



# OpenMP 6.0 Part 2: New Accelerator Features

Christian Terboven, RWTH Aachen University

Michael Klemm, OpenMP ARB

HORIZON-EUROHPC-JU-2023-COE



**EuroHPC**  
Joint Undertaking

1 January 2024– 31 December 2026

Grant Agreement No 101143931

# Agenda

- Who are Michael and Christian? (C+M)
- Review of OpenMP for Accelerators (C)
- Notable OpenMP 5.x accelerator extensions (C)
- Loop construct in the spotlight and Array syntax (M)
- Error and Requires (C)
- Unified Memory (M)
- Memory Management (C)
- Interoperability between OpenMP and HIP/CUDA (M)
- Q&A (M+C)



# *Who are Michael and Christian?*



# Michael and Christian

## ■ Michael ...

- Principal Member of Technical Staff
- Works in HPC since 2003
- Works on the Fortran OpenMP offload compiler for AMD Instinct™ Accelerators
- Is a member of the OpenMP language committee since 2009
- Chief Executive Officer of the OpenMP ARB since April 2016



# Michael and Christian

## ■ Michael ...

- Principal Member of Technical Staff
- Works in HPC since 2003
- Works on the Fortran OpenMP offload compiler for AMD Instinct™ Accelerators
- Is a member of the OpenMP language committee since 2009
- Chief Executive Officer of the OpenMP ARB since April 2016

## ■ Christian ...

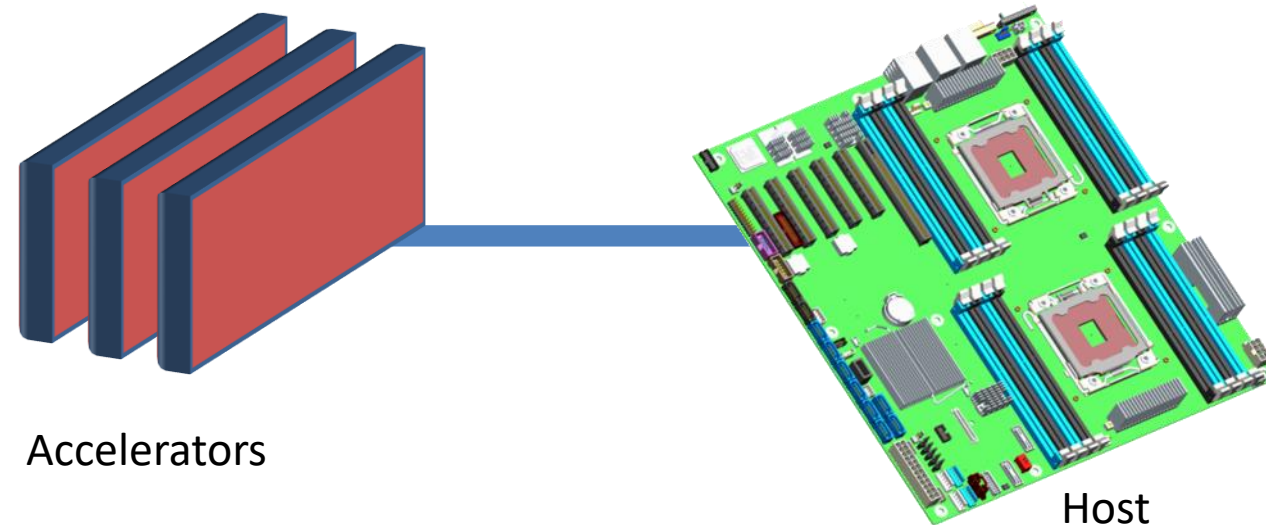
- ... is a senior scientist at RWTH Aachen University and leads the HPC group
- ... does research on Parallel Programming and Performance
- ... is a member of the OpenMP language committee since 2008 and co-chair of the Affinity subcom.
- is co-author of the book "Using OpenMP - The Next Step", published by MIT Press

# *Review of OpenMP for Accelerators*



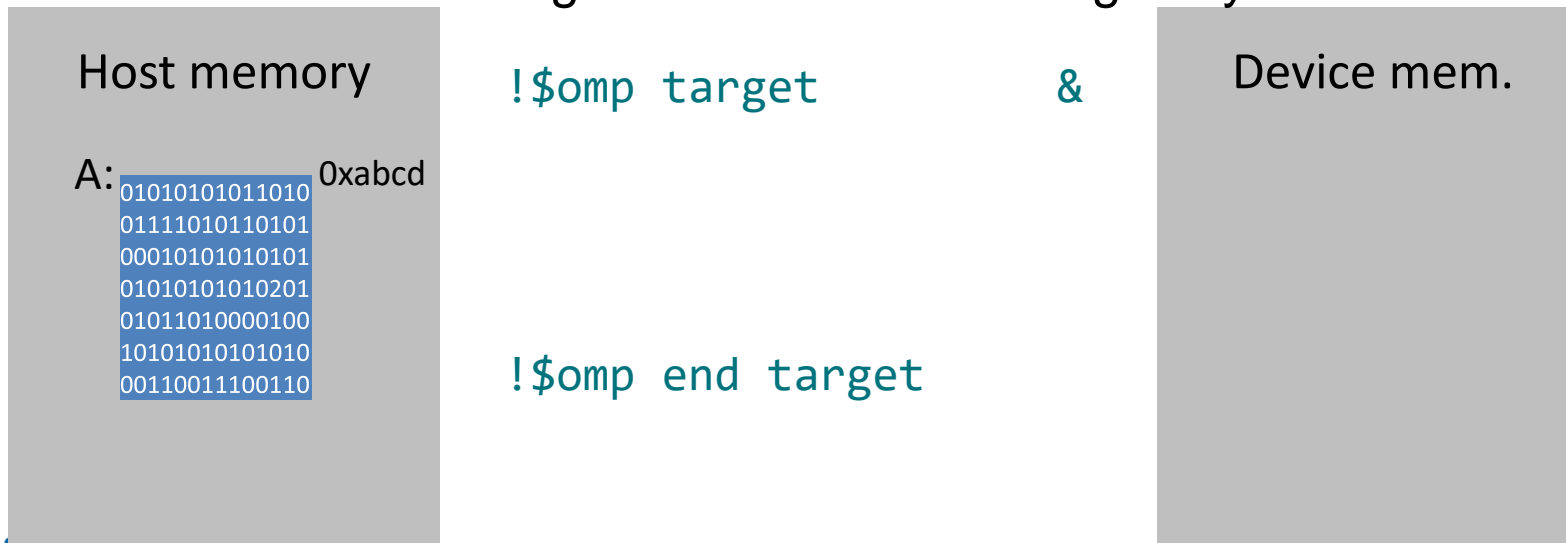
# Device Model

- Since version 4.0 the OpenMP API supports accelerators/coprocessors
- Device model:
  - One host for “traditional” multi-threading
  - Multiple accelerators/coprocessors of the same kind for offloading



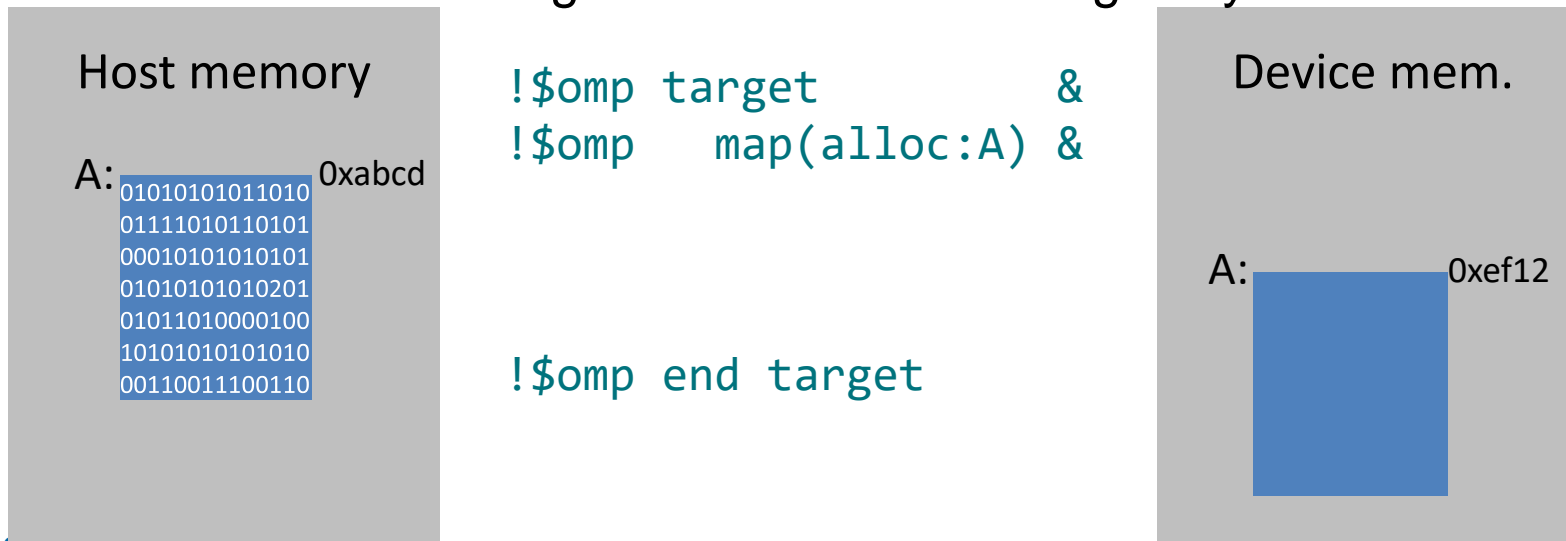
# OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - Data environment is created at the opening curly brace
  - Data environment is automatically destroyed at the closing curly brace
  - Data transfers (if needed) are done at the curly braces, too:
    - Upload data from the host to the target device at the opening curly brace.
    - Download data from the target device at the closing curly brace.



