
Advanced topics
archer2
epcc

Reusing this material

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

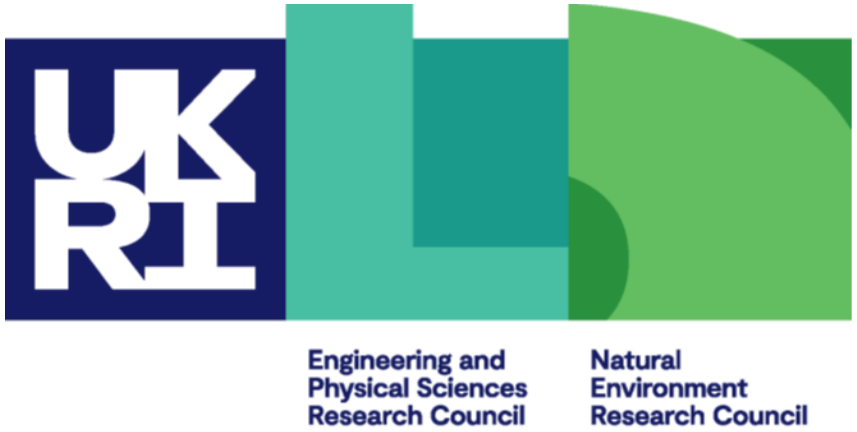
https://creativecommons.org/licenses/by-nc-sa/4.0/

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material, you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Partners

epcc



epcc

THE UNIVERSITY of EDINBURGH

Hewlett Packard Enterprise

Outline

- OpenMP/ROCm/CUDA interoperability
- Custom data mapper
- Memory types
- Memory allocation
- Asynchronous tasks

OpenMP/ROCm compatibility

Using ROCm in an OpenMP Program

- Sometimes need to call routines from a library that expects a device pointer, not a mapped variable.
- Might be needed if you want to port a computationally heavy kernel to CUDA/HIP to increase performance
- Might be needed if you want to call vendor libraries (rocFFT/cuFFT, cuBLAS,rocBLAS …)

Using ROCm in an OpenMP Program

- Suppose `var_a` is a variable already mapped to the device with OpenMP
- Want to avoid re-allocating and transferring data with CUDA/HIP
- Need to obtain the address of the variable mapped on the device

Using rocm in an OpenMP Program

- The clause `use_device_addr` makes all references to `var_a` in the code refer to the variable mapped on the GPU.

```C/C++
#pragma omp target data use_device_addr(var_a)
{ . . . . }
```

```Fortran
!$omp target data use_device_addr(var_a)
  . . . .
!$omp end target data
```

Using ROCm with OpenMP

- Example of a matrix multiplication using rocBLAS

```C/C++
#pragma omp target data map(tofrom:A[0:M*K],B[0:K*N],C[0:M*N]){
    #pragma omp target data use_device_addr(A[0:M*K],B[0:K*N],C[0:M*N]){
        rocblas_dgemm( . . . , A, . . . , B, . . . , C, . . .);
  }
}
```

```Fortran
!$omp target data map(to:A(1:M,1:K),B(1:K,1:N)) map(tofrom:C(1:M,1:N))
  !$omp target data use_device_addr(A(1:M,1:K), B(1:K,1:N), C(1:M,1:N))
      call rocblas_dgemm( . . . , A, . . . , B, . . . , C, . . .)
  !$omp end target data
!$omp end target data
```

Using OpenMP in an HIP/CUDA program

epcc

- Memory allocations on HIP/CUDA
- Want to use OpenMP code that uses memory allocated on the device with HIP/CUDA
- Useful when porting from a HIP/CUDA program to a performance portable implementation with OpenMP

Using OpenMP in an HIP/CUDA program

- The clause `is_device_ptr(a,b , . . . )` tells OpenMP `a, b,` … are pointers to data available on the device, not on the host.
- Prevents mapping of the pointer variable, allows using the pointer directly on the device.

Using OpenMP in an HIP/CUDA program

Ex. : Copying an array b to array c

```
double * b_device, c_device;
hipMalloc( &b_device, n*sizeof(double) ) );
hipMalloc( &c_device, n*sizeof(double) ) );

#pragma omp target teams distribute parallel for
is_device_ptr(c_device, b_device)
for (int i=0;i<n;i++) {
  c_device[i] = b_device[i];
}
```

C/C++

Using OpenMP in an HIP/CUDA program

epcc

Ex. : Copying an array b to array c

```fortran
integer, parameter :: fp_kind = kind(0.0)
type(c_ptr) :: cptr_b, cptr_c
real(fp_kind), pointer, dimension(:) :: fptr_b => null(), fptr_c => null()
integer :: n = 5, i, ierr

ierr = hipMalloc(cptr_b, n * sizeof(fp_kind))
ierr = hipMalloc(cptr_c, n * sizeof(fp_kind))
call c_f_pointer(cptr_b, fptr_b, [n])
call c_f_pointer(cptr_c, fptr_c, [n])

!$omp target teams distribute parallel do simd is_device_ptr(fptr_b, fptr_c)
do i = 1, n
  fptr_c(i) = fptr_b(i)
end do
```

