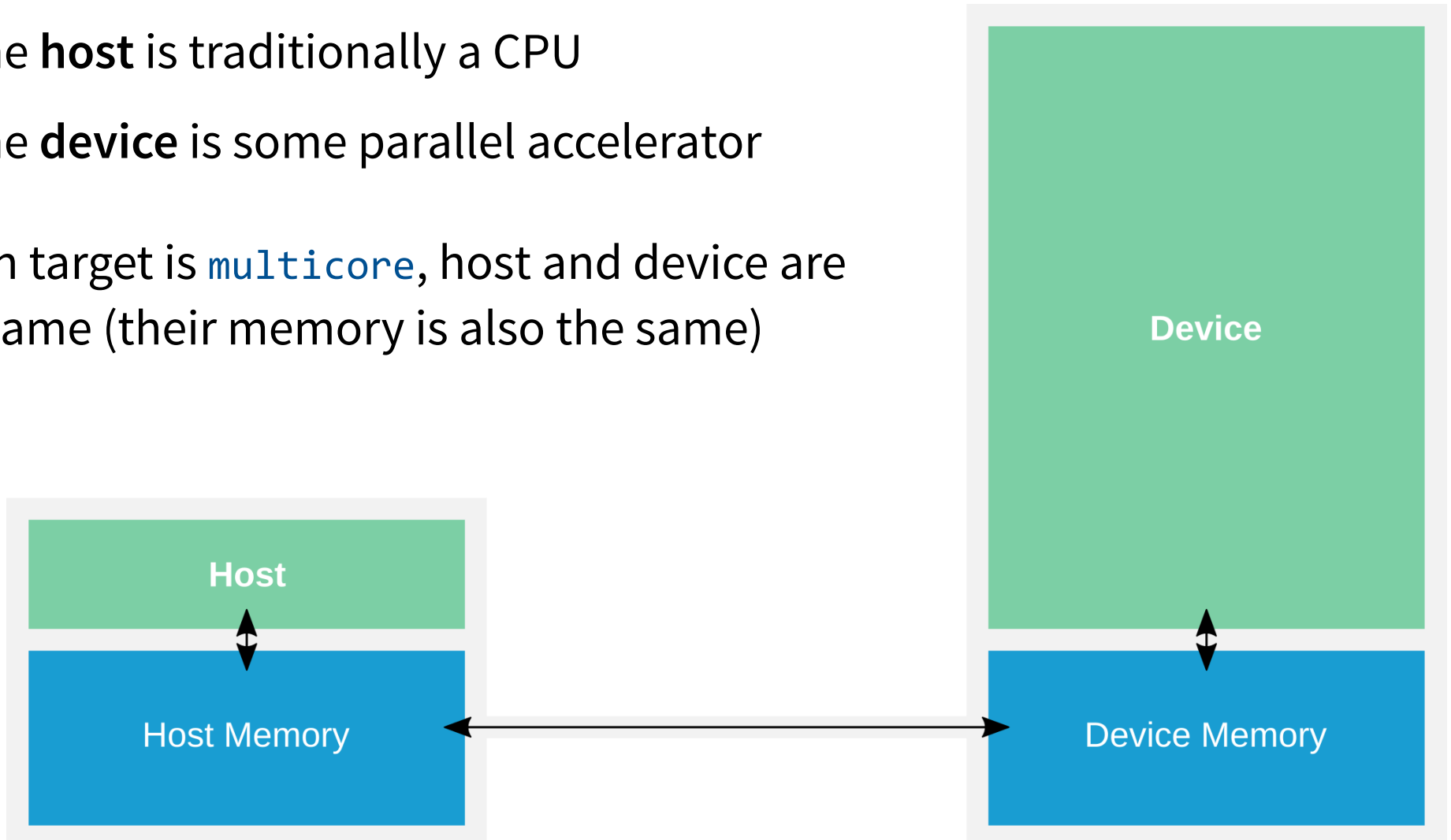



*Source: PRACE P2_44*

# Data management overview

- The **host** is traditionally a CPU
- The **device** is some parallel accelerator

When target is `multicore`, host and device are the same (their memory is also the same)




*Source: OpenACC O4_9*

# Data statements

```
1  !$acc data [clause [, clause]... ]
2
3  ! ...
4
5  !$acc end data
```

- Defines region of code in which data remains on device
- Data is shared among all kernels in region
- Explicit data transfers
- Transferring between these two memories can be a very time consuming process
- Clauses to augment the data regions

*Source: PRACE P2_55, OpenACC O5_5*

# Data clauses

`copy( <list> )`

- Allocates memory on GPU
- Copies data from host to GPU when entering region
- Copies data to the host when exiting region

Variants:

- `copyin( <list> )`: No copying to host at the end
- `copyout( <list> )`: No copying to GPU at the beginning
- `create( <list> )`: Only memory allocation, no copying

*Note: Scalars are first private by default, they rarely appear in a data clause.*