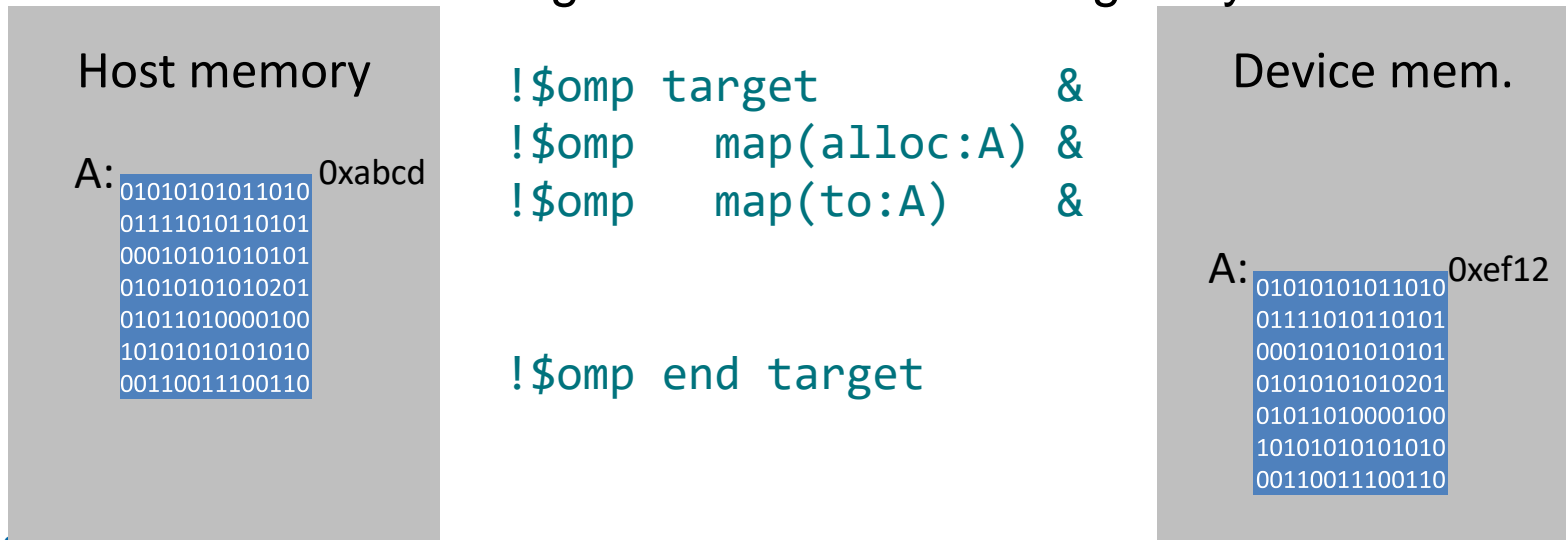
# OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - Data environment is created at the opening curly brace
  - Data environment is automatically destroyed at the closing curly brace
  - Data transfers (if needed) are done at the curly braces, too:
    - Upload data from the host to the target device at the opening curly brace.
    - Download data from the target device at the closing curly brace.



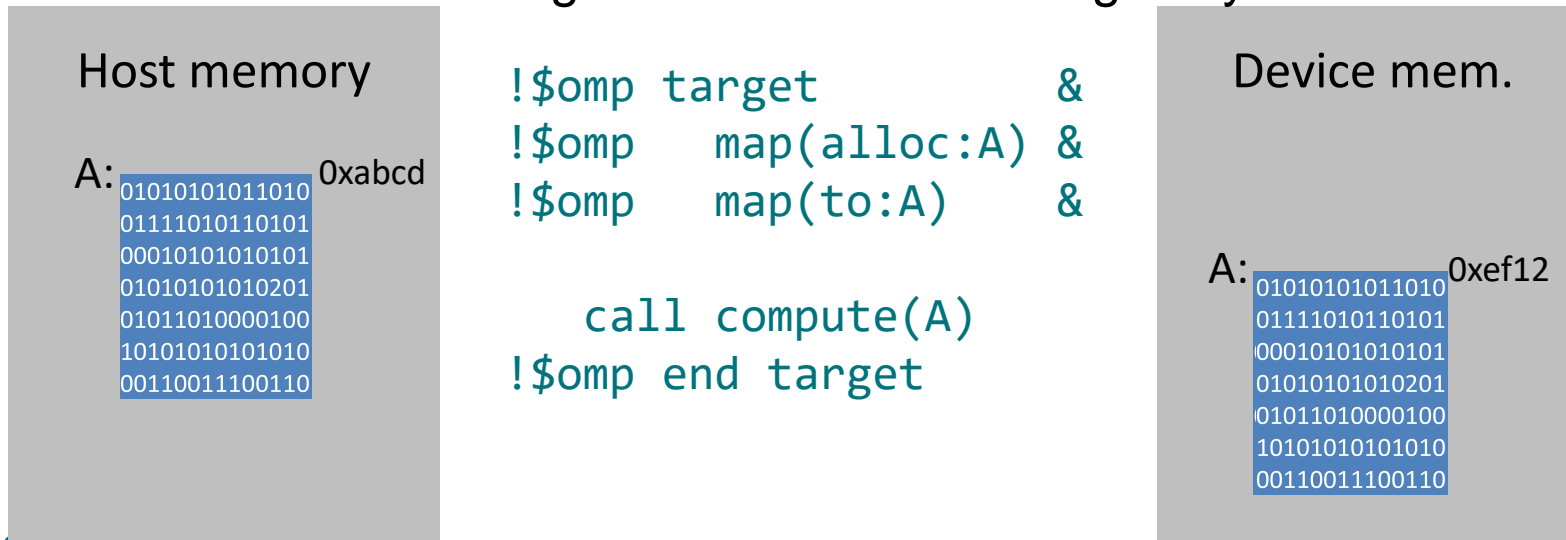
# OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - Data environment is created at the opening curly brace
  - Data environment is automatically destroyed at the closing curly brace
  - Data transfers (if needed) are done at the curly braces, too:
    - Upload data from the host to the target device at the opening curly brace.
    - Download data from the target device at the closing curly brace.