Fortran

Custom data mapper

Custom data mappers

- Consider a structure with three arrays: **`c`**, **`d`**, **`e`**.
- Each array is of length **`n`**.
- Mapping a variable of type **`A`** will correctly map scalar variables (like **`n`**) but not the value of the arrays **`c`**, **`d`**, **`e`**.

```c
struct my_struct {
  int n;
  double * c;
  double * d;
  double * e;
};
```
C/C++

```fortran
type my_type
  integer :: n
  real, allocatable :: c(:)
  real, allocatable :: d(:)
  real, allocatable :: e(:)
end type
```
Fortran

Custom data mappers

- All arrays need to be explicitly mapped every time a variable of type **`my_struct`** is used.
- Can be very verbose and error-prone with complex data structures.

```c
my_struct a, b;
#pragma omp target data map(tofrom:a,a.c[0:n],a.d[0:n],a.e[0:n]){
    // Do something with a
}
#pragma omp target data map(tofrom:b,b.c[0:n],b.d[0:n],b.e[0:n]) {
    // Do something with b
}
```

C/C++

```fortran
type(my_type) :: a, b
!$omp target data map(tofrom:a,a%c(1:a%n),a%d(1:a%n),a%e(1:a%n))
    ! Do something with a
!$omp end target data
!$omp target data map(tofrom:b,b%c(1:b%n),b%d(1:b%n),b%e(1:b%n))
    ! Do something with b
!$omp end target data
```

Fortran

Custom data mappers

- Declare all mapping rules for a custom structure only once.

```
#pragma omp declare mapper(my_struct x ) map(x, … )
```
C/C++

```
!$omp declare mapper(my_type :: x) map(x, …)
```
Fortran

- Here **x** is a dummy argument of type **my_struct** or **my_type**.

Custom data mappers

```c
#pragma omp declare mapper(my_struct x) \
  map(x, x.c[0:x.n], x.d[0:x.n], x.e[0:x.n])

my_struct a, b;

#pragma omp target data map(tofrom:a) {
        // Do something with a
}

#pragma omp target data map(tofrom:b) {
        // Do something with b
}

...
```

C/C++

```fortran
type(my_type) :: a, b

!$omp declare mapper(my_type :: x) &
!$omp map(x, x%c(1:x%n), x%d(1:x%n), &
!$omp x.e(1:x%n))

!$omp target data map(tofrom:a)
  ! Do something with a
!$omp end target data

!$omp target data map(tofrom:b)
  ! Do something with b
!$omp end target data

...
```

