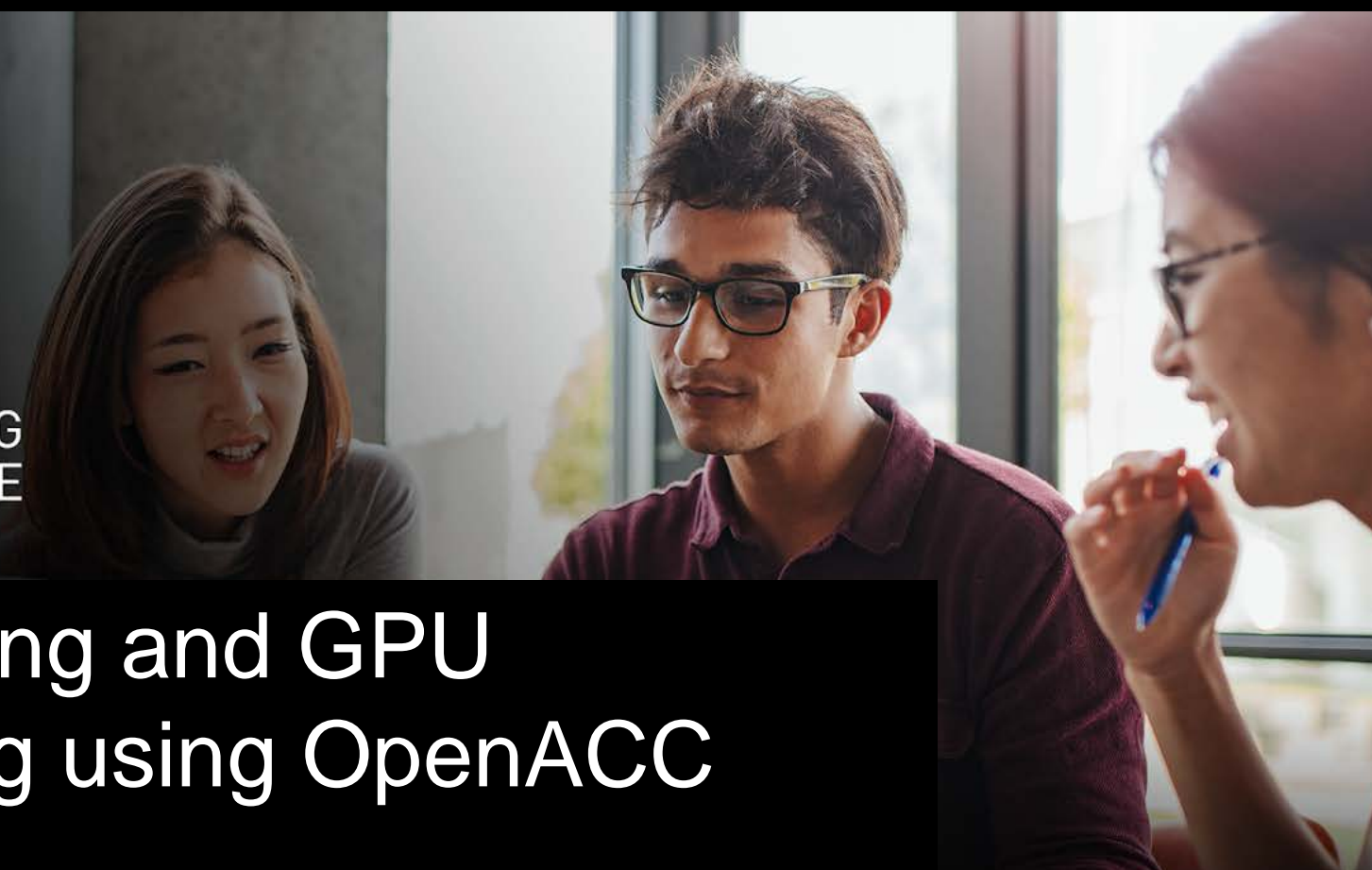




DEEP
LEARNING
INSTITUTE



Deep Learning and GPU programming using OpenACC

14 – 17 July 2020



MODULE FIVE: DATA MANAGEMENT

Dr. Volker Weinberg | LRZ | 16.07.2020