*Source: OpenACC O4_22; Jacob M1_16*

# Update clauses

*Note: The clauses from the previous slide have no effect if the data is already on the device! Within data regions host and device memory can be updated with the* `update` *clause.*

```
update device( <list> )
```

- Copy data from host to device

`update self( <list> )` or `update host( <list> )`

- Copy data from device to host

*Source: Jacob M1_17*

# Data movement (1/4)

```
21  ! ...
22  iter = 0
23
24  do iter = 1, iter_max
25    error = 0.0_dp
26
27
28    do j = 2, m-1
29      do i = 2, n-1
30        Anew(i, j) = 0.25_dp * (A(i+1, j) &
31          + A(i-1, j) + A(i, j-1) + A(i, j+1)
32        error = max(error, abs(Anew(i, j)-A(i
33      end do
34    end do
35
36
37    A = Anew
38
39    if (error <= tol) exit
40  end do
41
42  ! ...
```

*Source: PRACE P2_60*

Example code with

- outer convergence loop
- Stencil operation within nested loop
- Array assignment

# Data movement (2/4)

```
21  ! ...
22  iter = 0
23
24  do iter = 1, iter_max
25    error = 0.0_dp
26    !$acc parallel loop &
27    !$acc reduction(max:error)
28    do j = 2, m-1
29      do i = 2, n-1
30        Anew(i, j) = 0.25_dp * (A(i+1, j) &
31          + A(i-1, j) + A(i, j-1) + A(i, j+1)
32        error = max(error, abs(Anew(i, j)-A(i
33      end do
34    end do
35    !$acc end parallel loop
36    !$acc kernels
37    A = Anew
38    !$acc end kernels
39    if (error <= tol) exit
40  end do
41
42  ! ...
```

*Source: PRACE P2_60*

Porting

- stencil with `parallel` construct
- Array assignment with `kernels` construct

# Data movement (3/4)

```
21  ! ...
22  iter = 0
23
24  do iter = 1, iter_max
25    error = 0.0_dp
26    !$acc parallel loop copyin(A) copy(Anew)
27    !$acc reduction(max:error)
28    do j = 2, m-1
29      do i = 2, n-1
30        Anew(i, j) = 0.25_dp * (A(i+1, j) &
31          + A(i-1, j) + A(i, j-1) + A(i, j+1)
32        error = max(error, abs(Anew(i, j)-A(i
33      end do
34    end do
35    !$acc end parallel loop
36    !$acc kernels copyin(Anew) copy(A)
37    A = Anew
38    !$acc end kernels
39    if (error <= tol) exit
40  end do
41
42  ! ...
```

*Source: PRACE P2_60*

Data movement

- for each kernel with `copyin` and `copy` clauses
- states intent but does not improve performance

# Data movement (4/4)

```
21  ! ...
22  iter = 0
23  !$acc data copy(A, Anew)
24  do iter = 1, iter_max
25    error = 0.0_dp
26    !$acc parallel loop default(present) &
27    !$acc reduction(max:error)
28    do j = 2, m-1
29      do i = 2, n-1
30        Anew(i, j) = 0.25_dp * (A(i+1, j) &
31          + A(i-1, j) + A(i, j-1) + A(i, j+1)
32        error = max(error, abs(Anew(i, j)-A(i
33      end do
34    end do
35    !$acc end parallel loop
36    !$acc kernels default(present)
37    A = Anew
38    !$acc end kernels
39    if (error <= tol) exit
40  end do
41  !$acc end data
42  ! ...
```

*Source: PRACE P2_60*

Data movement

- combined movement of all non-scalar variables
- explicitly outside the loop

# Unstructured data clauses

`enter data` and `exit data` directives can be used to manage unstructured data regions, e.g. `!$acc enter data create(a)`

Additional data clause `delete`, e.g. `!$acc exit data delete(var)` to deallocate `var` on GPU

*Source: Jacob M1_25*

# Alternative OpenMP

# OpenACC - OpenMP

Most OpenACC directives can be directly translated to OpenMP 5.x `target` directives:

| OpenACC | OpenMP |
|---|---|
| `!$acc parallel` | `!$omp target teams distribute` |
| `!$acc parallel loop` | `!$omp target teams distribute parallel for` |
| `!$acc data` | `!$omp target data` |
| `!$acc data copyin` | `!$omp target data map(to:)` |
| `!$acc update host` | `!$omp target update from()` |

# Further training

# GPU hackathons

- A great way to learn GPU porting techniques and at the same time get started with porting your application are GPU hackathons
- For example there are annual hackathons hosted by Helmholtz or CSCS in Lugano
  - Helmholtz GPU Hackathon 2024
  - CSCS EuroHack 2024

# Training courses

- JSC online training course “Directive-based GPU programming with OpenACC” 29.10.-01.11.2024
- natESM Technical Training “Performance analysis and GPU programming (CUDA, OpenACC, OpenMP, kokkos)” 05.-06.11.2024 at JSC