Fortran

Memory types

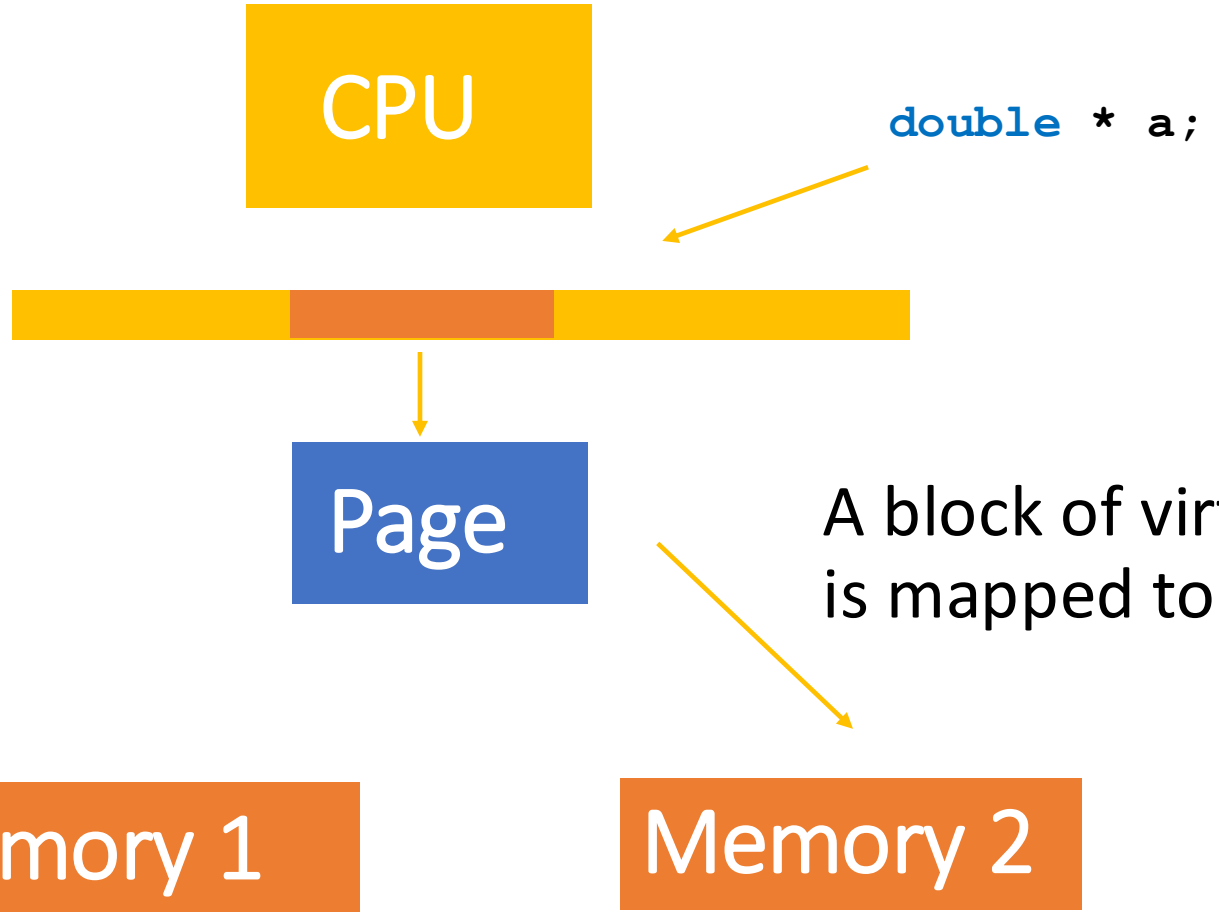
Host memory

- Pageable memory: Memory allocated in the usual way on the (**`malloc`**, **`new`**, **`allocate`**, etc...). Memory is only available on the host.

- Pinned memory: Memory optimized for transfer