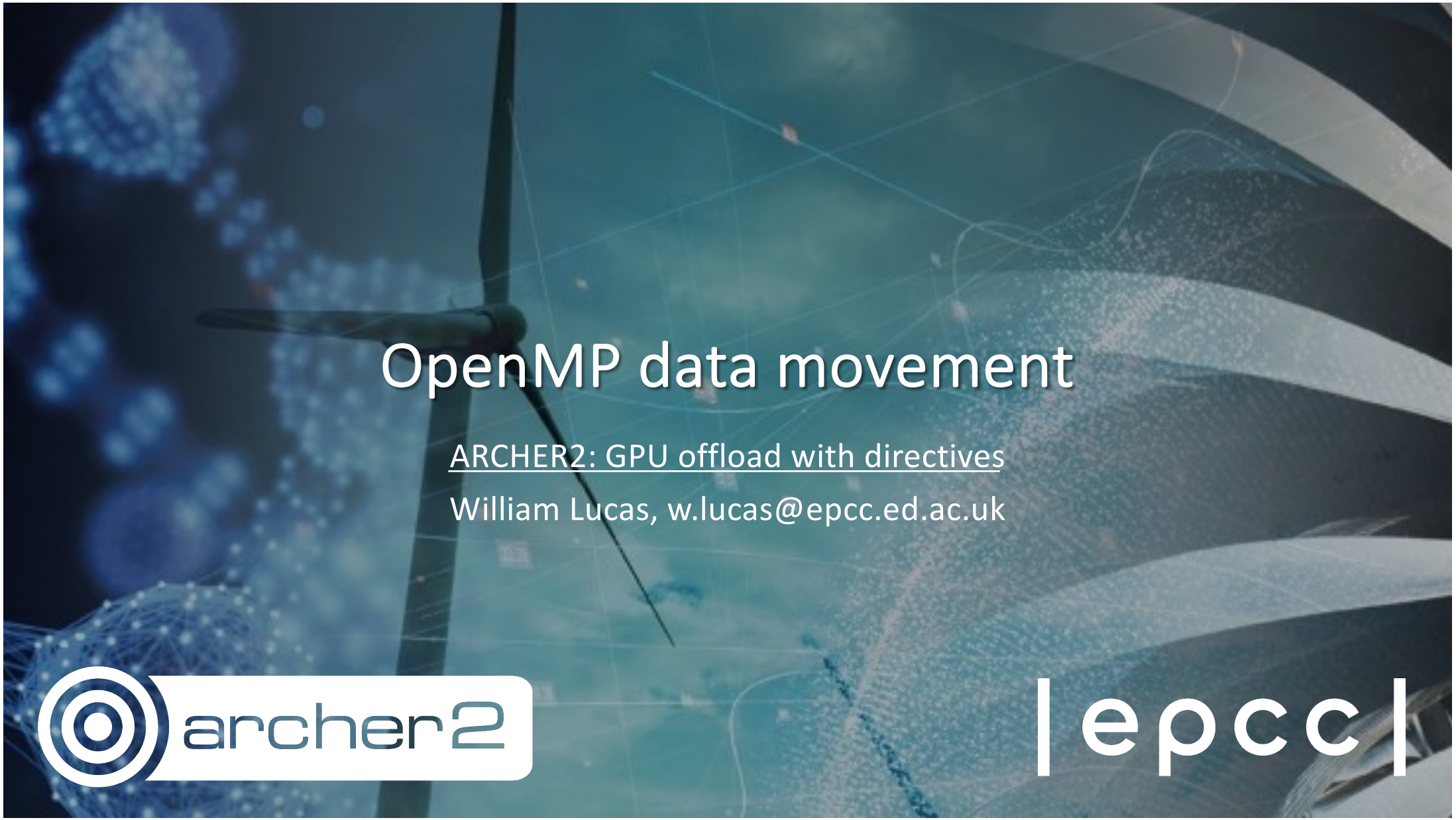

OpenMP data movement

ARCHER2: GPU offload with directives

William Lucas, w.lucas@epcc.ed.ac.uk

archer2

epcc

Reusing this material

Partners

epcc

epcc

THE UNIVERSITY of EDINBURGH

Hewlett Packard Enterprise

Outline

- GPU data movement basics
- `map` directives
  - Reference counting
  - Types and modifiers
- `data` directives
  - Structured memory management
  - Unstructured memory management
- Unified memory
- A few tips

