

# DATA MANAGEMENT

**OpenACC**  
More Science, Less Programming



DEEP  
LEARNING  
INSTITUTE

## MODULE OVERVIEW

### OpenACC Data Management

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

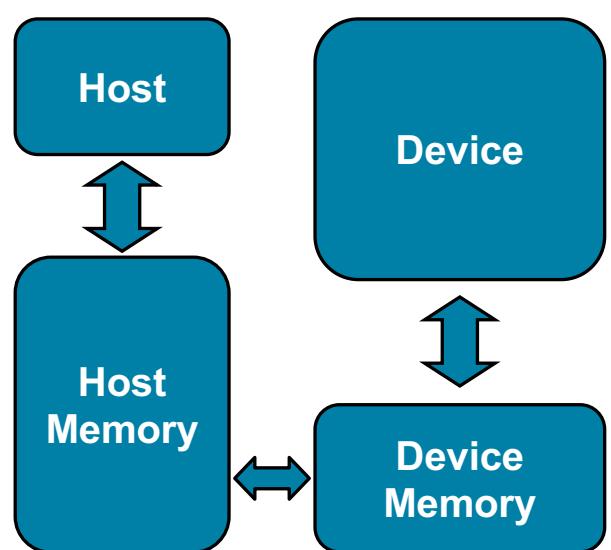


# EXPLICIT MEMORY MANAGEMENT

## EXPLICIT MEMORY MANAGEMENT

### Requirements

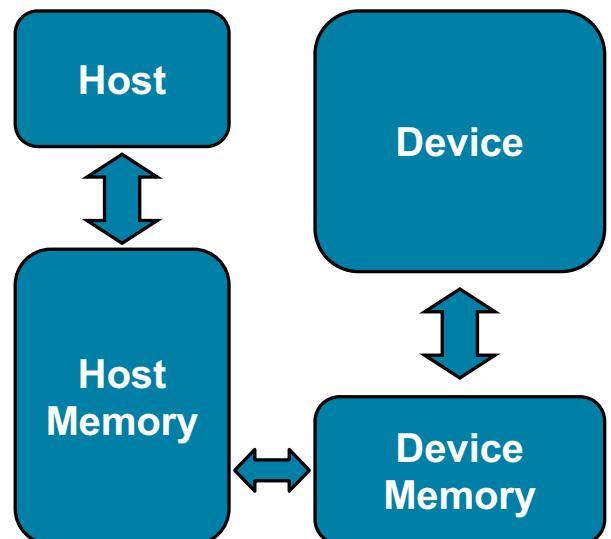
- Data must be visible on the **device** when we run our **parallel** code
- Data must be visible on the **host** when we run our **sequential** code
- When the host and device don't share memory, data movement must occur
- To maximize performance, the programmer should avoid all unnecessary data transfers



# EXPLICIT MEMORY MANAGEMENT

## Key problems

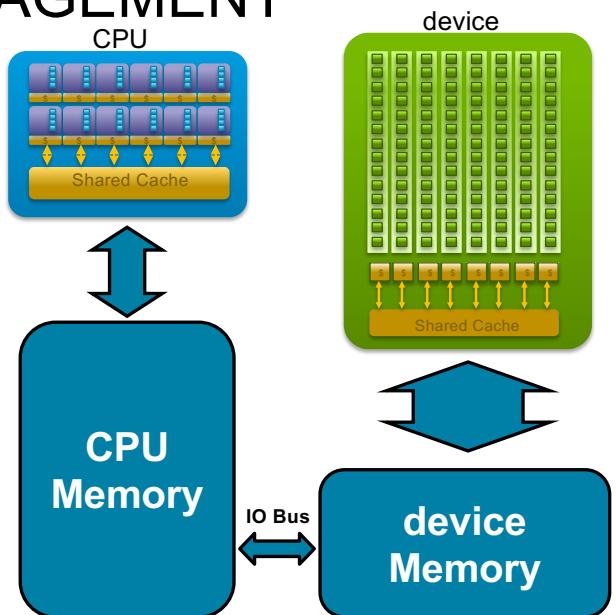
- Many parallel accelerators (such as devices) have a separate memory space from the host
- These separate memories can become out-of-sync and contain completely different data
- Transferring between these two memories can be a very time consuming process