to the GPU. Custom allocators are used on the host

- Unified memory: Memory is accessible on both the host and the device. Memory allocated in the usual way.

Pageable memory



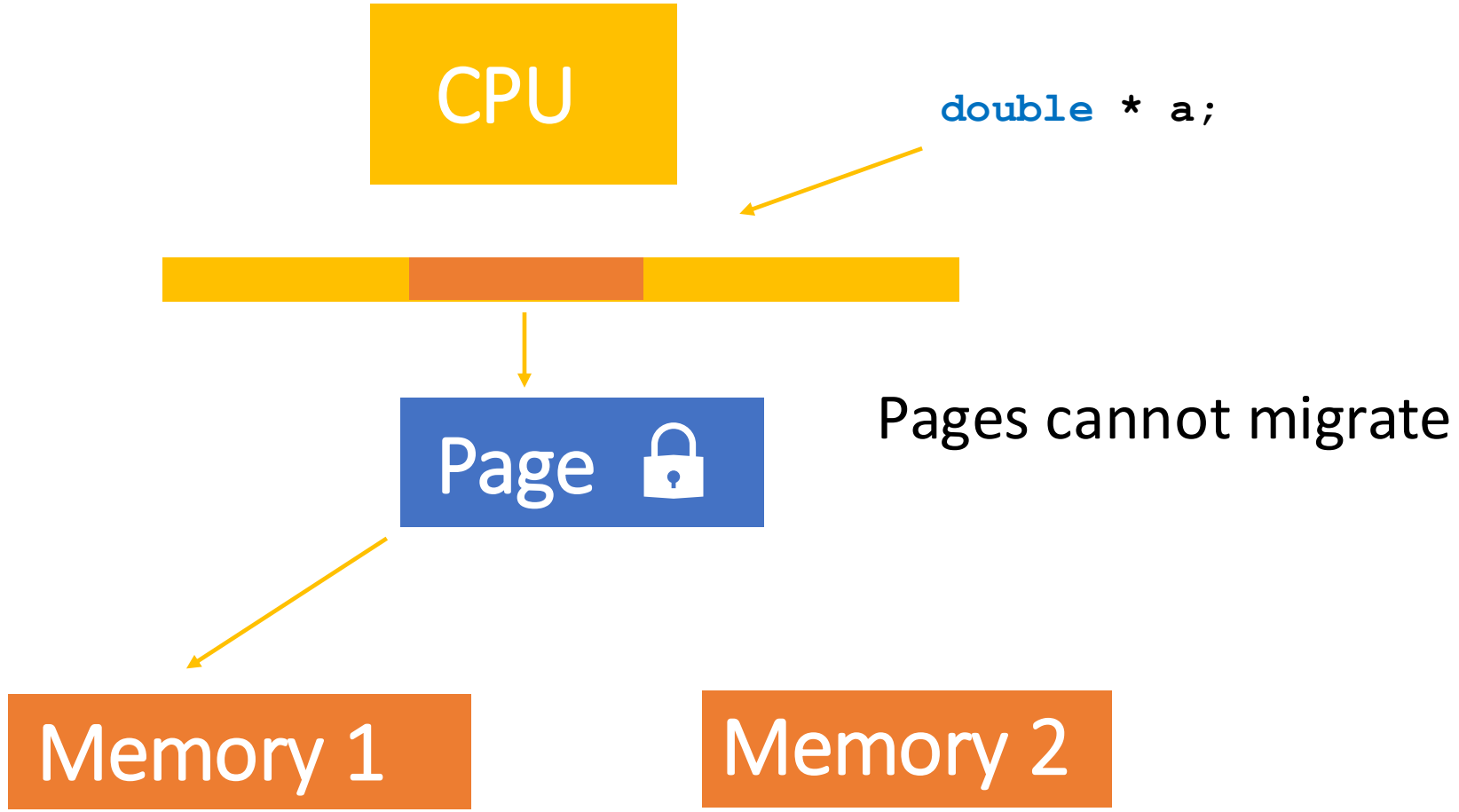
Pointer points to a location in virtual memory.