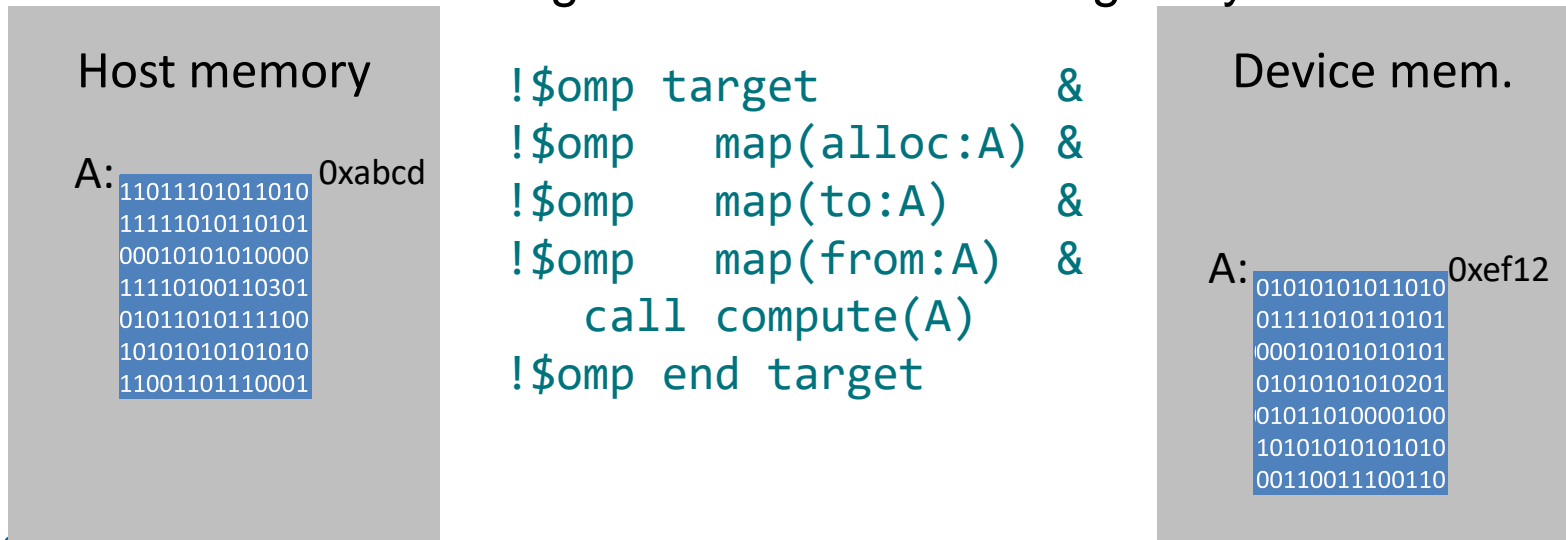
# OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - Data environment is created at the opening curly brace
  - Data environment is automatically destroyed at the closing curly brace
  - Data transfers (if needed) are done at the curly braces, too:
    - Upload data from the host to the target device at the opening curly brace.
    - Download data from the target device at the closing curly brace.



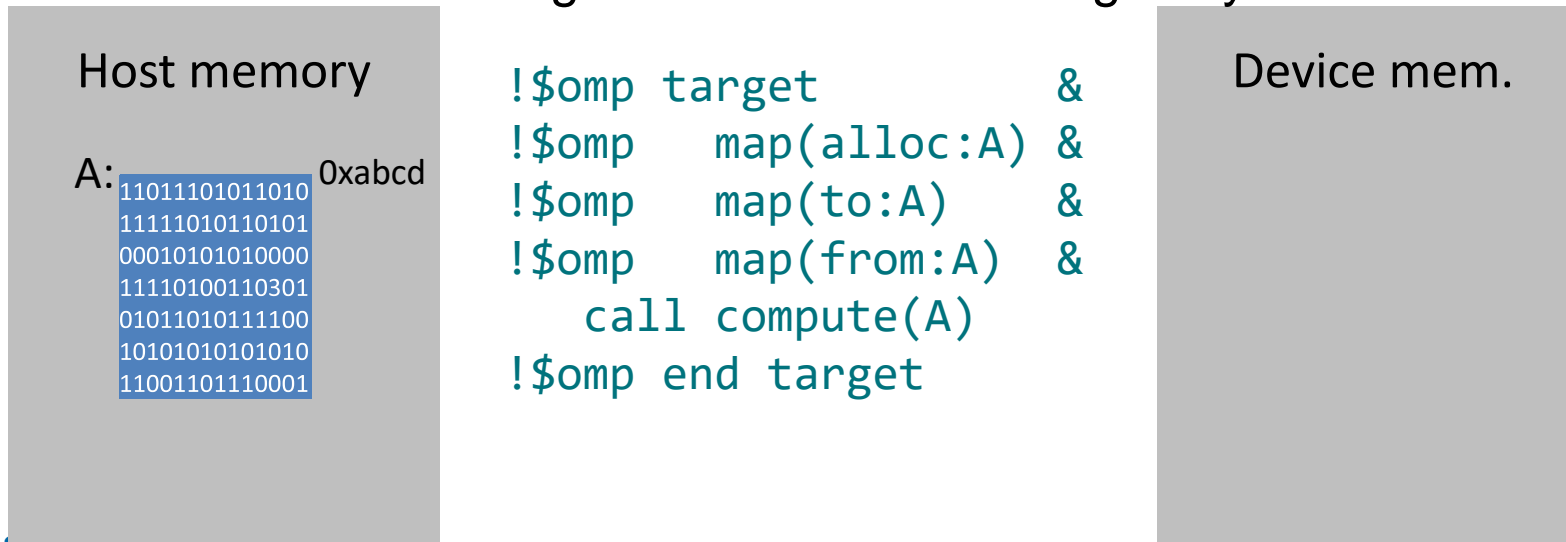
# OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - Data environment is created at the opening curly brace
  - Data environment is automatically destroyed at the closing curly brace
  - Data transfers (if needed) are done at the curly braces, too:
    - Upload data from the host to the target device at the opening curly brace.
    - Download data from the target device at the closing curly brace.



# OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - Data environment is created at the opening curly brace
  - Data environment is automatically destroyed at the closing curly brace
  - Data transfers (if needed) are done at the curly braces, too:
    - Upload data from the host to the target device at the opening curly brace.
    - Download data from the target device at the closing curly brace.



# Creating Parallelism on the Target Device

- OpenMP separates offload and parallelism
  - Programmers need to explicitly create parallel regions on the target device
  - In theory, this can be combined with any OpenMP construct
  - In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

# Creating Parallelism on the Target Device

## ■ OpenMP separates offload and parallelism

- Programmers need to explicitly create parallel regions on the target device
- In theory, this can be combined with any OpenMP construct
- In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

```
void saxpy(int n, float a,
          float* x, float* y) {
    #pragma omp target map(to:x[0:sz]) \
                    map(tofrom(y[0:sz]))
    #pragma omp parallel for simd
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

host  
target  
host

GPUs are multi-level devices:  
SIMD, threads, thread blocks

Create a team of threads to execute the loop in  
parallel using SIMD instructions.



# Creating Parallelism on the Target Device

## ■ OpenMP separates offload and parallelism

- Programmers need to explicitly create parallel regions on the target device
- In theory, this can be combined with any OpenMP construct
- In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

```
void saxpy(int n, float a,  
          float* x, float* y) {  
    #pragma omp target map(to:x[0:sz]) \  
                      map(tofrom(y[0:sz]))  
    #pragma omp parallel for simd  
    for (int i = 0; i < sz; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

host  
target  
host

GPUs are multi-level devices:  
SIMD, threads, thread blocks

Create a team of threads to execute the loop in  
parallel using SIMD instructions.



# Creating Parallelism on the Target Device

## ■ OpenMP separates offload and parallelism

- Programmers need to explicitly create parallel regions on the target device
- In theory, this can be combined with any OpenMP construct
- In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

```
void saxpy(int n, float a,  
          float* x, float* y) {  
    #pragma omp target map(to:x[0:sz]) \  
                      map(tofrom(y[0:sz]))  
    #pragma omp parallel for simd  
    for (int i = 0; i < sz; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

host  
target  
host

GPUs are multi-level devices:  
SIMD, threads, thread blocks

Create a team of threads to execute the loop in  
parallel using SIMD instructions.

# Creating Parallelism on the Target Device

- OpenMP separates offload and parallelism
  - Programmers need to explicitly create parallel regions on the target device
  - In theory, this can be combined with any OpenMP construct
  - In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.
  
- Better: Tile the loop into an outer loop and an inner loop
  - Assign the outer loop to “teams”.
  - Assign the inner loop to the “threads”.
  - (Assign the inner loop to SIMD units.)

# Multi-level Parallel saxpy

- For convenience, OpenMP defines composite constructs to implement the required code transformations

```
void saxpy(int n, float a, float* x, float* y) {  
    #pragma omp target teams distribute parallel for simd \  
        num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])  
    for (int i = 0; i < n; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

```
subroutine saxpy(n, a, x, y)  
    ! Declarations omitted  
!$omp omp target teams distribute parallel do simd &  
!$omp&        num_teams(num_blocks) map(to:x) map(tofrom:y)  
    do i = 1, n  
        y(i) = a * x(i) + y(i)  
    end do  
!$omp end target teams distribute parallel do simd  
end subroutine
```

