MODULE FIVE: DATA MANAGEMENT

Speaker, Date

OpenACC
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
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.

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

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

**copy**\( ( \text{list} ) \) 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**\( ( \text{list} ) \) 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**\( ( \text{list} ) \) 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**\( ( \text{list} ) \) Allocates memory on device but does not copy.

**Principal use:** Temporary arrays.
This parallel loop will execute on the accelerator, so \(a\), \(b\), and \(c\) must be visible on the accelerator.

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

Example

```c
#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];
    }
}
```
STRUCTURED DATA DIRECTIVE

Example

```c
#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];
}
}
```

<table>
<thead>
<tr>
<th>Action</th>
<th>Host Memory</th>
<th>Device memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate A on device</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Copy A from CPU to device</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>Allocate B on device</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Allocate C on device</td>
<td>C'</td>
<td>C'</td>
</tr>
<tr>
<td>Execute loop on device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy C from device to CPU</td>
<td>C'</td>
<td></td>
</tr>
<tr>
<td>Deallocate C from device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deallocate B from device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deallocate A from device</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This **parallel loop** will execute on the **accelerator**, so **a**, **b**, and **c** must be visible on the accelerator.

```plaintext
!$acc parallel loop
do  i=1,N
   c(i) = a(i) + b(i)
end do
```
Example

```c
!$acc data copyin(a(1:N), b(1:N)) copyout(c(1:N))

 !$acc parallel loop
 do i=1,N
  c(i) = a(i) + b(i)
 end do

 !$acc end data
```
STRUCTURED DATA DIRECTIVE

Example

```
!$acc data copyin(a(1:N),b(1:N)) copyout(c(1:N))

!$acc parallel loop
do i=1,N
  c(i) = a(i) + b(i)
end do

!$acc end data
```

Action

- Allocate A on device
- Copy A from CPU to device
- Allocate B on device
- Copy B from CPU to device
- Allocate C on device
- Execute loop on device
- Copy C from device to CPU
- Deallocate C from device
- Deallocate B from device
- Deallocate A from device
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])`  \hspace{1cm} C/C++

`copy(array(starting_index:ending_index))`  \hspace{1cm} Fortran
ARRAY SHAPING (CONT.)

Multi-dimensional Array shaping

**C/C++**

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

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

**Fortran**

`copy(array(1:N, 1:M))`
Partial Arrays

C/C++

\[
\text{copy(array}[i \times N/4:i \times N/4+N/4])
\]

Fortran

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

\[
\text{copy(array}(i \times N/4:i \times N/4+N/4))
\]
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.

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

```c
!$acc kernels copyin(a(1:100))
do  i=1,100
   a(i) = 0
end do
!$acc end kernels
```
IMPLIED DATA REGIONS

Explicit vs Implicit Data Regions

Explicit

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

Implicit

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

These two codes are functionally the same.
IMPLIED DATA REGIONS

Explicit vs Implicit Data Regions

Explicit

```plaintext
!$acc data copyin(a(1:100))
!$acc kernels copyin(a(1:100))
  do i=1,100
    a(i) = 0
  end do
!$acc end kernels
!$acc end data
```

Implicit

```plaintext
!$acc kernels copyin(a(1:100))
  do i=1,100
    a(i) = 0
  end do
!$acc end kernels
$acc end data
```

These two codes are functionally the same.
EXPLICIT VS. IMPLICIT DATA REGIONS

Limitation

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

Explicit

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

Implicit

```c
#pragma acc kernels copyout(a[0:100])
{
    a[i] = i;
}
#pragma acc kernels copy(a[0:100])
{
    a[i] = 2 * a[i];
}
```
**EXPLICIT VS. IMPLICIT DATA REGIONS**

**Limitation**

<table>
<thead>
<tr>
<th>Explicit</th>
<th>Implicit</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$acc data copyout(a(1:100))</td>
<td>!$acc kernels copyout(a(1:100))</td>
</tr>
<tr>
<td>!$acc kernels</td>
<td>!$acc kernels</td>
</tr>
<tr>
<td>a(i) = i</td>
<td></td>
</tr>
<tr>
<td>!$acc end kernels</td>
<td>!$acc end kernels</td>
</tr>
<tr>
<td>!$acc kernels</td>
<td></td>
</tr>
<tr>
<td>a(i) = 2 * a(i)</td>
<td></td>
</tr>
<tr>
<td>!$acc end kernels</td>
<td>!$acc end kernels</td>
</tr>
<tr>
<td>!$acc end data</td>
<td>!$acc end data</td>
</tr>
</tbody>
</table>

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.

```plaintext
#pragma acc enter data clauses
< Sequential and/or Parallel code >
#pragma acc exit data clauses
```

```plaintext
!$acc enter data clauses
< Sequential and/or Parallel code >
!$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.

```c
#pragma acc enter data clauses
  < Sequential and/or Parallel code >
#pragma acc exit data clauses
```

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

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

Basic Example

```c
#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

```plaintext
#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])
```

<table>
<thead>
<tr>
<th>Action</th>
<th>CPU MEMORY</th>
<th>device MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate A</td>
<td>A</td>
<td>C'</td>
</tr>
<tr>
<td>Copy A from CPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocate B</td>
<td>B</td>
<td>ABC</td>
</tr>
<tr>
<td>Copy B from CPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocate C</td>
<td>C</td>
<td>C'</td>
</tr>
<tr>
<td>Execute loop</td>
<td>C</td>
<td>C'</td>
</tr>
<tr>
<td>Copy C from device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deallocate C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)
```

**CPU MEMORY**

```
A B C
```

**device MEMORY**

```
A B C'
```

Action

Deallocate B from device
## UNSTRUCTURED VS STRUCTURED

With a simple code

### Unstructured

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

```c
#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

```c
#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];
    }
}
```
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

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

C/C++

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

Fortran
The data must exist on both the CPU and device for the update directive to work.
SYNCHRONIZE DATA WITH UPDATE

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.
SYNCHRONIZE DATA WITH UPDATE

Inside the `initialize` subroutine 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?
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

```c
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)
    free(v.arr);
}
```

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
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

```cpp
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);
        }
};
```
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.

```c++
void accUpdateSelf() {
    #pragma acc update self(arr[0:n])
}
void accUpdateDevice() {
    #pragma acc update device(arr[0:n])
}
```
#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);
}
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.
ADDITIONAL RESOURCES

YouTube OpenACC Introduction Series by Michael Wolfe

Introduction to Parallel Programming with OpenACC – Part 5

Follow along by downloading the code here!
BACKUP SLIDES
BASIC DATA MANAGEMENT
Moving data between the Host and Device using copy

Allocate ‘a’ on GPU 
Copy ‘a’ from CPU to GPU 
Execute Kernels 
Copy ‘a’ from GPU to CPU 
Deallocation ‘a’ from GPU

```
#pragma acc parallel loop copy(a[0:N])
for(int i = 0; i < N; i++){
    a[i] = 2 * a[i];
}
```
BASIC DATA MANAGEMENT
Moving data between the Host and Device using copy

Allocate ‘a’ on GPU
Copy ‘a’ from CPU to GPU
Execute Kernels
Copy ‘a’ from GPU to CPU
Deallocate ‘a’ from GPU

CPU MEMORY
A’

GPU MEMORY
A’