A block of virtual memory (called a page) is mapped to physical memory.

Pageable memory

- Memory is allocated using the usual allocators.
- Memory address is virtual.
- Virtual memory is mapped to physical memory using a page.
- Pages can migrate between different physical devices.
- Memory is only accessible from the host.

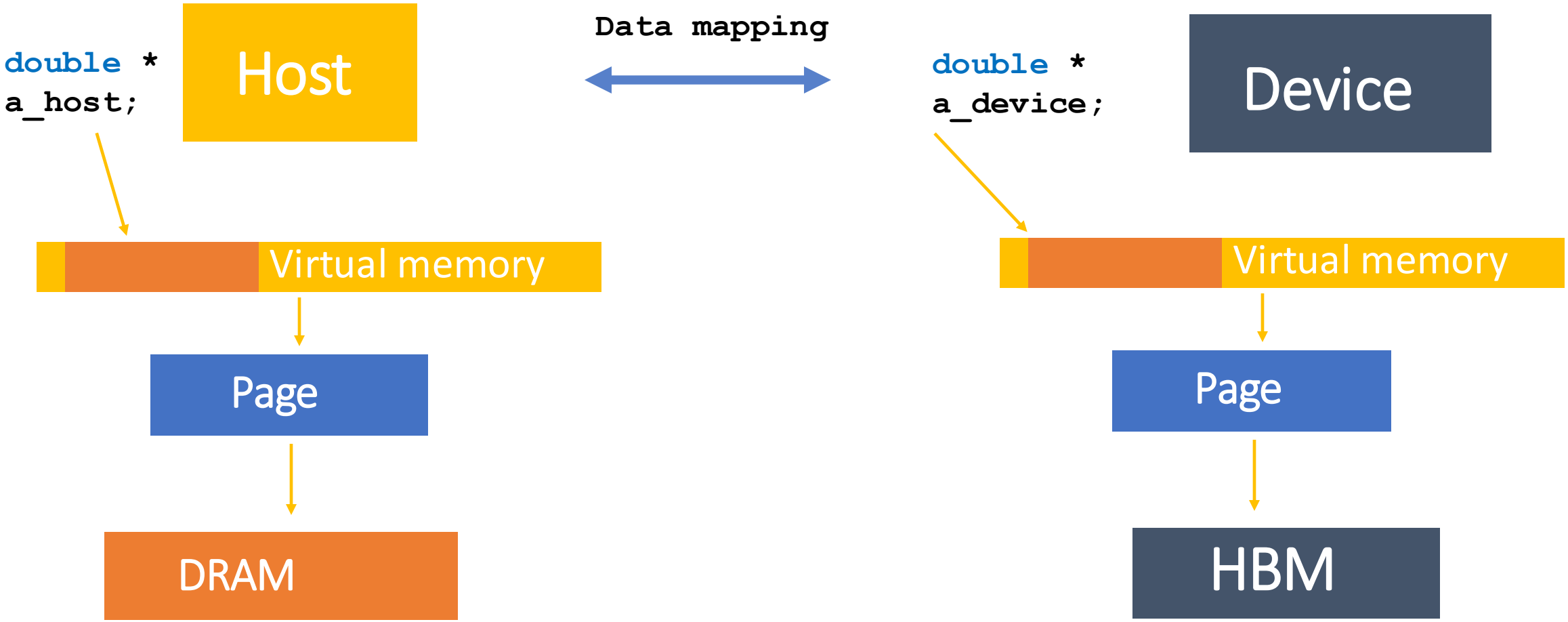
Pinned memory



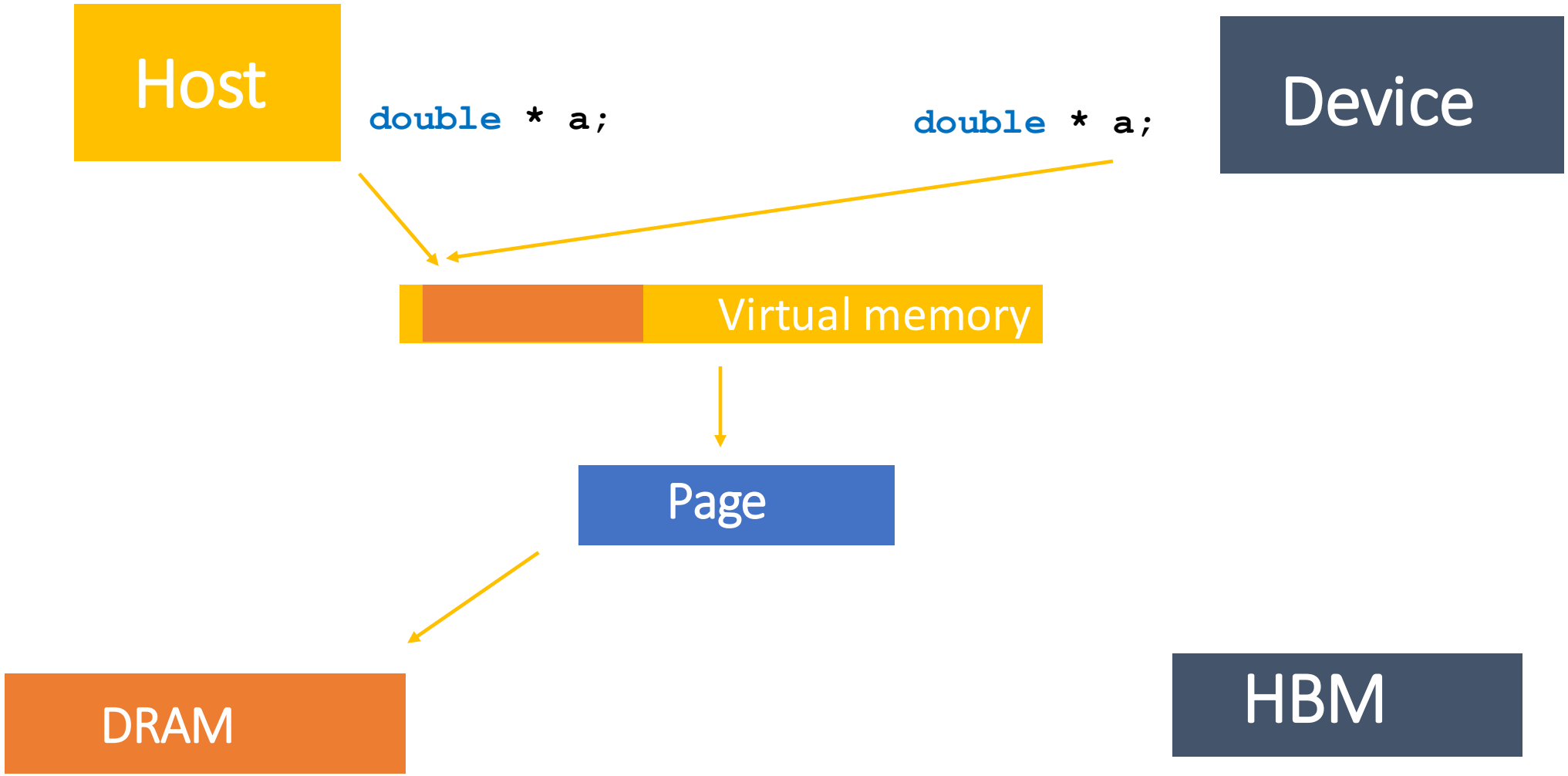
Pinned memory

- Memory is allocated using the usual allocators.
- Memory address is virtual.
- Virtual memory is mapped to physical memory using a page.
- Pages cannot migrate between different physical devices.
- Memory is only accessible from the host.
- Memory transfers are faster than pageable memory.