# *Notable OpenMP 5.x accelerator extensions*

# Mapping non-rectangular data

## ■ Extension of the array section syntax.

→ Jagged array: each row can have a different length (=> non-rectangular)

```
#define NROWS 3
int **jagged_array;
int row_lengths[NROWS] = {2, 3, 4};    // Varying row lengths

/* allocation and initialization omitted */

#pragma omp target map(to: jagged_array[0:NROWS])
                    map(to: jagged_array[0:NROWS][0:row_lengths[:]])
```

→ First map clause maps the array of pointers itself

→ Second map clause maps the actual data

→ `row_lengths[:]` creates an array section representing the lengths of all rows

# User-defined mappers

## ■ OpenMP 5.0 introduced the capability for user-defined mappers

→ The `declare mapper` directive allows to define own mappers to encapsulate complex mapping logic, optimizing data movement and access patterns for complex data structures

→ Declaration of a user-defined mapper for `myvec`

```
typedef struct myvec{
    size_t len;
    double *data;
} myvec_t;

#pragma omp declare mapper(vecpos: myvec_t v) map(v, v.pos[0:v.len])
#pragma omp declare mapper(vecvel: myvec_t v) map(v, v.vel[0:v.len])
```

→ Usage of the user-defined mapper

```
#pragma omp target map(mapper(vecvel), tofrom:s)
do_something_with_s(&s);
```

Example taken from OpenMP 6.0 example collection



# *Loop construct and Fortran array syntax*

# Multi-level Parallel saxpy

- For convenience, OpenMP defines composite constructs to implement the required code transformations

```
void saxpy(int n, float a, float* x, float* y) {  
    #pragma omp target teams distribute parallel for simd \  
        num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])  
    for (int i = 0; i < n; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

```
subroutine saxpy(n, a, x, y)  
    ! Declarations omitted  
!$omp omp target teams distribute parallel do simd &  
!$omp&        num_teams(num_blocks) map(to:x) map(tofrom:y)  
    do i = 1, n  
        y(i) = a * x(i) + y(i)  
    end do  
!$omp end target teams distribute parallel do simd  
end subroutine
```

# Multi-level Parallel saxpy

- For convenience, OpenMP defines composite constructs to implement the required code transformations

Still a lot of typing required!

```
void saxpy(int n, float a, float* x, float* y) {  
    #pragma omp target teams distribute parallel for simd \  
        num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])  
    for (int i = 0; i < n; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

```
subroutine saxpy(n, a, x, y)  
    ! Declarations omitted  
!$omp omp target teams distribute parallel do simd &  
!$omp&        num_teams(num_blocks) map(to:x) map(tofrom:y)  
    do i = 1, n  
        y(i) = a * x(i) + y(i)  
    end do  
!$omp end target teams distribute parallel do simd  
end subroutine
```

# Multi-level Parallel saxpy

- OpenMP loop construct simplifies expressing parallel loops.
  - Compiler can exploit parallelism for further optimizations

```
void saxpy(int n, float a, float* x, float* y) {  
    #pragma omp target teams loop \  
        num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])  
    for (int i = 0; i < n; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

```
subroutine saxpy(n, a, x, y)  
    ! Declarations omitted  
    !$omp omp target teams loop &  
    !$omp&        num_teams(num_blocks) map(to:x) map(tofrom:y)  
    do i = 1, n  
        y(i) = a * x(i) + y(i)  
    end do  
    !$omp end target teams distribute parallel do simd  
end subroutine
```

# Supporting Array Notation on the GPU

```
subroutine saxpy(n, a, x, y)
  use iso_fortran_env
  implicit none

  integer :: n
  real(kind=real32) :: a
  real(kind=real32), dimension(n) :: x, y

  y = a * x + y

end subroutine saxpy
```



# Supporting Array Notation on the GPU

```
subroutine saxpy(n, a, x, y)
  use iso_fortran_env
  implicit none

  integer :: n
  real(kind=real32) :: a
  real(kind=real32), dimension(n) :: x, y

  y = a * x + y
end subroutine saxpy
```

```
subroutine axpy_array(n, a, x, y)
  use iso_fortran_env
  implicit none

  integer :: n
  real(kind=real32) :: a
  real(kind=real32), dimension(n) :: x, y

  !$omp target teams loop
  do i = 1, n
    y(i) = a * x(i) + y(i)
  end do
end subroutine axpy_array
```

# Supporting Array Notation on the GPU

```
subroutine saxpy(n, a, x, y)
  use iso_fortran_env
  implicit none

  integer :: n
  real(kind=real32) :: a
  real(kind=real32), dimension(n) :: x, y

  !$omp target teams workdistribute
  y = a * x + y
  !$omp end target teams workdistribute
end subroutine saxpy
```



# Automatically Mapping ALLOCATABLEs

```
module mod
  integer, parameter :: rk = 8
  real(kind=rk), dimension(:), allocatable, save :: data
  !$omp declare target enter(      data)
contains
  subroutine alloc(n)
    implicit none
    integer :: n
    allocate(data(n))
    !$omp target enter data map(alloc:data)
  end subroutine

  subroutine dealloc()
    implicit none
    !$omp target exit data map(delete:data)
    deallocate(data)
  end subroutine
end module
```

# Automatically Mapping ALLOCATABLEs

```
module mod
  integer, parameter :: rk = 8
  real(kind=rk), dimension(:), allocatable, save :: data
  !$omp declare target enter(automap:data)
contains
  subroutine alloc(n)
    implicit none
    integer :: n
    allocate(data(n))
    !$omp target enter data map(alloc:data)
  end subroutine

  subroutine dealloc()
    implicit none
    !$omp target exit data map(delete:data)
    deallocate(data)
  end subroutine
end module
```

- “automap” greatly simplifies dealing with ALLOCATABLE objects

# Automatically Mapping ALLOCATABLEs

```
module mod
  integer, parameter :: rk = 8
  real(kind=rk), dimension(:), allocatable, save :: data
  !$omp declare target enter(automap:data)
contains
  subroutine alloc(n)
    implicit none
    integer :: n
    allocate(data(n))
    !$omp target enter data map(alloc:data)
  end subroutine

  subroutine dealloc()
    implicit none
    !$omp target exit data map(delete:data)
    deallocate(data)
  end subroutine
end module
```

- “automap” greatly simplifies dealing with ALLOCATABLE objects
  - ALLOCATE automatically allocates an object in GPU memory
  - DEALLOCATE automatically deallocates an object in GPU memory
  - Also works for implicit allocations/deallocations as part of intrinsic assignments, etc.

# Automatically Mapping ALLOCATABLEs

```
module mod
  integer, parameter :: rk = 8
  real(kind=rk), dimension(:), allocatable, save :: data
  !$omp declare target enter(automap:data)
contains
  subroutine alloc(n)
    implicit none
    integer :: n
    allocate(data(n))
    !$omp target enter data map(alloc:data)
  end subroutine

  subroutine dealloc()
    implicit none
    !$omp target exit data map(delete:data)
    deallocate(data)
  end subroutine
end module
```

- “automap” greatly simplifies dealing with ALLOCATABLE objects
  - ALLOCATE automatically allocates an object in GPU memory
  - DEALLOCATE automatically deallocates an object in GPU memory
  - Also works for implicit allocations/deallocations as part of intrinsic assignments, etc.
- Restrictions
  - Only works for variables with SAVE attribute
  - declare target directive must be visible at the ALLOCATION/DEALLOCATION statement

# *Error and Requires*

# Requires directive / 1

- The `requires` directive is used to express feature dependencies or requirements for specific regions of OpenMP code

→ `unified_shared_memory`: USM implies that host and device memory spaces are logically unified

```
#pragma omp requires unified_shared_memory
#pragma omp target teams distribute parallel for
for (int i = 0; i < size / sizeof(int); ++i)
    host_ptr[i] = i * 2; // Accessing host_ptr directly on dev.
```

→ `reverse_offload`: allows computations to be offloaded from a device back to the host

→ Possible use case: I/O



# Requires directive / 2

→ `dynamic_allocators`: indicates that the code region uses dynamic memory allocators within target regions

