INTRODUCTION TO OPENACC

Princeton Research Computing Fall Break Bootcamp
October 17, 2023
Stéphane Ethier
ethier@princeton.edu (or ethier@pppl.gov)
3 WAYS TO ACCELERATE APPLICATIONS ON GPU

Applications

Libraries
- Easy to use
- Most Performance

Compiler Directives
- Easy to use
- Portable code
- OpenACC

Programming Languages
- Most Performance
- Most Flexibility

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
OPENACC DIRECTIVES

a directive-based parallel programming model designed for usability, performance and portability

<table>
<thead>
<tr>
<th>APPLICATIONS</th>
<th>PLATFORMS SUPPORTED</th>
<th>COMMUNITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>250+</td>
<td>NVIDIA GPU</td>
<td>~2700 Slack Members</td>
</tr>
<tr>
<td>3 out of Top 5</td>
<td>X86 CPU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>POWER CPU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sunway</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARM CPU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AMD GPU</td>
<td></td>
</tr>
</tbody>
</table>
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 vector
  for (i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
    ...
  }
}
...
```

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore
APPLY TO GPU HACKATHONS
Accelerate your code on GPUs with mentors by your side

- Over 20 events globally.
- 4 full days over 2 weeks.
- Online or in-person.
- 2 mentors per team. Up to 10 teams.
- Free to participate.
- GPU resource is provided.

www.openhackathons.org/events

check out https://www.openhackathons.org/s/technical-resources
"NEW" ALTERNATIVE: OPENMP

- OpenMP is another directive-based programming model that was developed in the late 90s for shared memory multithreading parallelism.

- It was a natural extension of OpenMP to include GPU programming. However, it was hard to implement in a 20+ year-old programming model.

- Several compiler companies (PGI, CRAY,…) got together in ~2010 and started from scratch, resulting in OpenACC.

- OpenMP is catching up though. Even Nvidia is adopting it in its NVHPC compilers.

- OpenMP might end up being the only common directive-based model for Nvidia, AMD, and Intel GPUs.

- AMD and Intel GPUs are the accelerators in the Exascale computers Frontier (OLCF -- #1 on top500) and Aurora (ALCF).
**OPENACC**

<table>
<thead>
<tr>
<th>Incremental</th>
<th>Single Source</th>
<th>Low Learning Curve</th>
</tr>
</thead>
<tbody>
<tr>
<td>§ Maintain existing sequential code</td>
<td>§ Rebuild the same code on multiple architectures</td>
<td>§ OpenACC is meant to be easy to use, and easy to learn</td>
</tr>
<tr>
<td>§ Add annotations to expose parallelism</td>
<td>§ Compiler determines how to parallelize for the desired machine</td>
<td>§ Programmer remains in familiar C, C++, or Fortran</td>
</tr>
<tr>
<td>§ After verifying correctness, annotate more of the code</td>
<td>§ Sequential code is maintained</td>
<td>§ No reason to learn low-level details of the hardware.</td>
</tr>
</tbody>
</table>

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
OPENACC

Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

```c
for(i = 0; i < N; i++)
{
    < loop code >
}
```

```
#pragma acc parallel loop
for(i = 0; i < N; i++)
{
    < loop code >
}
```

```
#pragma acc parallel loop
for(i = 0; i < N; i++)
{
    < loop code >
}
```

Begin with a working sequential code.
Parallelize it with OpenACC.
Rerun the code to verify correctness and performance.

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
OPENACC

Supported Platforms
- POWER
- Sunway
- x86 CPU
- AMD GPU
- NVIDIA GPU
- PEZY-SC

Single Source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

The compiler can ignore your OpenACC code additions, so the same code can be used for parallel or sequential execution.