Basics of data movement

- The ‘traditional’ GPU model has separate, non-unified memory on the host and on the device(s).
  - Host memory is the CPU memory
    - Where all our ‘normal’ code and variables live.
    - Where our program actually executes.
  - Device memory is the GPU memory
    - Can copy data to and from this memory from the host memory.
    - Individual GPU kernels are spawned from the CPU program and run here.
- NVIDIA Grace Hopper and AMD MI300A have superchip designs
  - ‘Unified memory’
  - Can affect approach to data movement.
    - And potentially remove the need to deal with this altogether.

Basics of data movement

- This means that any data the GPU will work on needs to be present in the GPU's memory.
- Why is this important?

DATA MOVEMENT IS SLOW!

Implicit data movement

- All data used in GPU work needs to be in memory visible to the GPU.
  - Otherwise, what will the GPU do work on?
  - This traditionally means the GPU's own memory, separate from CPU memory.
  - Newer unified memory designs in some way combine CPU and GPU memory.
- OpenMP keeps track of what data is available on GPU.
- If an array is used in a `#pragma omp target`/`$omp target` region isn't already on the GPU, then it is implicitly copied in at the beginning and copied back out at the end i.e. `map(tofrom:…)`.
- Scalars are implicitly `firstprivate`.

map directive

- What if we want to perform repeated calculations on the GPU with the same variables across multiple `#pragma omp target`/`$omp target` regions?
- Repeated transfers are costly.
- Instead, keep the data on GPU and only move when necessary.
  - Copy variables from CPU to GPU only when required.
  - Copy variables from GPU to CPU only when required.
- Use the OpenMP `map` directive to control data movement.
- Almost always safer and more performant to do `map` manually rather than let things be taken care of through the implicit rules.
  - Just like how it's safest to use `default(none)` in 'regular' OpenMP.

map directive

- Map directives let us tell OpenMP:
  - Which data should be copied to device memory (from main CPU memory).
  - Which data should be copied from device memory (to main CPU memory).
- Use `map(to: a, b)` to copy variables `a` and `b` from host memory into device memory on entry to this OpenMP region.
  - Values of `a` and `b` on host unchanged by the `target` region.
- Use `map(from: a, b)` to copy variables `a` and `b` from device memory into host memory on exit from this OpenMP region.
  - Values of `a` and `b` are uninitialised on device on entry to the `target` region.
- Use `map(tofrom: a, b)` to copy variables a and b from host to device memory on entry and back again from device to host on exit.

Reminder of `map(…)`

```
#pragma omp target teams distribute parallel for \
map(to: B, C), map(from: A)
for (int i=0; i<N; i++) {
  A[i] = B[i] + C[i];
}
```
C/C++

```
!$omp target teams distribute parallel do
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(from: A(1:N))
do i = 1,N
  A(i) = B(i) + C(i)
end do
!$omp end target teams distribute parallel do
```
Fortran

Arrays with map

epcc

- With C/C++, need to provide sizes with dynamic arrays:

```
int A[N], B[N], C[N];
. . .
#pragma omp target teams distribute parallel for \
map(to: B, C), map(from: A)
```

Static

```
int *A, *B, *C;
A = (int*) malloc(N*sizeof(int));
...
#pragma omp target teams distribute parallel for \
map(to: B[0:N], C[0:N]), map(from: A[0:N])
```

Dynamic

Reference counts

- OpenMP tracks reference counts to the list items in `map` clauses.
- A `map(to:…)` clause increments the count by 1 on entry, and `map(from:…)` likewise decrements it by 1 on exit.
- If a count is 0 on exiting a region, the variable is deallocated.
- For example:
  - With a prior reference count of 0, on entering `map(to:…)` will allocate counterparts of the listed host variables on the device memory, give them a reference count of 1, and copy their values in from the host memory.
  - With a prior reference count of 1, `map(from:…)` will copy values from the listed variables on device memory back to their counterparts on the host, decrement their reference counts to 0, and then deallocate them from the device.