```
#pragma omp requires dynamic_allocators
#pragma omp target
{
    device_ptr = (int*)omp_target_alloc(size,
                                       omp_get_default_device());
}
```

→ `atomic_default_mem_order(memory-order)`: OpenMP atomics have different memory order options (e.g., relaxed, acquire, release, seq\_cst) and this clause sets a default for atomic directives within the scope.

→ `device_functions_implicit_declaration`: functions called within target regions might be implicitly treated as device functions even if they are not explicitly declared with `#pragma omp declare target`



# Requires directive / 3

- `self_map`: **implies** `unified_shared_memory` and indicates that the corresponding mapped storage is the same as the original storage
  - Guarantees that no copy is made
- Use case: Improved code portability
  - If you rely on a specific OpenMP feature, the `requires` directive acts as a flag.
- Use case: Feature discovery and documentation
  - `requires` directives serve as documentation within your code, clearly indicating which OpenMP features are being used, making it easier for other developers (or your future self) to understand the code's dependencies and requirements
- Use case: Early error detection
  - Compilers can potentially check for the presence of required features at compile time and if a required feature is missing in the target OpenMP implementation, the compiler can issue a diagnostic, preventing runtime surprises and making debugging easier
- Use case: Some implementations may eventually use `requires` directives as a hint for optimization



# Error directive / 1

- The primary purpose of the `error` directive is to provide a mechanism for error detection and handling within OpenMP parallel regions
  - When an `error` directive is encountered, it triggers a warning or the termination of the entire OpenMP program (consequence: use for critical errors only)
    - `severity` clause: `warning` or `fatal`
    - Condition: triggers only if the expression evaluates to true
  - Use case: Enforcing device constraints and/or ensuring correct device selection
  - Use case: Detecting unsupported features

```
if (omp_get_num_devices() == 0)
{
    #pragma omp error message("No GPU device available!")
    severity(fatal)
}
```

# Error directive / 2

## → Use case: Debugging

```
#pragma omp target data map(from: device_array[0:100])
if (!omp_target_is_present(device_array,
                           omp_get_default_device()))
{
    #pragma omp error message("Device memory allocation
                              failed!") severity(fatal)
}
```

# *Unified Shared Memory*

# Unified Shared Memory

- Memory on host and GPU share the same address space
  - Software emulation: via page migration
  - Hardware: via memory coherency protocol via fast memory link
  - Hardware: via shared physical memory

```
subroutine saxpy(n, a, x, y) {  
  integer :: n  
  real(kind=real32) :: a  
  real(kind=real32), dimension(:) :: x, y  
  !$omp target map(to:x) map(tofrom(y))  
  !$omp teams loop  
  do i = 1, n  
    y(i) = a * x(i) + y(i);  
  end do  
end subroutine saxpy
```

host

target

# Unified Shared Memory

- Memory on host and GPU share the same address space
  - Software emulation: via page migration
  - Hardware: via memory coherency protocol via fast memory link
  - Hardware: via shared physical memory

```
subroutine saxpy(n, a, x, y) {  
  integer :: n  
  real(kind=real32) :: a  
  real(kind=real32), dimension(:) :: x, y  
  !$omp target map(to:x) map(tofrom:y))  
  !$omp teams loop  
  do i = 1, n  
    y(i) = a * x(i) + y(i);  
  end do  
end subroutine saxpy
```

host

target

These map clauses should not be needed for a unified shared memory system.

# Unified Shared Memory

- Memory on host and GPU share the same address space
  - Software emulation: via page migration
  - Hardware: via memory coherency protocol via fast memory link
  - Hardware: via shared physical memory

```
subroutine saxpy(n, a, x, y) {  
  integer :: n  
  real(kind=real32) :: a  
  real(kind=real32), dimension(:) :: x, y  
  !$omp target map(to:x) map(tofrom:y)  
  !$omp teams loop  
  do i = 1, n  
    y(i) = a * x(i) + y(i);  
  end do  
end subroutine saxpy
```

host

target

These map clauses should not be needed for a unified shared memory system.

# Unified Shared Memory

- When removing explicit map clauses, the compiler is required to add them back for the target constructs in the code.
  - Implicit data mapping will still happen, unless the compiler can elide the map clauses (at runtime!)
  - Some compilers offer special compiler switches to statically elide map clauses.

```
subroutine saxpy(n, a, x, y) {  
  integer :: n  
  real(kind=real32) :: a  
  real(kind=real32), dimension(:) :: x, y  
  !$omp target "map(tofrom:x) map(tofrom(y))"  
  !$omp teams loop  
  do i = 1, n  
    y(i) = a * x(i) + y(i);  
  end do  
end subroutine saxpy
```

host

target



# Requiring Unified Shared Memory

- OpenMP programmers can remove the implicit map clauses when compiling for a USM target
  - Use “requires” directive to provide special code requirements

```
subroutine saxpy(n, a, x, y) {  
    !$omp requires unified_shared_memory  
    integer :: n  
    real(kind=real32) :: a  
    real(kind=real32), dimension(:) :: x, y  
    !$omp target  
    !$omp teams loop  
    do i = 1, n  
        y(i) = a * x(i) + y(i);  
    end do  
end subroutine saxpy
```

host

target

No more map clauses!

# *Memory Management*

# Memory Management

- Allocator := an OpenMP object that fulfills requests to allocate and deallocate storage for program variables
- OpenMP allocators are of type `omp_allocator_handle_t`
- Default allocator for Host
  - via `OMP_ALLOCATOR` env. var. or corresponding API
- OpenMP 5.0 supports a set of memory allocators
  - With constant improvements to memory management in every update

- Pre-defined allocators enable the selection of a certain kind of memory

Allocator name	Storage selection intent
omp_default_mem_alloc	use default storage
omp_large_cap_mem_alloc	use storage with large capacity
omp_const_mem_alloc	use storage optimized for read-only variables
omp_high_bw_mem_alloc	use storage with high bandwidth
omp_low_lat_mem_alloc	use storage with low latency
omp_cgroup_mem_alloc	use storage close to all threads in the contention group of the thread requesting the allocation
omp_pteam_mem_alloc	use storage that is close to all threads in the same parallel region of the thread requesting the allocation
omp_thread_mem_alloc	use storage that is close to the thread requesting the allocation

# OpenMP Allocators / 2

## ■ Construction of allocators with traits via

```
→ omp_allocator_handle_t omp_init_allocator(  
    omp_memspace_handle_t memspace,  
    int ntraits, const omp_alloctrail_t traits[]);
```

→ Selection of memory space mandatory

→ Empty traits set: use defaults

## ■ Allocators have to be destroyed with `*_destroy_*`

## ■ Custom allocator can be made default with

```
omp_set_default_allocator(omp_allocator_handle_t allocator)
```

# OpenMP Allocator Traits / 1

sync_hint	contended, uncontended, serialized, private	default: contended
alignment	positive integer value that is a power of two	default: 1 byte
access	all, memspace, device, cgroup, pteam, thread	default: memspace
pool_size	positive integer value	
fallback	default_mem_fb, null_fb, abort_fb, allocator_fb	default: default_mem_fb
fb_data	an allocator handle	
pinned	true, false	default: false
partition	environment, nearest, blocked, interleaved	default: environment
pin_device	conforming device number	
preferred_device	conforming device number	
target access	single, multiple	default: single
atomic scope	all, device	default: device
part_size	positive integer value	
partitioner	a memory partitioner handle	
partitioner_arg	an integer value	0



# OpenMP Allocator Traits / 2

- `fallback`: describes the behavior if the allocation cannot be fulfilled
  - `default_mem_fb`: return system's default memory
  - Other options: null, abort, or use different allocator
- `pinned`: request pinned memory, i.e. for GPUs,
  - device may be specified



# OpenMP Allocator Traits / 3

- `partition`: partitioning of allocated memory of physical storage resources (think of NUMA)
  - `environment`: use system's default behavior
  - `nearest`: most closest memory
  - `blocked`: partitioning into approx. same size with at most one block per storage resource
  - `interleaved`: partitioning in a round-robin fashion across the storage resources, in which `part_size` specifies the size of individual partitions
  - `partitioner`: definition of memory parts and distribution across storage are defined by a memory partitioner



# OpenMP Allocator Traits / 4

## ■ Example code:

```
const omp_alloctrail_t traits[] ={{omp_atk_partition,  
                                omp_atv_interleaved},  
                                {omp_atk_part_size, 1024*1024} };  
  
omp_allocator_handle_t numa_dev_alloc =  
    omp_init_allocator(omp_default_mem_space, 2, traits);  
int * a = omp_alloc(numa_dev_alloc, 6*1024*1024);
```

→ Distributes chunks of memory:



# OpenMP Allocator Traits / 5

sync_hint	contended, uncontended, serialized, private	default: contended
alignment	positive integer value that is a power of two	default: 1 byte
access	all, memspace, device, cgroup, pteam, thread	default: memspace
pool_size	positive integer value	
fallback	default_mem_fb, null_fb, abort_fb, allocator_fb	default: default_mem_fb
fb_data	an allocator handle	
pinned	true, false	default: false
partition	environment, nearest, blocked, interleaved	default: environment
pin_device	conforming device number	
preferred_device	conforming device number	
target access	single, multiple	default: single
atomic_scope	all, device	default: device
part_size	positive integer value	
partitioner	a memory partitioner handle	
partitioner_arg	an integer value	0



# OpenMP Allocator Traits / 6

- `partition`: partitioning of allocated memory of physical storage resources (think of NUMA)
  - `environment`: use system's default behavior
  - `nearest`: most closest memory
  - `blocked`: partitioning into approx. same size with at most one block per storage resource
  - `interleaved`: partitioning in a round-robin fashion across the storage resources, in which `part_size` specifies the size of individual partitions
  - `partitioner`: definition of memory parts and distribution across storage are defined by a memory partitioner

# OpenMP Memory Partitioner

- Memory Partitioner := an OpenMP object that represents mechanisms to create and destroy memory partitions
  - Memory Partition := a definition how an allocator divides memory into parts
    - Memory Part := a storage block in a single storage resource within a memory space
- `omp_init_mempartitioner` routine: initializes a partitioner that ...
  - ... can be used with an OpenMP allocator
  - ... takes the argument `compute_proc` to determine the number of memory parts and their distribution across the storage resources
  - + further management and cleanup routines
- Memory Space Retrieving Routines: return memory space handles

# *Interoperability between OpenMP and HIP/CUDA*

# Example: Calling saxpy (in HIP)

```
void example() {
    float a = 2.0;
    float * x;
    float * y;

    // allocate the device memory
    #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])
    {
        compute_1(n, x);
        compute_2(n, y);
        saxpy(n, a, x, y)
        compute_3(n, y);
    }
}
```



# Example: Calling saxpy (in HIP)

```
void example() {  
    float a = 2.0;  
    float * x;  
    float * y;  
  
    // allocate the device memory  
    #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])  
    {  
        compute_1(n, x);  
        compute_2(n, y);  
        saxpy(n, a, x, y)  
        compute_3(n, y);  
    }  
}
```

```
void saxpy(int n, float a,  
           float * x, float * y) {  
    #pragma omp target teams distribute \  
        parallel for simd  
    for (size_t i = 0; i < n; ++i) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

# Example: Calling saxpy (in HIP)

```
void example() {  
    float a = 2.0;  
    float * x;  
    float * y;  
  
    // allocate the device memory  
    #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])  
    {  
        compute_1(n, x);  
        compute_2(n, y);  
        saxpy(n, a, x, y)  
        compute_3(n, y);  
    }  
}
```

Let's assume that we want to implement the saxpy() function in a low-level language.

```
void saxpy(int n, float a,  
           float * x, float * y) {  
    #pragma omp target teams distribute \  
        parallel for simd  
    for (size_t i = 0; i < n; ++i) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

# HIP Kernel for saxpy()

- Assume a HIP version of the SAXPY kernel:

```
__global__ void saxpy_kernel(int n, float a, float * x, float * y) {
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;
    y[i] = a * x[i] + y[i];
}

void saxpy_hip(int n, float a, float * x, float * y) {
    assert(n % 256 == 0);
    saxpy_kernel<<<n/256,256,0,NULL>>>(n, a, x, y);
}
```

- We need a way to translate the host pointer that was mapped by OpenMP directives and retrieve the associated device pointer.



# HIP Kernel for saxpy()

- Assume a HIP version of the SAXPY kernel:

```
__global__ void saxpy_kernel(int n, float a, float * x, float * y) {  
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;  
    y[i] = a * x[i] + y[i];  
}
```

