OPENACC ONLINE COURSE

Module 2 – OpenACC Data Management

Robert Searles
NVIDIA Corporation
ABOUT THIS COURSE

3 Part Introduction to OpenACC

- Module 1 – Introduction to OpenACC
- Module 2 – Data Management with OpenACC
- Module 3 – Optimizations with OpenACC

Each module will have a corresponding lab
COURSE OBJECTIVE

Enable *YOU* to accelerate *YOUR* applications with OpenACC.
MODULE 2 OUTLINE

Topics to be covered

- CPU and GPU Memories
- CUDA Unified (Managed) Memory
- OpenACC Data Management
- Lab 2
MODULE 1 REVIEW
OPENACC DEVELOPMENT CYCLE

- **Analyze** your code to determine most likely places needing parallelization or optimization.
- **Parallelize** your code by starting with the most time consuming parts and check for correctness.
- **Optimize** your code to improve observed speed-up from parallelization.
OpenACC Directives

- Manage Data Movement
- Initiate Parallel Execution
- Optimize Loop Mappings

```c
#pragma acc data copyin(a,b) copyout(c)
{
    ...
    #pragma acc parallel
    {
        #pragma acc loop gang
        for (i = 0; i < n; ++i) {
            #pragma acc loop vector
            for (j = 0; j < n; ++j) {
                c[i][j] = a[i][j] + b[i][j];
                ...
            }
        }
    }
    ...
}
```

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore
OpenACC Directives

This Module: Manage Data Movement

Last Module: Run loops in parallel

#pragma acc data copyin(a,b) copyout(c) 
{
    ...
    #pragma acc parallel
    {
        #pragma acc loop gang
        for (i = 0; i < n; ++i) {
            #pragma acc loop vector
            for (j = 0; j < n; ++j) {
                c[i][j] = a[i][j] + b[i][j];
                ...
            }
        }
    }
    ...
}
PARALLELIZE WITH OPENACC PARALLEL LOOP

```
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                 A[j-1][i] + A[j+1][i]);
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc parallel loop
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

Parallelize first loop nest, max reduction required.

Parallelize second loop.

We didn’t detail how to parallelize the loops, just which loops to parallelize.
OPENACC SPEED-UP

PGI 19.10, NVIDIA Tesla V100, IBM POWER9 22-core CPU @ 3.07GHz
CPU AND GPU MEMORIES
CPU + GPU

Physical Diagram

- CPU memory is larger, GPU memory has more bandwidth
- CPU and GPU memory are usually separate, connected by an I/O bus (traditionally PCIe)
- Any data transferred between the CPU and GPU will be handled by the I/O Bus
- The I/O Bus is relatively slow compared to memory bandwidth
- The GPU cannot perform computation until the data is within its memory
CUDA UNIFIED MEMORY
CUDA UNIFIED MEMORY
Simplified Developer Effort

Without Managed Memory

With Managed Memory

Commonly referred to as “managed memory.”

CPU and GPU memories are combined into a single, shared pool.
Handling explicit data transfers between the host and device (CPU and GPU) can be difficult.

The PGI compiler can utilize CUDA Managed Memory to defer data management.

This allows the developer to concentrate on parallelism and think about data movement as an optimization.

```
$ pgcc -fast -ta=tesla:managed -Minfo=accel main.c
```

```
$ pgfortran -fast -ta=tesla:managed -Minfo=accel main.f90
```
MANAGED MEMORY

Limitations

- The programmer will almost always be able to get better performance by manually handling data transfers.
- Memory allocation/deallocation takes longer with managed memory.
- Cannot transfer data asynchronously.
- Currently only available from PGI on NVIDIA GPUs.
SPEC ACCEL 1.2 OPENACC BENCHMARKS
OpenACC with Unified Memory vs OpenACC Data Directives
LAST MODULE WE USED UNIFIED MEMORY

Now let’s make our code run without.

Why?

- Removes reliance on PGI and NVIDIA GPUs
- Currently the data always arrives “Just Too Late”, let’s do better
BASIC DATA MANAGEMENT
BASIC DATA MANAGEMENT
Between the host and device

- The **host** is traditionally a CPU
- The **device** is some parallel accelerator
- When our target hardware is multicore, the host and device are the same, meaning that their memory is also the same
- There is no need to explicitly manage data when using a shared memory accelerator, such as the multicore target
BASIC DATA MANAGEMENT

Between the host and device

- When the target hardware is a GPU data will usually need to migrate between CPU and GPU memory
- Each array used on the GPU must be allocated on the GPU
- When data changes on the CPU or GPU the other must be updated
TRY TO BUILD WITHOUT “MANAGED”
Change –ta=tesla:managed to remove “managed”