Reference counts

- OpenMP tracks reference counts to the variables in offload regions.
- Generally, but not specifically, if a variable is in an offload region:
  - Its reference count increments by 1 on entry to the region.
  - Its reference count decrements by 1 on exit from the region.
- Allows OpenMP to do a presence check: is a device counterpart to the host variable already present?
  - This can occur with nested offload data regions as we'll see.
- If already present, behaviour of `to`, `from` and `tofrom` change: they copy no data and instead only check that sizes are correct.
- If the ref count = 0 on exit, OpenMP knows it can safely deallocate.

More `map` types

- These offer more control if desired. None perform data copies.
- `map(alloc: …)`
  - If listed variable is not already present on device then allocate counterpart to the host variable there, and set the reference count to 1.
  - If already present, check size on device and increment reference count.
- `map(release: …)`
  - On exit from the region decrement the reference count for the listed variables by 1 and delete the device counterpart if it's now 0.
- `map(delete: …)`
  - On exit from the region, set the reference count for listed variables to 0 and delete device counterparts.

Map modifiers

- Add even further control with these modifiers to be used in `map()`.
- `always` – Always copy data `to`, `from` and `tofrom` the device. These otherwise don't copy data if it's already present (ref count $\geqslant 1$).
- `close` – A hint to allocate memory `close` to the target device.
- `mapper` – Create with `declare mapper` and then use in `map` to choose how derived types and structs should be mapped.
- `present` – Run a `map` with `present` first before any without.
- `iterator` – Control indexing of variables, e.g. map every second element of `A:`
  - `map(iterator(int i=0:N:2), to: A[i])` (C/C++)
  - `map(iterator(integer :: i=0:N:2), to: A(i))` (Fortran)

Intro to structured data regions

- This code is not efficient!
  - Both B and C are copied to the GPU on entry to each parallel for.
- If they don't change in the meantime, why not just copy once and leave the data there for both kernels?
- It's often the case that we want to run multiple kernels on the GPU while keeping data there.

```C/C++
#pragma omp target teams distribute \
parallel for map(to: B, C), map(from: A)
for (int i=0; i<N; i++) {
  A[i] = B[i] + C[i];
}
#pragma omp target teams distribute \
parallel for map(to: B, C), map(from: D)
for (int i=0; i<N; i++) {
  D[i] = B[i] * C[i];
}
```

Intro to structured data regions

- It's often the case that we want to run multiple kernels on the GPU while keeping data there.
- This code is not efficient!
  - Both B and C are copied to the GPU on entry to each parallel for.
- If they don't change in the meantime, why not just copy once and leave the data there for both kernels?

```fortran
!$omp target teams distribute
!$omp& parallel do
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(from: A(1:N))
do i = 1,N
  A(i) = B(i) + C(i)
end do
!$omp end target teams distribute
!$omp& parallel do
!$omp target teams distribute
!$omp& parallel do
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(from: D(1:N))
do i = 1,N
  D(i) = B(i) + C(i)
end do
!$omp end target teams distribute
!$omp& parallel do
```

Fortran

Structured data regions

- Use a `data` construct to define a larger data offload region.
- Provide a `map` clause to the construct to tell it, just as before, what to do with the variables inside the region.
- Now `B` and `C` are only copied in once on entry to the data region.

```C/C++
#pragma omp target data map(to: B, C)
{
#pragma omp target teams distribute \
parallel for map(from: A)
  for (int i=0; i<N; i++) {
    A[i] = B[i] + C[i];
  }
#pragma omp target teams distribute \
parallel for map(from: D)
  for (int i=0; i<N; i++) {
    D[i] = B[i] * C[i];
  }
} // end target data
```

Structured data regions

- Use a `data` construct to define a larger data offload region.
- Provide a `map` clause to the construct to tell it, just as before, what to do with the variables inside the region.
- Now `B` and `C` are only copied in once on entry to the data region.

```
!$omp target data map(to: B(1:N),C(1:N))
!$omp target teams distribute
!$omp& parallel do
!$omp& map(from: A(1:N))
do i = 1,N
  A(i) = B(i) + C(i)
end do
!$omp end target teams distribute
!$omp& parallel do
!$omp target teams distribute
!$omp& parallel do
!$omp& map(from: D(1:N))
do i = 1,N
  D(i) = B(i) + C(i)
end do
!$omp end target teams distribute
!$omp& parallel do
!$omp end target data
```