MODULE OVERVIEW

OpenACC Data Management

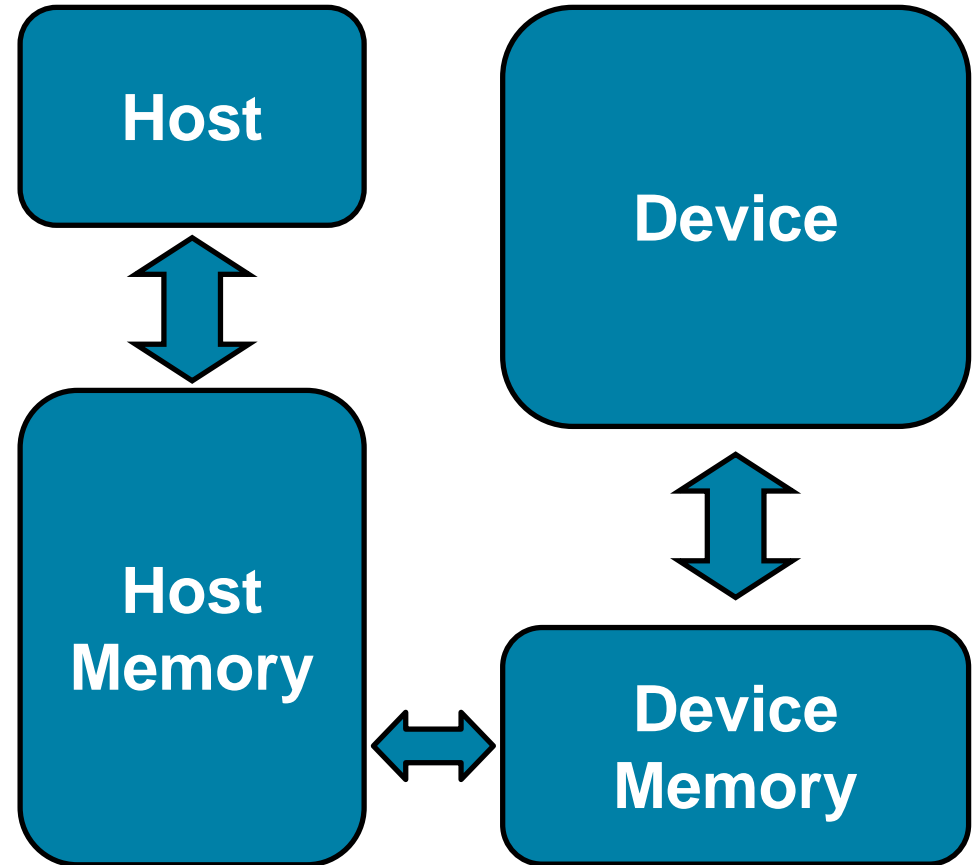
- Explicit Data Management
- OpenACC Data Regions and Clauses
- Unstructured Data Lifetimes
- Data Synchronization