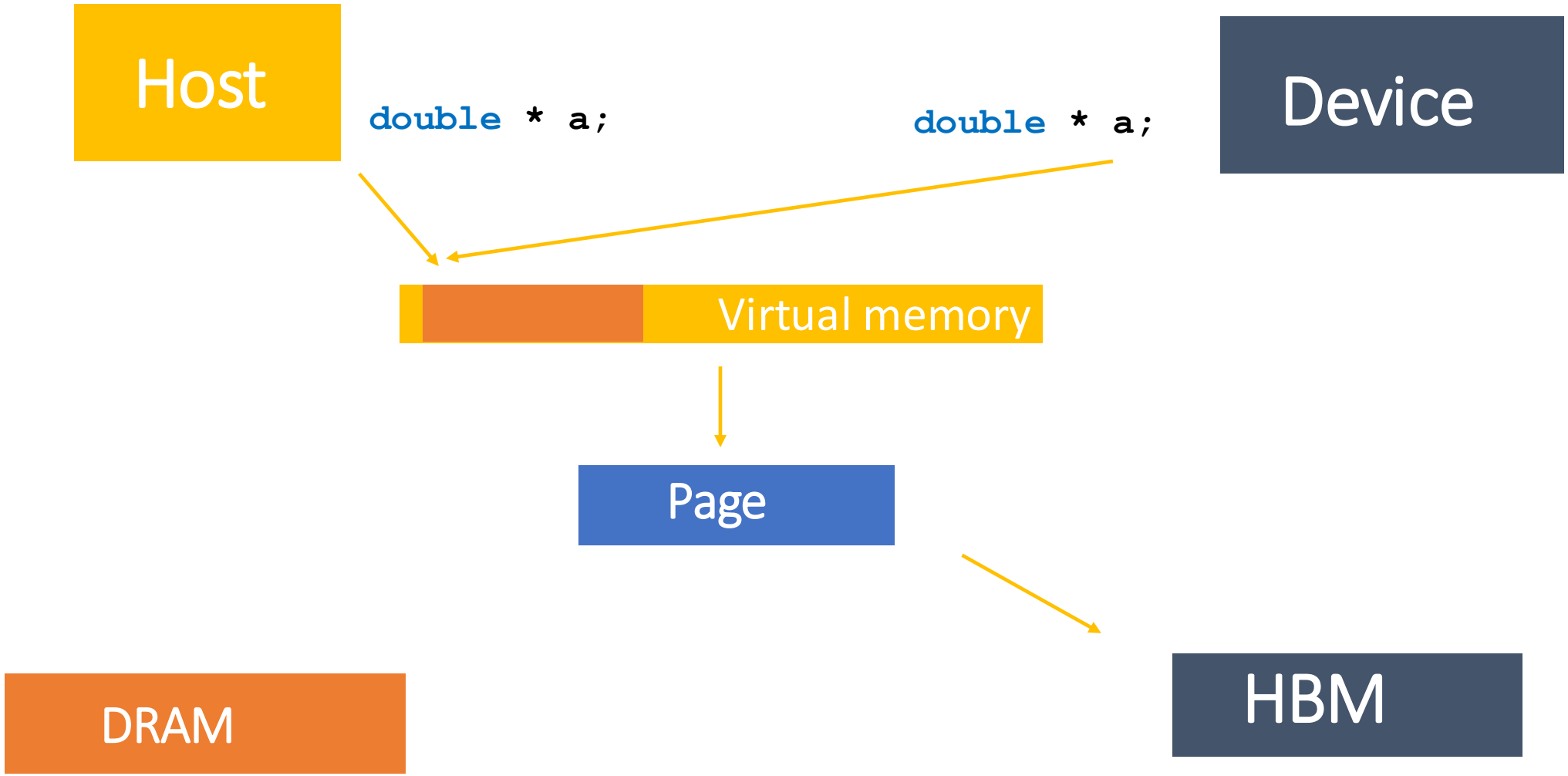
Non-unified memory



Unified memory



Non-unified memory



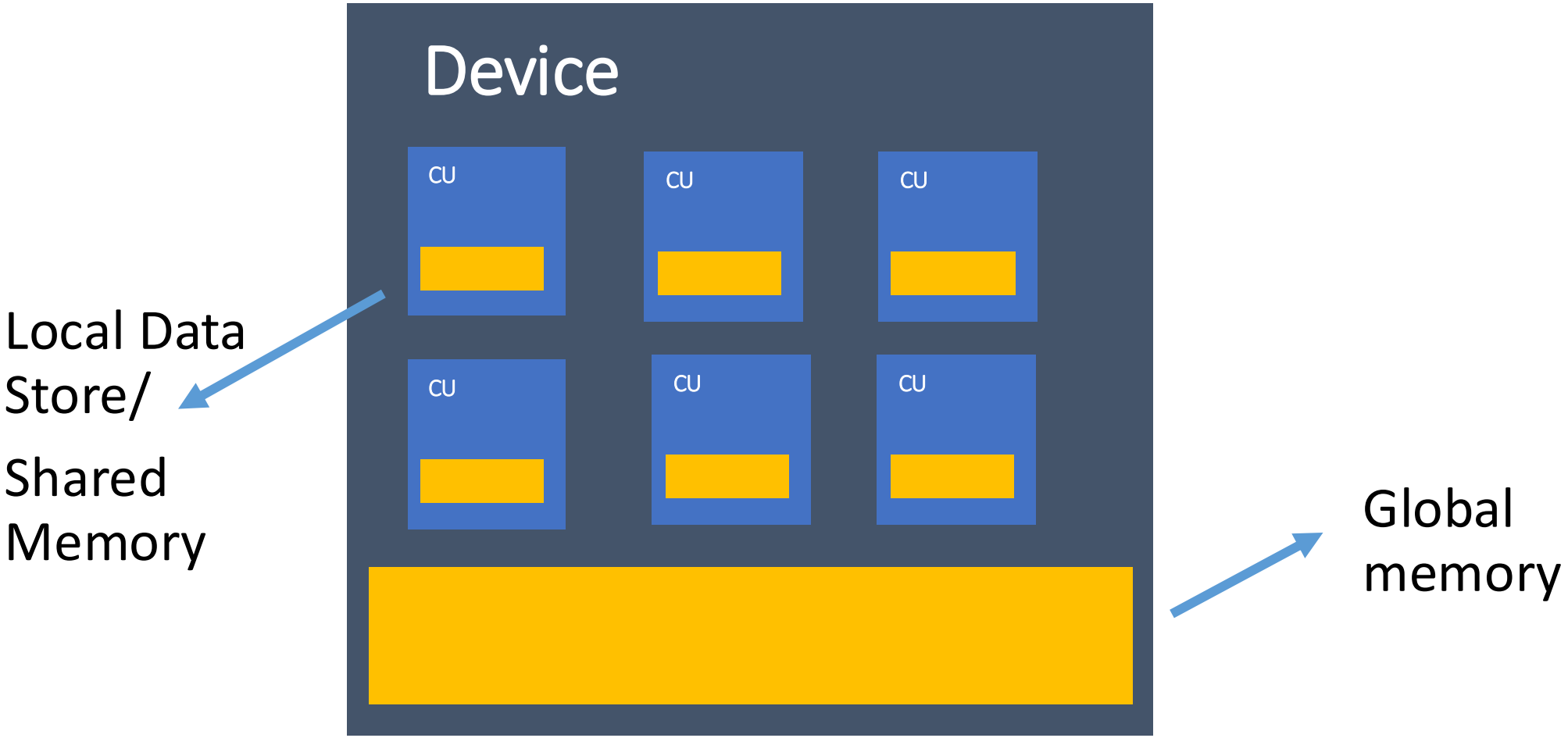
Unified memory

- Standard allocators are used, which are not GPU aware.
- Needs to use **`requires unified_shared_memory`**
- Host can access pages on the device and vice-versa
- Mapping is optional

Unified memory

- Does not work on older devices
- Performance is good only on architectures that have fast interconnect between host and device memory (MI300A, Grace-Hopper etc..).

Memory on the Device



Memory

Device	
Global memory	Shared by all threads on GPU
	Slower than the local data store
Local Data Store	Shared by threads on a CU/team
	Fast but small

Host	
Pageable	Allocated with malloc, new, allocate
	Slow data transfer to GPU
Pinned memory	Fast data transfer
	Requires special allocation calls

OpenMP memory allocators

epcc

OpenMP allocators

epcc

- Define how to allocate memory; is a combination of a memory space and an allocator traits.

- Memory space: hint to the compiler on where to allocate memory. The `omp_default_mem_space` memory space is sufficient for most applications.

- Allocator traits: hints to the compiler on how the memory should be allocated and how the memory will be accessed.
  - *Implementation is undefined*.

OpenMP allocator traits

Allocator trait	Allowed values	Notes
alignment	A positive integer must be a multiple of 2	
access	all	can be accessed from anywhere.
	cgroup	can be accessed by all threads in a contention group.
	pteam	only shared by threads in a team.
	thread	memory private to each thread.
pinned	true , false	

OpenMP allocation on the host

```C/C++
// Use the default memory space.
omp_memspace_handle_t c_memspace = omp_default_mem_space;
// Create an array of 2 traits. The first trait signals that the memory should be pinned.
// The second trait specifies that the memory should be allocated with 128 byte alignment.
omp_alloctrait_t c_traits[2] = { { omp_atk_pinned , true }, {omp_atk_alignment, 128} } ;
// Create a custom allocator.
// Takes as input the memory space, the number of traits and and array of traits.
omp_allocator_handle_t my_allocator = omp_init_allocator(c_memspace, 2, c_traits);