Fortran

Update in data region

- Remember: map clauses like `map(to: A)` would not copy inside the previous examples as the ref count inside the data region is already 1.
- If a change is needed, could use `map(always to: A)`, or `update`.

```C/C++
#pragma omp target data map(alloc: A, B)
{
// A and B were created on device, but no copy.
// Do some work, then copy A from host to device:
#pragma omp target update to(A)
// Do more work, then copy B from device to host.
#pragma omp target update from(B)
// Do more work.
}
```

Update in data region

- Remember: map clauses like `map(to: A)` would not copy inside the previous examples as the ref count inside the data region is already 1.
- If a change is needed, could use `map(always to: A)`, or `update:`

```
!$ omp target data map(alloc: A, B)
! A and B were created on device, but no copy.
! Do some work, then copy A from host to device:
!$omp target update to(A)
! Do more work, then copy B from device to host.
!$omp target update from(B)
! Do more work.
!$omp end target data
```

Fortran

Update in data region

- Can potentially make update even more performant by adding `if`, `depend` and/or `nowait` clauses.
  - Of course this makes code more complex and more prone to bugs.
  - Probably best to code without these initially, then try to introduce them for more performance once you're sure everything's working.
- `if`: provide an expression and only update if true.
  - `#pragma omp target update if(i >= 100) to(A)`
- `nowait`:
  - Can be added to target regions in general.

Unstructured `data` regions

epcc

- The prior `data` regions were structured: they began, OpenMP offloaded work to the GPU, then the data region ended.
- There is a requirement also for *un*structured `data` regions: define a region of code where data is created on the device, and a region of code where that data is copied back to host or deleted.
- Necessary e.g. in OOP: we might want to copy data to GPU memory in an object's constructor, later on do work with that data, then delete the data from the GPU memory in the object destructor.
- Use the `enter data` and `exit data` constructs.
  - Often use `enter` right after allocating, and `exit` right before freeing.

Data regions in objects (C++)

- `enter` the data region in constructor and `alloc` the array `A` on device
- `exit` the data region in the destructor and `delete` array `A` on device

```
struct example {
      int n; double *A;
      example(int n) : n(n) {
            A = new double[n];
#pragma omp target enter data map(alloc: A[0:n])
      }
      ~example() {
#pragma omp target exit data map(delete: A[0:n])
            delete[] A;
      }
}
```

C/C++

Data regions in objects (Fortran) I

```
module TExample_class
     implicit none
     type, public :: TExample
          integer, allocatable :: A(:)
     contains
          private
          final :: destructor
     end type TExample
     interface TExample
          procedure :: constructor
     end interface
contains
…
```

Fortran

Fortran object boilerplate…

Data regions in objects (Fortran) II

```
     function constructor(N) result(example)
          integer, intent(in) :: N
          type(TExample) :: example
          allocate(example%A(N))
!$omp target enter data map(alloc: example)
     end function constructor
     subroutine destructor(this)
          type(TExample) :: this
          if (allocated(this%A)) then
               deallocate(this%A)
!$omp target exit data map(delete: this)
          end if
     end subroutine destructor
end module TExample_class
```

Fortran

Unified memory

- Some GPUs support a single address space across both CPU and GPU memory
- This can be done in software, but it is not very efficient
  - Useful for testing, not so useful for production code!
- Some newer designs support this in hardware
  - Either physically just one memory (e.g AMD MI300A) or cache coherent NUMA (e.g NVIDIA GraceHopper).
- With unified memory support, we can omit all the data mapping clauses and have the code still work.

omp requires I

- To use unified shared memory we need a `requires` directive at the top of every routine that contains offloaded code:

```
void boo(...){
#pragma omp requires unified_shared_memory
  ...
}
```

C/C++

```
subroutine boo(...)
implicit none
!$omp requires unified_shared_memory
  ...
end subroutine
```

Fortran

`omp requires` II

- However, for some compilers the code will still work without the `requires` directive.
- May depend on compiler flags and/or environment variables.
  - Beware of cases where the wrong settings give correctly functioning code but silently deliver bad performance!

Tips

- When mapping data, think of all words like ‘to’ and ‘from’ as being from the CPU’s perspective, looking out towards the GPU.

- When keeping data consistent between host and device, remember a simple `map(to:…)` or `map(from:…)` will be ignored if data is already present on device!
  - Either use `update` or force a `map` with the `always` modifier.