```
void saxpy_hip(int n, float a, float * x, float * y) {  
    assert(n % 256 == 0);  
    saxpy_kernel<<<n/256,256,0,NULL>>>(n, a, x, y);  
}
```

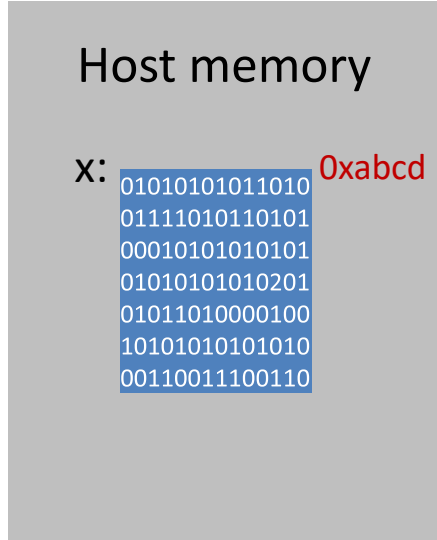


These are device pointers!

- We need a way to translate the host pointer that was mapped by OpenMP directives and retrieve the associated device pointer.

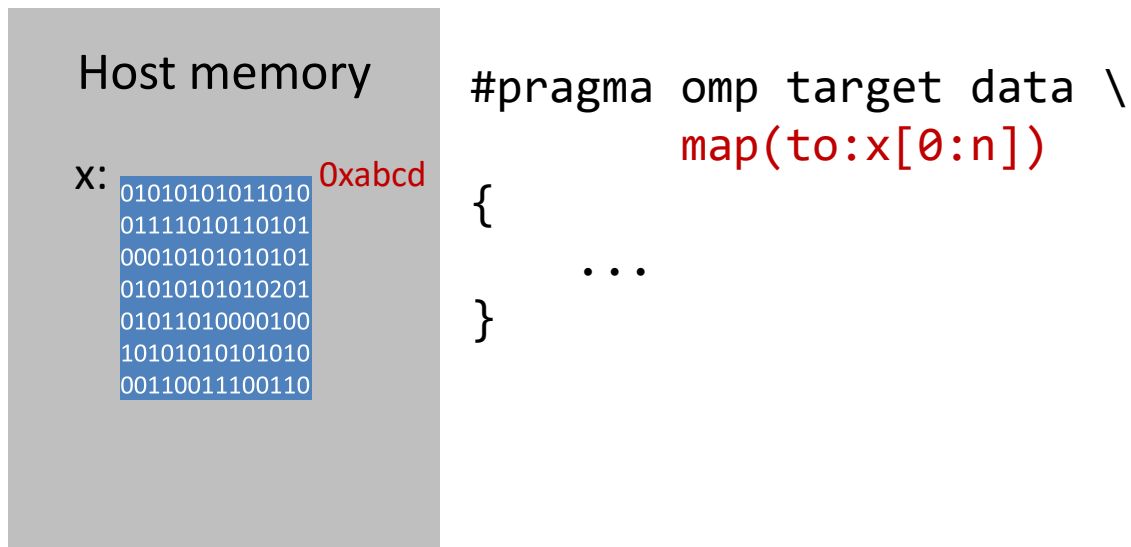
# Pointer Translation /1

- When creating the device data environment, OpenMP creates a mapping between
  - the (virtual) memory pointer on the host and
  - the (virtual) memory pointer on the target device.
- This mapping is established through the data-mapping directives and their clauses.



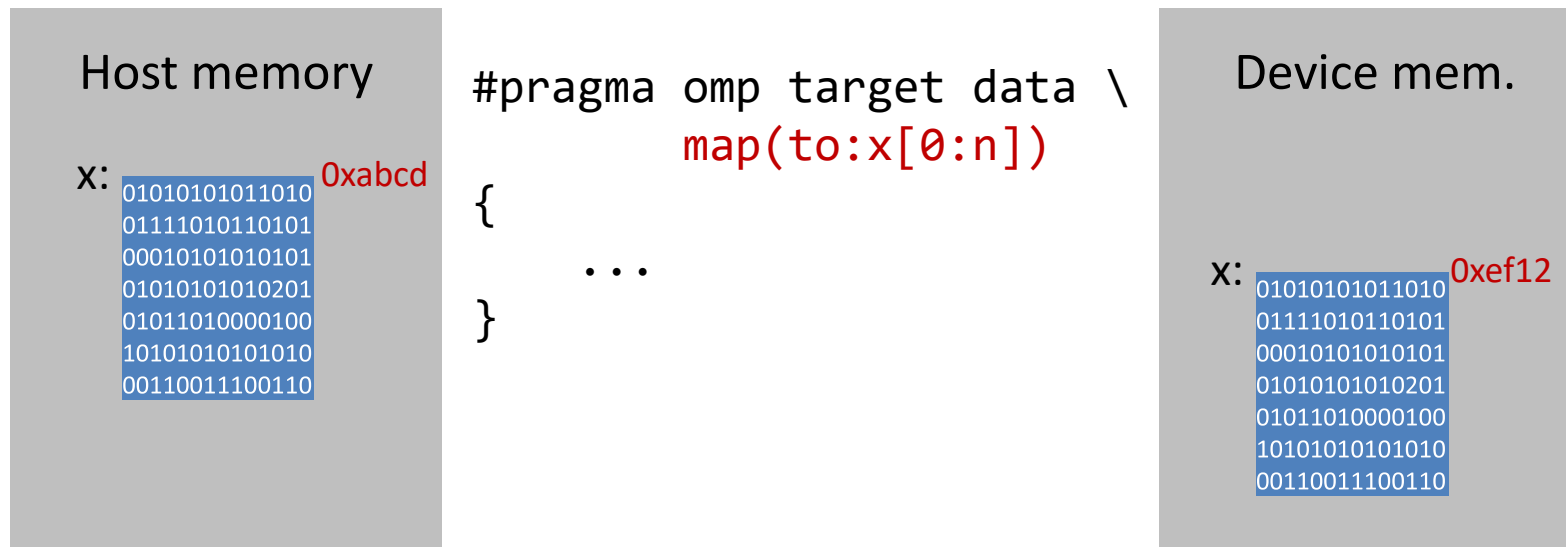
# Pointer Translation /1

- When creating the device data environment, OpenMP creates a mapping between
  - the (virtual) memory pointer on the host and
  - the (virtual) memory pointer on the target device.
- This mapping is established through the data-mapping directives and their clauses.



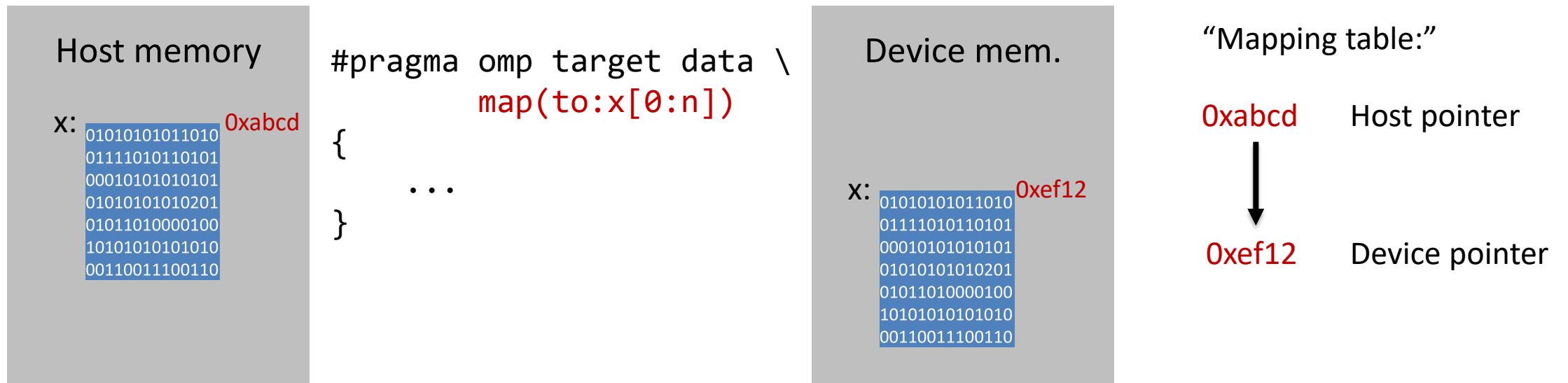
# Pointer Translation /1

- When creating the device data environment, OpenMP creates a mapping between
  - the (virtual) memory pointer on the host and
  - the (virtual) memory pointer on the target device.
- This mapping is established through the data-mapping directives and their clauses.



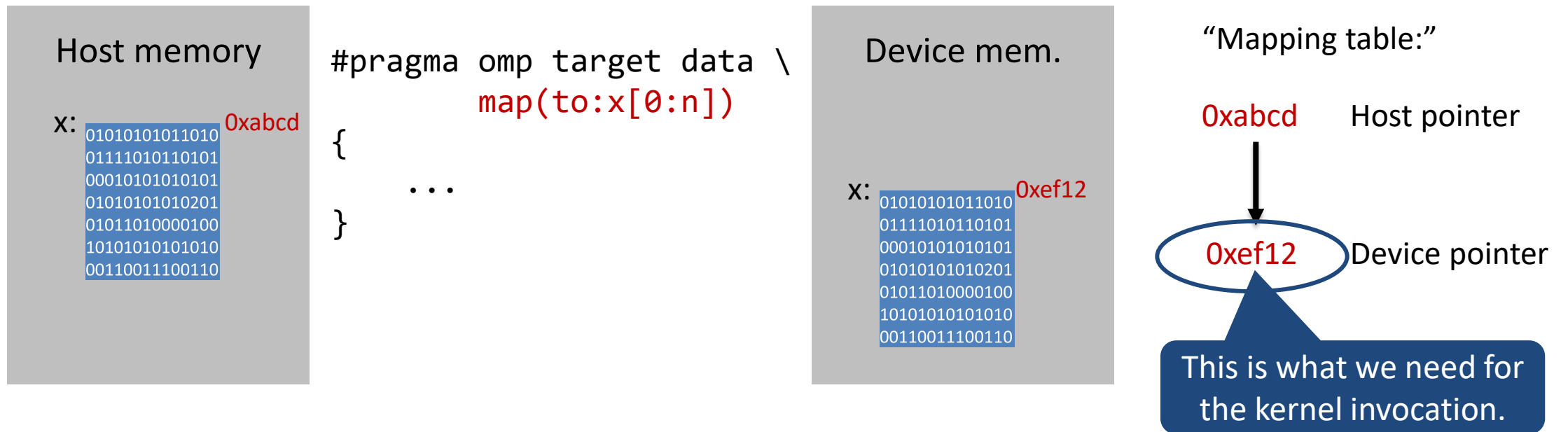
# Pointer Translation /1

- When creating the device data environment, OpenMP creates a mapping between
  - the (virtual) memory pointer on the host and
  - the (virtual) memory pointer on the target device.
- This mapping is established through the data-mapping directives and their clauses.



# Pointer Translation /1

- When creating the device data environment, OpenMP creates a mapping between
  - the (virtual) memory pointer on the host and
  - the (virtual) memory pointer on the target device.
- This mapping is established through the data-mapping directives and their clauses.



## Example: Calling saxpy (in HIP)

- The target data construct defines the `use_device_addr` clause to perform pointer translation.
  - The OpenMP implementation searches for the host pointer in its internal mapping tables.
  - The associated device pointer is then returned.

```
type * x = 0xabcd;
#pragma omp target data use_device_addr(x[:0])
{
    example_func(x);    // x == 0xef12
}
```

- Note: the original pointer variable is hidden within the target data construct for the translation; only device pointer variable is visible.



# Putting it Together...

```
void example() {
    float a = 2.0;
    float * x = ...;    // assume: x = 0xabcd
    float * y = ...;

    // allocate the device memory
    #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])
    {
        compute_1(n, x); // mapping table: x:[0xabcd,0xef12], x = 0xabcd
        compute_2(n, y);
        #pragma omp target data use_device_addr(x[:0],y[:0])
        {
            saxpy_hip(n, a, x, y) // mapping table: x:[0xabcd,0xef12], x = 0xef12
        }
        compute_3(n, y);
    }
}
```

# *Advanced Task/GPU Synchronization*

# Asynchronous API Interaction

- Some APIs are based on asynchronous operations
  - MPI asynchronous send and receive
  - Asynchronous I/O
  - HIP, CUDA and OpenCL stream-based offloading
  - In general: any other API/model that executes asynchronously with OpenMP (tasks)



# Asynchronous API Interaction

- Some APIs are based on asynchronous operations
  - MPI asynchronous send and receive
  - Asynchronous I/O
  - HIP, CUDA and OpenCL stream-based offloading
  - In general: any other API/model that executes asynchronously with OpenMP (tasks)
- Example: HIP memory transfers

```
do_something();  
hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);  
do_something_else();  
hipStreamSynchronize(stream);  
do_other_important_stuff(dst);
```

# Asynchronous API Interaction

- Some APIs are based on asynchronous operations
  - MPI asynchronous send and receive
  - Asynchronous I/O
  - HIP, CUDA and OpenCL stream-based offloading
  - In general: any other API/model that executes asynchronously with OpenMP (tasks)
- Example: HIP memory transfers