```

C/C++

```Fortran
! Use the default memory space.
integer(omp_memspace_handle_kind) :: c_memspace = omp_default_mem_space
! Create an arry of 2 traits.
type(omp_alloctrait) :: c_traits(2)
! Create a custom allocator.
integer(omp_allocator_handle_kind) :: c_alloc
c_traits(1) = omp_alloctrait(omp_atk_pinned, .TRUE.)
c_traits(2) = omp_alloctrait(omp_atk_alignment, 128)
c_alloc = omp_init_allocator(c_memspace, 2, c_traits)
```

Fortran

OpenMP allocation on the host

- The allocate directive provides a hint to the compiler. It is not guaranteed that the compiler will use the traits you provide.
- You can use the clause `allocate` (`my_allocator`: <item-list>)
- The clause needs to be associated with an allocation statement

OpenMP allocation on the host

```
#pragma omp allocate(my_alloc:b)
b = new double [n];
```
C/C++

```
!$omp allocate(my_alloc:b)
allocate (b(n))
```
Fortran

OpenMP allocation on the device

- On the device you must specify which allocators can be used on the host using the `uses_allocators` clause

C/C++

```
#pragma omp target uses_allocators(my_allocator)
    map(alloc:c[0:BLOCK_SIZE]) allocate(my_allocator:c)
```

Fortran

```
real, allocatable, dimension(:) :: c
!$omp target teams num_teams(4) &
!$omp private(c) &
!$omp uses_allocators(my_allocator) &
!$omp map(alloc:c(1:BLOCK_SIZE)) &
!$omp allocate(my_allocator:c)
```

OpenMP allocation on the LDS

- OpenMP does not (yet) provide a mechanism for allocating on the local store.
- However, you can provide hints to the compiler.
- You can use also compiler extensions, but they are not portable between compilers

OpenMP allocation on the LDS

- Statically allocate memory inside a team.

```c
#pragma omp target teams num_teams(4) {
    double c[BLOCK_SIZE];
    // do something with a team
    #pragma omp parallel for
    for(int i=0; i<BLOCK_SIZE; i++) {
        c[i] = i;
    }
}
```

C/C++

```fortran
program main
  implicit none
  !$omp target teams num_teams(4)
    call loop()
  !$omp end target teams
end program main

subroutine loop
  integer, parameter :: N = 1000
  integer :: i
  real, dimension(N) :: c
  !$omp parallel do
  do i = 1, N
    c(i) = i
  end do
end subroutine loop
```