```c
int main(){
    ...
    #pragma acc parallel loop
    for(int i = 0; i < N; i++)
        < loop code >
}
```
OpenACC is meant to be easy to use, and easy to learn.

The programmer remains in familiar C, C++, or Fortran.

No reason to learn low-level details of the hardware.

The programmer will give hints to the compiler.

The compiler parallelizes the code.

int main(){
    <sequential code>
    
    #pragma acc kernels
    {
        <parallel code>
    }  
}

Low Learning Curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.
OPENACC SYNTAX
A **pragma** in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.

A **directive** in Fortran is a specially formatted comment that likewise instructions the compiler in its compilation of the code and can be freely ignored.

“**acc**” informs the compiler that what will come is an OpenACC directive.

**Directives** are commands in OpenACC for altering our code.

**Clauses** are specifiers or additions to directives.
EXAMPLE CODE
LAPLACE HEAT TRANSFER

Introduction to lab code - visual

We will observe a simple simulation of heat distributing across a metal plate.

We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.
EXAMPLE: JACOBI ITERATION

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.

- Common, useful algorithm

- Example: Solve Laplace equation in 2D: \( \nabla^2 f(x, y) = 0 \)

\[
A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}
\]
while ( err > tol && iter < iter_max ) {
    err=0.0;

    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]));
        }
    }

    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
OPENACC DEVELOPMENT CYCLE

- **Analyze/profile** 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.

Profile with Nvidia Nsight Systems and Nsight Compute

https://developer.nvidia.com/nsight-systems
https://developer.nvidia.com/nsight-compute

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
PROFILING SEQUENTIAL CODE

Profile Your Code

Obtain detailed information about how the code ran.

This can include information such as:
- Total runtime
- Runtime of individual routines
- Hardware counters

Identify the portions of code that took the longest to run. We want to focus on these “hotspots” when parallelizing.

Lab Code: Laplace Heat Transfer

Total Runtime: 39.43 seconds

- swap: 19.04s
- calcNext: 21.49s
NVIDIA HPC SDK COMPILER BASICS

nvc, nvc++ and nvfortran (not to be confused with “nvcc” for CUDA!)

- Use nvc, nvc++, and nvfortran to compile for C, C++, Fortran
- The -acc flag enables building OpenACC code for a “Target Accelerator” (TA)
- -acc=multicore – Build the code to run across threads on a multicore CPU
- -gpu=cc80,managed – Build the code for Nvidia A100 GPU and manage the data movement for me. “cc70” is for the previous generation Volta V100 Ampere GPU.

```bash
$ nvc -fast -Minfo=accel -acc gpu=cc80,managed main.c
$ nvc++ -fast -Minfo=accel -acc -gpu=cc80,managed main.cpp
$ nvfortran -fast -Minfo=accel -acc -gpu=cc80,managed main.f90
```
OPENACC PARALLEL LOOP DIRECTIVE
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```c
#pragma acc parallel
{
    When encountering the `parallel` directive, the compiler will generate 1 or more parallel `gangs`, which execute redundantly.
}
```

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```c
#pragma acc parallel
{
  for(int i = 0; i < N; i++)
  {
    // Do Something
  }
}
```

This loop will be executed redundantly on each gang.
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```c
#pragma acc parallel
{
    for (int i = 0; i < N; i++)
    {
        // Do Something
    }
}
```

This means that each **gang** will execute the entire loop.
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```c
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }
    The **loop** directive informs the compiler which loops to parallelize.
}
```

The loop directive informs the compiler which loops to parallelize.

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
Parallelizing a single loop

### C/C++

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

### Fortran

```fortran
!$acc parallel
!$acc loop
    do i = 1, N
        a(i) = 0
    end do
!$acc end parallel
```

- Use a `parallel` directive to mark a region of code where you want parallel execution to occur.
- This parallel region is marked by curly braces in C/C++ or a start and end directive in Fortran.
- The `loop` directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs.
OPENACC PARALLEL DIRECTIVE

Parallelizing a single loop

- This pattern is so common that you can do all of this in a single line of code.
- In this example, the parallel loop directive applies to the next loop.
- This directive both marks the region for parallel execution and distributes the iterations of the loop.
- When applied to a loop with a data dependency, parallel loop may produce incorrect results.

C/C++

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

Fortran

```
!$acc parallel loop
do i = 1, N
    a(i) = 0
end do
```
OPENACC PARALLEL LOOP DIRECTIVE
Parallelizing many loops

- To parallelize multiple loops, each loop should be accompanied by a parallel directive.
- Each parallel loop can have different loop boundaries and loop optimizations.
- Each parallel loop can be parallelized in a different way.
- This is the recommended way to parallelize multiple loops. Attempting to parallelize multiple loops within the same parallel region may give performance issues or unexpected results.

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

#pragma acc parallel loop
for(int j = 0; j < M; j++)
    b[j] = 0;
```
### REDUCTION CLAUSE