EXPLICIT MEMORY MANAGEMENT

EXPLICIT MEMORY MANAGEMENT

Requirements

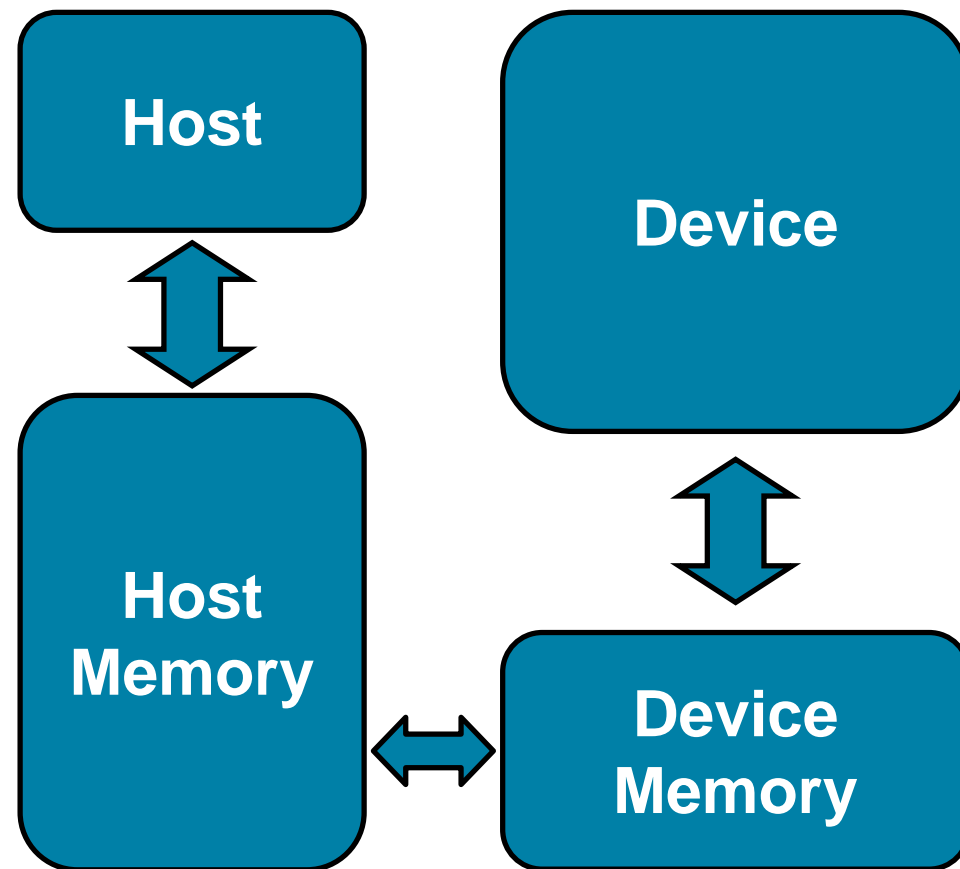
- Data must be visible on the **device** when we run our **parallel** code
- Data must be visible on the **host** when we run our **sequential** code
- When the host and device don't share memory, data movement must occur
- To maximize performance, the programmer should avoid all unnecessary data transfers



EXPLICIT MEMORY MANAGEMENT

Key problems

- Many parallel accelerators (such as devices) have a separate memory space from the host
- These separate memories can become out-of-sync and contain completely different data
- Transferring between these two memories can be a very time consuming process



OPENACC DATA DIRECTIVE

OPENACC DATA DIRECTIVE

Definition

- The data directive defines a lifetime for data on the device
- During the region data should be thought of as residing on the accelerator
- Data clauses allow the programmer to control the allocation and movement of data

```
#pragma acc data clauses  
{  
    < Sequential and/or Parallel code >  
}
```

```
!$acc data clauses  
    < Sequential and/or Parallel code >  
!$acc end data
```


DATA CLAUSES

`copy(list)`

Allocates memory on device and copies data from host to device when entering region and copies data to the host when exiting region.

Principal use: For many important data structures in your code, this is a logical default to input, modify and return the data.

`copyin(list)`

Allocates memory on device and copies data from host to device when entering region.

Principal use: Think of this like an array that you would use as just an input to a subroutine.

`copyout(list)`

Allocates memory on device and copies data to the host when exiting region.

Principal use: A result that isn't overwriting the input data structure.

`create(list)`

Allocates memory on device but does not copy.

Principal use: Temporary arrays.

IMPLIED DATA REGIONS

IMPLIED DATA REGIONS

Definition

- Every **kernels** and **parallel** region has an implicit data region surrounding it
- This allows data to exist solely for the duration of the region
- All data clauses usable on a **data** directive can be used on a **parallel** and **kernels** as well

```
#pragma acc kernels copyin(a[0:100])
{
    for( int i = 0; i < 100; i++ )
    {
        a[i] = 0;
    }
}
```

IMPLIED DATA REGIONS

Explicit vs Implicit Data Regions

Explicit

```
#pragma acc data copyin(a[0:100])
{
    #pragma acc kernels
    {
        for( int i = 0; i < 100; i++ )
        {
            a[i] = 0;
        }
    }
}
```

Implicit

```
#pragma acc kernels copyin(a[0:100])
{
    for( int i = 0; i < 100; i++ )
    {
        a[i] = 0;
    }
}
```

These two codes are functionally the same.