Fortran

OpenMP allocation on the LDS

- Use the default allocator `omp_pteam_mem_alloc`
- On some compilers you need to use `omp_cgroup_mem_alloc`
- On some compilers neither is implemented

```c
double c[BLOCK_SIZE];

#pragma omp target teams
    num_teams(4) private(c)
    uses_allocators(omp_pteam_mem_alloc)
    map(alloc:c[0:BLOCK_SIZE])
    allocate(omp_pteam_mem_alloc:c)

#pragma omp parallel for shared(c)
for(int i=0;i<BLOCK_SIZE;i++) {
    // do something with c
}
```

C/C++

```fortran
real, allocatable, dimension(:) :: c
!$omp target teams num_teams(4) &
!$omp private(c) &
!$omp uses_allocators(omp_pteam_mem_alloc) &
!$omp map(alloc:c(1:BLOCK_SIZE)) &
!$omp allocate(omp_pteam_mem_alloc:c)

!$omp parallel do shared(c)
do i = 1, BLOCK_SIZE
  ! do something with c
end do
!$omp end target teams
```

Fortran

Asynchronous offloading

Asynchronous offloading

- map directives are blocking : the CPU and all GPU kernel launches will stall until the data is transferred to the GPU.
- kernels offloaded to the device are blocking: the CPU will wait until the kernel finishes executing on the GPU. Only one kernel at a time is executed.
- You can use non blocking calls by adding the `nowait` clause.
- Synchronize tasks launched from a CPU thread using the `taskwait` directive.
- Synchronize tasks and multiple CPU threads using the `barrier` directive.
- Specify dependencies using the `depend` clause.

Asynchronous data transfer

- Add a nowait clause to map directives to overlap memory transfers and computations

```c
#pragma omp target enter data map(to:xsi[0:m],ysi[0:m]) nowait
depend(out:xsi[0:m],ysi[0:m])

my_host_computation()

#pragma omp taskwait
```

C/C++

```fortran
!$omp target enter data map(alloc:xsi(1:m),ysi(1:m)) nowait
depend(out:xsi(1:m),ysi(1:m))

call my_host_computation()

!$omp taskwait
```

Fortran

Asynchronous execution

- Add a **`nowait`** clause to overlap computation on the host and device.

```c
#pragma omp target teams distribute parallel for nowait
for (int j=0; j<m; j++) {
    // computation A on device
}
for (int j=0; j<m; j++) {
    // independent computation B on host
}
#pragma omp taskwait
```

C/C++

```fortran
!$omp target teams distribute parallel do nowait
do j = 1, m
    ! computation A on device
end do
do j = 1, m
    ! independent computation B on host
end do
!$omp taskwait
```

Fortran

Parallel execution on the device

- Add a `nowait` clause to allow running multiple kernels on the device at the same time.
- Compiler might choose to run the kernels serially on the device or in parallel.
- Only advantageous if the offloaded region does not exhaust the resources on the device.

Parallel execution on the device

```c
#pragma omp target teams distribute parallel for nowait
for (int j=0; j<m; j++) {
    // computation A
}
#pragma omp target teams distribute parallel for nowait
for (int j=0; j<m; j++) {
    // independent computation B
}
#pragma omp taskwait
```

C/C++

```fortran
!$omp target teams distribute parallel do nowait
do j = 1, m
    ! computation A on device
end do
!$omp target teams distribute parallel do nowait
do j = 1, m
    ! independent computation B
end do
!$omp taskwait
```

Fortran

Dependencies

- If your kernels depend on data produced by other kernels you can tell that the kernel has a data dependency.
- You use the **depend** clause to mark data dependencies.

`depend(out:x)`	Mark that other tasks depend on the variable **x**, which might be modified by the current task.
`depend(in:x)`	This kernel depend on the variable **x**, modified by other kernels that specify the out modifier.
`depend(inout:x)`	Specify both input and output dependencies.

Dependencies

```
// Initiate data transfer. Mark x as a dependency of a future task.
#pragma omp target enter data map(to:x) nowait depend(out: x[0:m])

// Once x is available on the device start computation on the device.
#pragma omp target teams distribute parallel for nowait depend(inout: x[0:m])
for (int j=0; j<m; j++) {
    // update the array x
}

// Once x is updated from the previous kernel start this kernel.
#pragma omp target teams distribute parallel for nowait depend(in: x[0:m])
for (int j=0; j<m; j++) {
    // use the x array for my computation
}
```

C/C++

Dependencies

```fortran
! Initiate data transfer. Mark x as a dependency of a future task.
!$omp target enter data map(to:x) nowait depend(out: x(1:m))

! Once x is available on the device start computation on the device.
!$omp target teams distribute parallel do nowait depend(inout: x(1:m))
do j = 1, m
    ! update the array x
end do

! Once x is updated from the previous kernel start this kernel.
!$omp target teams distribute parallel do nowait depend(in: x(1:m))
do j = 1, m
    ! use the x array for my computation
end do
```

Fortran

Summary

- OpenMP/ROCm/CUDA interoperability
- Custom data mapper
- Memory types
- Memory allocation
- Asynchronous tasks