```
do_something();  
hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);  
do_something_else();  
hipStreamSynchronize(stream);  
do_other_important_stuff(dst);
```

- Programmers need a mechanism to marry asynchronous APIs with the parallel task model of OpenMP
  - How to synchronize completions events with task execution?



# Try 1: Use just OpenMP Tasks

```
void hip_example() {  
#pragma omp task    // task A  
    {  
        do_something();  
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);  
    }  
#pragma omp task // task B  
    {  
        do_something_else();  
    }  
#pragma omp task // task C  
    {  
        hipStreamSynchronize(stream);  
        do_other_important_stuff(dst);  
    }  
}
```

■ This solution does not work!



# Try 1: Use just OpenMP Tasks

```
void hip_example() {  
#pragma omp task      // task A  
  {  
    do_something();  
    hipMemcpyAsync(dst, src, bytes, hipMemcpyDeviceToHost, stream);  
  }  
#pragma omp task // task B  
  {  
    do_something_else();  
  }  
#pragma omp task // task C  
  {  
    hipStreamSynchronize(stream);  
    do_other_important_stuff(dst);  
  }  
}
```

Race condition between the tasks A & C,  
task C may start execution before  
task A enqueues memory transfer.

■ This solution does not work!

# Try 2: Use just OpenMP Tasks Dependences

```
void hip_example() {
#pragma omp task depend(out:stream)    // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
#pragma omp task                        // task B
    {
        do_something_else();
    }
#pragma omp task depend(in:stream)    // task C
    {
        hipStreamSynchronize(stream);
        do_other_important_stuff(dst);
    }
}
```

- This solution should work, but takes a thread away from execution while the system is handling the data transfer.



# Try 2: Use just OpenMP Tasks Dependences

```
void hip_example() {  
#pragma omp task depend(out:stream) // task A  
  {  
    do_something();  
    hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);  
  }  
#pragma omp task // task B  
  {  
    do_something_else();  
  }  
#pragma omp task depend(in:stream) // task C  
  {  
    hipStreamSynchronize(stream);  
    do_other_important_stuff(dst);  
  }  
}
```

Synchronize execution of tasks through dependence.  
May work, but task C will be blocked waiting for  
the data transfer to finish

- This solution should work, but takes a thread away from execution while the system is handling the data transfer.

# OpenMP Detachable Tasks

- OpenMP 5.0 introduces the concept of a detachable task
  - Task can detach from executing thread without being “completed”
  - Regular task synchronization mechanisms can be applied to await completion of a detached task
  - Runtime API to complete a task

# OpenMP Detachable Tasks

- OpenMP 5.0 introduces the concept of a detachable task
  - Task can detach from executing thread without being “completed”
  - Regular task synchronization mechanisms can be applied to await completion of a detached task
  - Runtime API to complete a task
- Detached task events: `omp_event_handle_t` datatype

# OpenMP Detachable Tasks

- OpenMP 5.0 introduces the concept of a detachable task
  - Task can detach from executing thread without being “completed”
  - Regular task synchronization mechanisms can be applied to await completion of a detached task
  - Runtime API to complete a task
- Detached task events: `omp_event_handle_t` datatype
- Detached task clause: `detach(event)`

# OpenMP Detachable Tasks

- OpenMP 5.0 introduces the concept of a detachable task
  - Task can detach from executing thread without being “completed”
  - Regular task synchronization mechanisms can be applied to await completion of a detached task
  - Runtime API to complete a task
- Detached task events: `omp_event_handle_t` datatype
- Detached task clause: `detach(event)`
- Runtime API: `void omp_fulfill_event(omp_event_handle_t event)`

# Detaching Tasks

```
omp_event_handle_t event;  
void detach_example() {  
#pragma omp task detach(event)  
    {  
        important_code();  
    }  
  
#pragma omp taskwait  
}
```

# Detaching Tasks

```
omp_event_handle_t event;  
void detach_example() {  
#pragma omp task detach(event)  
    {  
        important_code();  
    } ①  
#pragma omp taskwait  
}
```

## 1. Task detaches

# Detaching Tasks

```
omp_event_handle_t event;  
void detach_example() {  
#pragma omp task detach(event)  
    {  
        important_code();  
    } ①  
#pragma omp taskwait ②  
}
```

1. Task detaches
2. taskwait construct cannot complete

# Detaching Tasks

```
omp_event_handle_t event;  
void detach_example() {  
#pragma omp task detach(event)  
    {  
        important_code();  
    } ①  
#pragma omp taskwait ②  
}
```

Some other thread/task:

```
omp_fulfill_event(event); ③
```

1. Task detaches
2. taskwait construct cannot complete
3. Signal event for completion

# Detaching Tasks

```
omp_event_handle_t event;  
void detach_example() {  
#pragma omp task detach(event)  
  {  
    important_code();  
  } ①  
#pragma omp taskwait ② ④  
}
```

Some other thread/task:

```
omp_fulfill_event(event); ③
```

1. Task detaches
2. taskwait construct cannot complete
3. Signal event for completion
4. Task completes and taskwait can continue

# Putting It All Together

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
    omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
    }
#pragma omp task // task B
    do_something_else();

#pragma omp taskwait
#pragma omp task // task C
    {
        do_other_important_stuff(dst);
    } }
```



# Putting It All Together

```

void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
    omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
    }
#pragma omp task // task B
    do_something_else();

#pragma omp taskwait
#pragma omp task // task C
    {
        do_other_important_stuff(dst);
    }
}

```

1. Task A detaches

# Putting It All Together

```

void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
    omp_ufill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
    }
#pragma omp task // task B
    do_something_else();

#pragma omp taskwait ②
#pragma omp task // task C
    {
        do_other_important_stuff(dst);
    }
}

```

1. Task A detaches
2. taskwait does not continue



# Putting It All Together

```

void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
    ③ omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
    }
#pragma omp task // task B
    do_something_else();

#pragma omp taskwait ②
#pragma omp task // task C
    {
        do_other_important_stuff(dst);
    }
}

```

1. Task A detaches
2. taskwait does not continue
3. When memory transfer completes, callback is invoked to signal the event for task completion

# Putting It All Together

```

void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
    ③ omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
    }
#pragma omp task // task B
    do_something_else();

#pragma omp taskwait ② ④
#pragma omp task // task C
    {
        do_other_important_stuff(dst);
    }
}

```

1. Task A detaches
2. taskwait does not continue
3. When memory transfer completes, callback is invoked to signal the event for task completion
4. taskwait continues, task C executes

# Removing the `taskwait` Construct

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
    omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}

void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task depend(out:dst) detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
    }
#pragma omp task // task B
    do_something_else();

#pragma omp task depend(in:dst) // task C
    {
        do_other_important_stuff(dst);
    }
}
```



# Removing the taskwait Construct

```

void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
    omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task depend(out:dst) detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        ① hipStreamAddCallback(stream, callback, &hip_event, 0);
    }
#pragma omp task // task B
    do_something_else();

#pragma omp task depend(in:dst) // task C
    {
        do_other_important_stuff(dst);
    } }

```

1. Task A detaches and task C will not execute because of its unfulfilled dependency on A



# Removing the taskwait Construct

```

void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
    ② omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task depend(out:dst) detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        ① hipStreamAddCallback(stream, callback, &hip_event, 0);
    }
#pragma omp task // task B
    do_something_else();

#pragma omp task depend(in:dst) // task C
    {
        do_other_important_stuff(dst);
    }
}

```

1. Task A detaches and task C will not execute because of its unfulfilled dependency on A
2. When memory transfer completes, callback is invoked to signal the event for task completion


# Removing the taskwait Construct

```

void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
    ② omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task depend(out:dst) detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        ① hipStreamAddCallback(stream, callback, &hip_event, 0);
    }
#pragma omp task // task B
    do_something_else();

#pragma omp task depend(in:dst) // task C
    {
        do_other_important_stuff(dst);
    }
}

```



1. Task A detaches and task C will not execute because of its unfulfilled dependency on A
2. When memory transfer completes, callback is invoked to signal the event for task completion
3. Task A completes and C's dependency is fulfilled

# Q&A





# Performance Optimisation and Productivity 3

A Centre of Excellence in HPC

## Contact:



<https://www.pop-coe.eu>



[pop@bsc.es](mailto:pop@bsc.es)



[@POP\\_HPC](#)



[youtube.com/POPHPC](https://www.youtube.com/POPHPC)



This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101143931. The JU receives support from the European Union's Horizon Europe research and innovation programme and Spain, Germany, France, Portugal and the Czech Republic.