# EXPLICIT MEMORY MANAGEMENT

## Key problems

- Many parallel accelerators (such as devices) have a separate memory pool 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 >
}
```

# DATA CLAUSES

<code>copy( list )</code>	<b>Allocates memory on device and copies data from host to device when entering region and copies data to the host when exiting region.</b>
	<b>Principal use:</b> For many important data structures in your code, this is a logical default to input, modify and return the data.
<code>copyin( list )</code>	<b>Allocates memory on device and copies data from host to device when entering region.</b>
	<b>Principal use:</b> Think of this like an array that you would use as just an input to a subroutine.
<code>copyout( list )</code>	<b>Allocates memory on device and copies data to the host when exiting region.</b>
	<b>Principal use:</b> A result that isn't overwriting the input data structure.
<code>create( list )</code>	<b>Allocates memory on device but does not copy.</b>
	<b>Principal use:</b> Temporary arrays.

# ARRAY SHAPING

- Sometimes the compiler needs help understanding the *shape* of an array
- The first number is the start index of the array
- In C/C++, the second number is how much data is to be transferred
- In Fortran, the second number is the ending index

`copy(array[starting_index:length])`

C/C++

# ARRAY SHAPING (CONT.)

## Multi-dimensional Array shaping

```
copy(array[0:N][0:M])
```

C/C++

Both of these examples copy a 2D array to the device

# ARRAY SHAPING (CONT.)

## Partial Arrays

```
copy(array[i*N/4:N/4])
```

C/C++

Both of these examples copy only  $\frac{1}{4}$  of the full array

# STRUCTURED DATA DIRECTIVE

## Example

This **parallel loop** will execute on the **accelerator**, so **a**, **b**, and **c** must be visible on the accelerator.

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

# STRUCTURED DATA DIRECTIVE

## Example

Start of 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];
    }
}
```

End of Data Region



# STRUCTURED DATA DIRECTIVE

Example

```
#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];
}
}
```

Action

Data locate AA  
Data move from host  
Data move to device

Host Memory



Device memory



OpenACC | NVIDIA | DEEP LEARNING INSTITUTE

## IMPLIED DATA REGIONS

OpenACC | NVIDIA | DEEP LEARNING INSTITUTE

# IMPLIED DATA REGIONS

## Definition

- Every **kernel**s 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 **kernel**s 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
```

# 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
```

# 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 transfer  
from host  
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  from device

CPU MEMORY



device MEMORY



OpenACC  
More Science. Less Programming.

## 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];
    }
}
```

OpenACC  
More Science. Less Programming.



DEEP  
LEARNING  
INSTITUTE

# 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

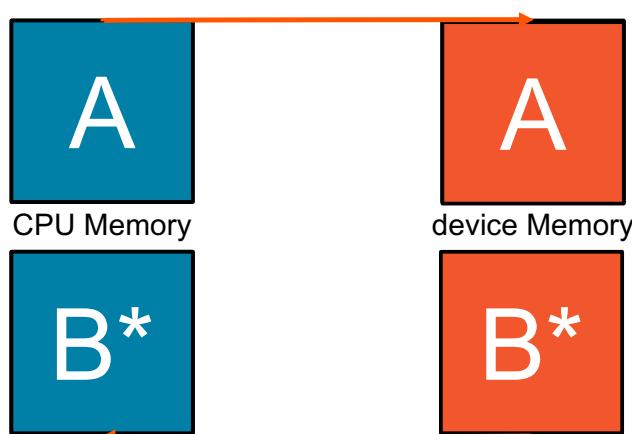
```
#pragma acc update self(x[0:count])
#pragma acc update device(x[0:count])
```

C/C++

# OPENACC UPDATE DIRECTIVE

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

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



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

# 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.

## C/C++ STRUCTS/CLASSES

# C STRUCTS

## Without dynamic data members

- Dynamic data members are anything contained within a struct that can have a **variable size**, such as dynamically allocated arrays
- OpenACC is easily able to copy our struct to device memory because everything in our float3 struct has a **fixed size**
- But what if the struct had dynamically allocated members?

```
typedef struct {  
    float x, y, z;  
} float3;  
  
int main(int argc, char* argv[]){  
    int N = 10;  
    float3* f3 = malloc(N * sizeof(float3));  
  
    #pragma acc enter data create(f3[0:N])  
  
    #pragma acc kernels  
    for(int i = 0; i < N; i++){  
        f3[i].x = 0.0f;  
        f3[i].y = 0.0f;  
        f3[i].z = 0.0f;  
    }  
  
    #pragma acc exit data delete(f3)  
    free(f3);  
}
```

# C STRUCTS

## With dynamic data members

- OpenACC does not have enough information to copy the struct and its dynamic members
- You must first copy the struct into device memory, then allocate/copy the dynamic members into device memory
- To deallocate, first deal with the dynamic members, then the struct
- OpenACC will automatically *attach* your dynamic members to the struct

```
typedef struct {  
    float *arr;  
    int n;  
} vector;  
  
int main(int argc, char* argv[]){  
  
    vector v;  
    v.n = 10;  
    v.arr = (float*) malloc(v.n*sizeof(float));  
  
    #pragma acc enter data copyin(v)  
    #pragma acc enter data create(v.arr[0:v.n])  
  
    ...  
  