EXPLICIT VS. IMPLICIT DATA REGIONS

Limitation

Explicit

1 Data Copy

```
#pragma acc data copyout(a[0:100])
{
    #pragma acc kernels
    {
        a[i] = i;
    }

    #pragma acc kernels
    {
        a[i] = 2 * a[i];
    }
}
```

Implicit

2 Data Copies

```
#pragma acc kernels copyout(a[0:100])
{
    a[i] = i;
}

#pragma acc kernels copy(a[0:100])
{
    a[i] = 2 * a[i];
}
```

The code on the left will perform better than the code on the right.

UNSTRUCTURED DATA DIRECTIVES

UNSTRUCTURED DATA DIRECTIVES

Enter Data Directive

- Data lifetimes aren't always neatly structured.
- The **enter data** directive handles device memory **allocation**
- You may use either the **create** or the **copyin** clause for memory allocation
- The enter data directive is **not** the start of a data region, because you may have multiple enter data directives

```
#pragma acc enter data clauses
```

```
< Sequential and/or Parallel code >
```

```
#pragma acc exit data clauses
```

```
!$acc enter data clauses
```

```
< Sequential and/or Parallel code >
```

```
!$acc exit data clauses
```

UNSTRUCTURED DATA DIRECTIVES

Exit Data Directive

- The **exit data** directive handles device memory **deallocation**
- You may use either the **delete** or the **copyout** clause for memory deallocation
- You should have as many **exit data** for a given array as **enter data**
- These can exist in different functions

```
#pragma acc enter data clauses
```

```
< Sequential and/or Parallel code >
```

```
#pragma acc exit data clauses
```

```
!$acc enter data clauses
```

```
< Sequential and/or Parallel code >
```

```
!$acc exit data clauses
```


UNSTRUCTURED DATA CLAUSES

- `copyin (list)` Allocates memory on device and copies data from host to device on enter data.
- `copyout (list)` Allocates memory on device and copies data back to the host on exit data.
- `create (list)` Allocates memory on device without data transfer on enter data.
- `delete (list)` Deallocates memory on device without data transfer on exit data

UNSTRUCTURED DATA DIRECTIVES

Basic Example

```
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
}
```

UNSTRUCTURED DATA DIRECTIVES

Basic Example

```
#pragma acc enter data copyin(a[0:N],b[0:N]) create(c[0:N])

#pragma acc parallel loop
for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
}

#pragma acc exit data copyout(c[0:N])
```

UNSTRUCTURED DATA DIRECTIVES

Basic Example

```
#pragma acc enter data copyin(a[0:N], b[0:N]) create(c[0:N])  
  
#pragma acc parallel loop  
for(int i = 0; i < N; i++){  
    c[i] = a[i] + b[i];  
}  
  
#pragma acc exit data copyout(c[0:N])
```

Action

Copy B
Data from
device to
device

CPU MEMORY



device MEMORY



UNSTRUCTURED DATA DIRECTIVES

Basic Example – proper memory deallocation

```
#pragma acc enter data copyin(a[0:N],b[0:N]) create(c[0:N])

#pragma acc parallel loop
for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
}

#pragma acc exit data copyout(c[0:N]) delete(a,b)
```

Action

Deallocate ~~B~~
from
device

CPU MEMORY



device MEMORY



UNSTRUCTURED VS STRUCTURED

With a simple code

Unstructured

- Can have multiple starting/ending points
- Can branch across multiple functions
- Memory exists until explicitly deallocated

```
#pragma acc enter data copyin(a[0:N],b[0:N]) \  
create(c[0:N])
```

```
#pragma acc parallel loop  
for(int i = 0; i < N; i++){  
    c[i] = a[i] + b[i];  
}
```

```
#pragma acc exit data copyout(c[0:N]) \  
delete(a,b)
```