```bash
ggcc -ta=tesla -Minfo=accel -Mcuda -lnvToolsExt laplace2d.c jacobi.c
```

`laplace2d.c`:

PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index for symbol (laplace2d.c: 47)
```
calcNext:
  47, Accelerator kernel generated
  Generating Tesla code
  48, #pragma acc loop gang /* blockIdx.x */
  Generating reduction(max:error)
  50, #pragma acc loop vector(128) /* threadIdx.x */
```

48, Accelerator restriction: size of the GPU copy of Anew,A is unknown
50, Loop is parallelizable

PGC-F-0704-Compilation aborted due to previous errors. (laplace2d.c)
```
PGC/x86-64 Linux 18.7-0: compilation aborted
```

`jacobi.c`:
DATA SHAPING
DATA CLAUSES

**copy( list )**
Allocates memory on GPU and copies data from host to GPU 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 GPU and copies data from host to GPU 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 GPU 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 GPU but does not copy.

**Principal use:** 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.

C/C++:

```
copy(array[starting_index:length])
```

Fortran:

```
copy(array(starting_index:ending_index))
```
ARRAY SHAPING (CONT.)
Multi-dimensional Array shaping

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

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

C/C++

Fortran

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

**C/C++**

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

**Fortran**

```fortran
copy(array(i*N/4:i*N/4+N/4))
```

Both of these examples copy only ¼ of the full array.
while (err > tol && iter < iter_max) {
    err = 0.0;

    #pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
    for(int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc parallel loop copyin(Anew[0:n*m]) copyout(A[0:n*m])
    for(int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
TRY TO BUILD WITHOUT "MANAGED"

Change –ta=tesla:managed to remove "managed"

ggcc -ta=tesla -Minfo=accel -Mcuda -lnvToolsExt laplace2d.c jacobi.c

laplace2d.c:

calcNext:

47, Generating copyin(A[:m*n])
    Accelerator kernel generated
    Generating Tesla code
48, #pragma acc loop gang /* blockIdx.x */
    Generating reduction(max:error)
50, #pragma acc loop vector(128) /* threadIdx.x */

47, Generating implicit copy(error)

47, Generating copyin(Anew[:m*n])

50, Loop is parallelizable

swap:

62, Generating copyin(Anew[:m*n])

63, #pragma acc loop gang /* blockIdx.x */

65, Loop is parallelizable

jacobi.c:

OpenACC  
NVIDIA
OPENACC **SPEED-UP** SLOWDOWN

![Speed-up Chart]

- **Serial**: 1.00X
- **Multicore**: 10.43X
- **V100**: 34.88X
- **V100 (Data Clauses)**: 0.58X
WHAT WENT WRONG?

- The code now has all of the information necessary to build without managed memory, but it runs much slower.
- Profiling tools are here to help!
APPLICATION PROFILE

Kernels

Data Copies
RUNTIME BREAKDOWN

Nearly all of our time is spent moving data to/from the GPU.
OPTIMIZED DATA MOVEMENT

while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++) {
      A[j][i] = Anew[j][i];
    }
    iter++;
  }

Currently we’re copying to/from the GPU for each loop, can we reuse it?
OPTIMIZE DATA MOVEMENT
OPENACC DATA DIRECTIVE

Definition

- The data directive defines a lifetime for data on the device beyond individual loops
- During the region data is essentially “owned by” the accelerator
- Data clauses express shape and data movement for the region

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

!$acc data clauses

< Sequential and/or Parallel code >

!$acc end data
```
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>C'</td>
</tr>
<tr>
<td>Allocate B on device</td>
<td>A</td>
<td>C'</td>
</tr>
<tr>
<td>Allocate C on device</td>
<td>A</td>
<td>C'</td>
</tr>
<tr>
<td>Execute loop on device</td>
<td>C</td>
<td>C'</td>
</tr>
<tr>
<td>Copy C from device to CPU</td>
<td>C'</td>
<td>B</td>
</tr>
<tr>
<td>Deallocate C from device</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Deallocate B from device</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>Deallocate A from device</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
  err=0.0;