- The **reduction** clause takes many values and "reduces" them to a single value, such as in a sum or maximum.
- Each thread calculates its part.
- The compiler will perform a final reduction to produce a **single global result** using the specified operation.

```c
for( i = 0; i < size; i++ )
    for( j = 0; j < size; j++ )
        for( k = 0; k < size; k++ )
            c[i][j] += a[i][k] * b[k][j];

for( i = 0; i < size; i++ )
    for( j = 0; j < size; j++ )
        double tmp = 0.0f;
        #pragma parallel acc loop
        reduction(+:tmp)
        for( k = 0; k < size; k++ )
            tmp += a[i][k] * b[k][j];
        c[i][j] = tmp;
```
## Reduction Clause Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition/Summation</td>
<td>reduction(+:sum)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication/Product</td>
<td>reduction(*:product)</td>
</tr>
<tr>
<td>max</td>
<td>Maximum value</td>
<td>reduction(max:max)</td>
</tr>
<tr>
<td>min</td>
<td>Minimum value</td>
<td>reduction(min:min)</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
<td>reduction(&amp;:val)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise or</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
<td>reduction(&amp;&amp;:val)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SCALARS AND PRIVATE CLAUSE

- By default, scalars are `firstprivate` when used in a parallel region and `private` when used in a kernels region.

- Except in some cases, scalars do not need to be added to a private clause. These cases may include but are not limited to:
  1. Scalars with global storage such as global variables in C/C++, Module variables in Fortran
  2. When the scalar is passed by reference to a device subroutine
  3. When the scalar is used as an rvalue after the compute region, aka “live-out”

- Note that putting scalars in a private clause may actually hurt performance!
BUILD AND RUN THE CODE
HANDS-ON EXERCISE

- Log onto adroit.princeton.edu
- cd OPENACC/C    or    OPENACC/Fortran
- module purge ; module load nvhpc/21.5
- cp -r /home/ethier/Fall_Break_2023/OPENACC.
- explore “Makefile”, “slurm_script”, and exercise.{c,f90}
- Compile exercise.{c,f90} with –acc=host and run it with “sbatch slurm_script”
- Compile with “make exercise”
- Add “acc parallel loop” to the code, recompile and run. What time are you getting?
- Add “export NV_ACC_NOTIFY=3” to the slurm script to get more info.
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++;
}
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 PCI-e)
- 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

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
CUDA UNIFIED MEMORY
CUDA UNIFIED MEMORY
Simplified Developer Effort

Without Managed Memory

With Managed Memory

CPU and GPU memories are combined into a single, shared pool

Commonly referred to as “managed memory.”
CUDA MANAGED MEMORY

Usefulness

- 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

```
$ nvc -fast -acc -gpu=cc70,managed -Minfo=accel main.c

$ nvfortran -fast -acc -gpu=cc70,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 on NVIDIA GPUs with NVIDIA HPC SDK.
TRY TO BUILD WITHOUT “MANAGED”
Change -gpu=cc70,cuda11.3,managed to remove “managed”

make exercise
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.

---

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
ARRAY SHAPING

- Sometimes the compiler needs help understanding the *shape* of an array (although *not always necessary* when transferring the whole 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
```
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

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

```c
!$acc data clauses
    < Sequential and/or Parallel code >
!$acc end data
```
HANDS-ON EXERCISE

- Add data clauses to the “acc parallel loop” directives.
- Recompile and run. What time are you getting? Is it faster?
- Have a look at the information from
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?
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, B, C on device</td>
<td>A B C</td>
<td>A B C’</td>
</tr>
<tr>
<td>Copy A from CPU to device</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Copy B from CPU to device</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>Allocate A, B, C on device</td>
<td>A B C</td>
<td>A B 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>C</td>
</tr>
<tr>
<td>Deallocate A, B, C from device</td>
<td>A B C</td>
<td>A B C’</td>
</tr>
</tbody>
</table>

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
HANDS-ON EXERCISE

- How can you use the structured data directives to keep data in device memory?
- Recompile and run. What time are you getting? Is it faster?
- Have a look at the information from
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

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
The data must exist on both the CPU and device for the update directive to work.

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

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

```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);
    }
}
```

- 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.
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.

```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 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

!$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];
}
```
#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

**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];
    }
}
```
C STRUCTS
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);
}

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)
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)
#pragma acc exit data delete(v)
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.
OPENACC RESOURCES

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

FREE Compilers
NVIDIA HPC SDK

https://www.openacc.org/resources

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

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

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

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

FREE Compilers
NVIDIA HPC SDK

https://www.openacc.org/community