    #pragma acc exit data delete(v.arr)  
    #pragma acc exit data delete(v)  
    free(v.arr);  
}
```

# C++ STRUCTS/CLASSES

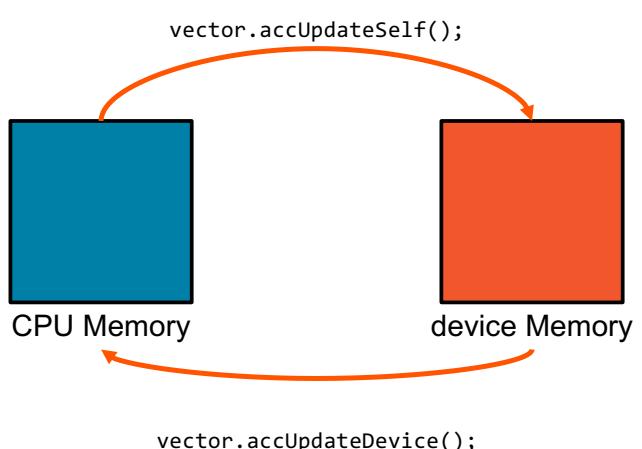
## With dynamic data members

- C++ Structs/Classes work the same exact way as they do in C
- The main difference is that now we have to account for the implicit “this” pointer

```
class vector {  
private:  
    float *arr;  
    int n;  
public:  
    vector(int size){  
        n = size;  
        arr = new float[n];  
        #pragma acc enter data copyin(this)  
        #pragma acc enter data create(arr[0:n])  
    }  
    ~vector(){  
        #pragma acc exit data delete(arr)  
        #pragma acc exit data delete(this)  
        delete(arr);  
    }  
};
```

# C++ CLASS DATA SYNCHRONIZATION

- Since data is encapsulated, the class needs to be extended to include data synchronization methods
- Including explicit methods for host/device synchronization may ease C++ data management
- Allows the class to be able to naturally handle synchronization, creating less code clutter

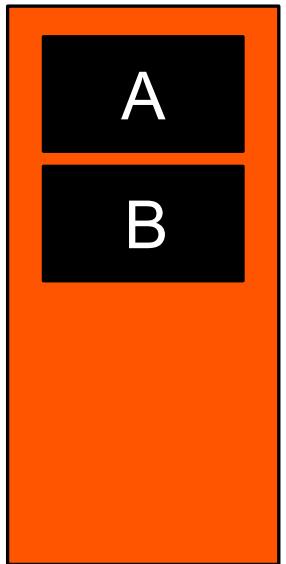
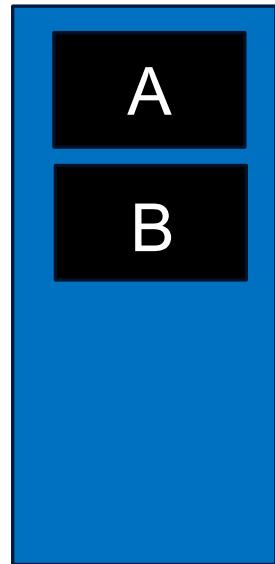


```
void accUpdateSelf() {  
    #pragma acc update self(arr[0:n])  
}  
void accUpdateDevice() {  
    #pragma acc update device(arr[0:n])  
}
```

# USING A OPENACC AWARE C++ CLASS

```
#include <vector.h>

int main() {
    vector A(N), B(N);
    for (int i=0; i < B.size(); ++i) {
        B[i]=2.5;
    }
    B.accUpdateDevice();
    #pragma acc parallel loop present(A,B)
    for (int i=0; i < A.size(); ++i) {
        A[i]=B[i]+i;
    }
    A.accUpdateSelf();
    for(int i=0; i<10; ++i) {
        cout << "A[" << i << "]": " << A[i] << endl;
    }
    exit(0);
}
```



Host Memory

Device Memory

## 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