  #pragma acc parallel loop reduction(max:err) copyin(A[0:n*m])
  for( int j = 1; j < n-1; j++ ) {
    for(int i = 1; i < m-1; i++) {


      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

  #pragma acc parallel loop copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++ ) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
REBUILD THE CODE

```
pgcc -fast -ta=tesla -Minfo=accel laplace2d_uvm.c
main:

60, Generating copy(A[:m*n])
    Generating copyin(Anew[:m*n])
64, Accelerator kernel generated
    Generating Tesla code
    64, Generating reduction(max:error)
    65, #pragma acc loop gang /* blockIdx.x */
    67, #pragma acc loop vector(128) /* threadIdx.x */
67, Loop is parallelizable
75, Accelerator kernel generated
    Generating Tesla code
    76, #pragma acc loop gang /* blockIdx.x */
    78, #pragma acc loop vector(128) /* threadIdx.x */
78, Loop is parallelizable
```

Now data movement only happens at our outer data region.
OPENACC SPEED-UP

Speed-up

- OPENACC SPEED-UP
- SERIAL
- MULTICORE
- V100
- V100 (DATA)
WHAT WE’VE LEARNED SO FAR

- CUDA Unified (Managed) Memory is a powerful porting tool
- GPU programming without managed memory often requires data shaping
- Moving data at each loop is often inefficient
- The OpenACC Data region can decouple data movement and computation
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/C++

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

Fortran

```
!$acc update self(x(1:end_index))
!$acc update device(x(1:end_index))
```
The data must exist on both the CPU and device for the update directive to work.

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

```
#pragma acc update self(B[0:N])
```
SYNCHRONIZE DATA WITH UPDATE

Sometimes data changes on the host or device inside a data region

Ending the data region and starting a new one is expensive

Instead, update the data so that the host and device data are the same

Examples: File I/O, Communication, etc.

```c
int* A=(int*) malloc(N*sizeof(int))
#pragma acc data create(A[0:N])
while( timesteps++ < numSteps )
{
    #pragma acc parallel loop
    for(int i = 0; i < N; i++){
        a[i] *= 2;
    }
    if (timestep % 100 ) {
        #pragma acc update self(A[0:N])
        checkpointAToFile(A, N);
    }
}
```
UNSTRUCTURED DATA DIRECTIVES
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

Enter data:

- **copyin** *(list)* Allocates memory on device and copies data from host to device on enter data.

- **create** *(list)* Allocates memory on device without data transfer on enter data.

Exit data:

- **copyout** *(list)* Allocates memory on device and copies data back to the host on exit 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]) delete(a,b)
```
### UNSTRUCTURED VS STRUCTURED

**With a simple code**

<table>
<thead>
<tr>
<th>Unstructured</th>
<th>Structured</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Can have multiple starting/ending points</td>
<td>- Must have explicit start/end points</td>
</tr>
<tr>
<td>- Can branch across multiple functions</td>
<td>- Must be within a single function</td>
</tr>
<tr>
<td>- Memory exists until explicitly deallocated</td>
<td>- Memory only exists within the data region</td>
</tr>
</tbody>
</table>

```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)
```

```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];
    }
}
```
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

```c++
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);
        }
};
```
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.
OPENACC RESOURCES

Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

Resources
https://www.openacc.org/resources

Success Stories
https://www.openacc.org/success-stories

FREE Compilers
PGI
Community Edition

Compilers and Tools
https://www.openacc.org/tools

Events
https://www.openacc.org/events

FREE Compilers
https://www.openacc.org/community#slack
CLOSING REMARKS
KEY CONCEPTS
In this module we discussed…

- Differences between CPU, GPU, and Unified Memories
- OpenACC Array Shaping
- OpenACC Data Clauses
- OpenACC Structured Data Region
- OpenACC Update Directive
- OpenACC Unstructured Data Directives

Next Module: Loop Optimizations