Structured

- Must have explicit start/end points
- Must be within a single function
- Memory only exists within the data region

```
#pragma acc data copyin(a[0:N],b[0:N]) \  
copyout(c[0:N])
```

```
{  
    #pragma acc parallel loop  
    for(int i = 0; i < N; i++){  
        c[i] = a[i] + b[i];  
    }  
}
```

UNSTRUCTURED DATA DIRECTIVES

Branching across multiple functions

```
int* allocate_array(int N){
    int* ptr = (int *) malloc(N * sizeof(int));
    #pragma acc enter data create(ptr[0:N])
    return ptr;
}

void deallocate_array(int* ptr){
    #pragma acc exit data delete(ptr)
    free(ptr);
}

int main(){
    int* a = allocate_array(100);
    #pragma acc kernels
    {
        a[0] = 0;
    }
    deallocate_array(a);
}
```

- In this example enter data and exit data are in different functions
- This allows the programmer to put device allocation/deallocation with the matching host versions
- This pattern is particularly useful in C++, where structured scopes may not be possible.

DATA SYNCHRONIZATION

OPENACC UPDATE DIRECTIVE

update: Explicitly transfers data between the host and the device

Useful when you want to synchronize data in the middle of a data region

Clauses:

self: makes host data agree with device data

device: makes device data agree with host data

```
#pragma acc update self(x[0:count])  
#pragma acc update device(x[0:count])
```

C/C++

```
!$acc update self(x(1:end_index))  
!$acc update device(x(1:end_index))
```

Fortran

OPENACC UPDATE DIRECTIVE

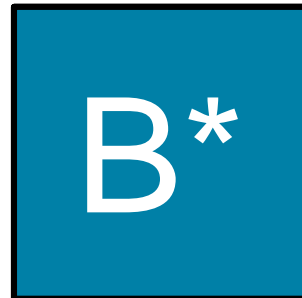
```
#pragma acc update device(A[0:N])
```



CPU Memory



device Memory



```
#pragma acc update self(A[0:N])
```

The data must exist on both the CPU and device for the update directive to work.

SYNCHRONIZE DATA WITH UPDATE

```
int* allocate_array(int N){
    int* A=(int*) malloc(N*sizeof(int));
    #pragma acc enter data create(A[0:N])
    return A;
}

void deallocate_array(int* A){
    #pragma acc exit data delete(A)
    free(A);
}

void initialize_array(int* A, int N){
    for(int i = 0; i < N; i++){
        A[i] = i;
    }
    #pragma acc update device(A[0:N])
}
```

- Inside the **initialize** function we alter the host copy of 'A'
- This means that after calling **initialize** the host and device copy of 'A' are out-of-sync
- We use the **update** directive with the **device** clause to update the device copy of 'A'
- Without the **update** directive later compute regions will use incorrect data.

COPYING DATA IN DATA REGIONS

```
#pragma acc enter data copyin(A[:m*n],Anew[:m*n])  
#pragma acc parallel loop copy(A,Anew)  
for( int j = 1; j < n-1; j++)
```

- But wouldn't this code now result in my arrays being copied twice, once by the `data` region and then again by the `parallel loop`? In fact, the OpenACC runtime is smart enough to handle exactly this case. Data will be copied `_in_` only the first time its encountered in a data clause and `_out_` only the last time its encountered in a data clause. This allows you to create fully-working directives within your functions and then later `_"hoist_"` the data movement to a higher level without changing your code at all. This is part of incrementally accelerating your code to avoid incorrect results.

MODULE REVIEW

KEY CONCEPTS

In this module we discussed...

- Why explicit data management is necessary for best performance
- Structured and Unstructured Data Lifetimes
- Explicit and Implicit Data Regions
- The **data**, **enter data**, **exit data**, and **update** directives
- Data Clauses

LAB ASSIGNMENT

In this module's lab you will...

- Update the code from the previous module to use explicit data directives
- Analyze the different between using CUDA Managed Memory and explicit data management in the lab code.