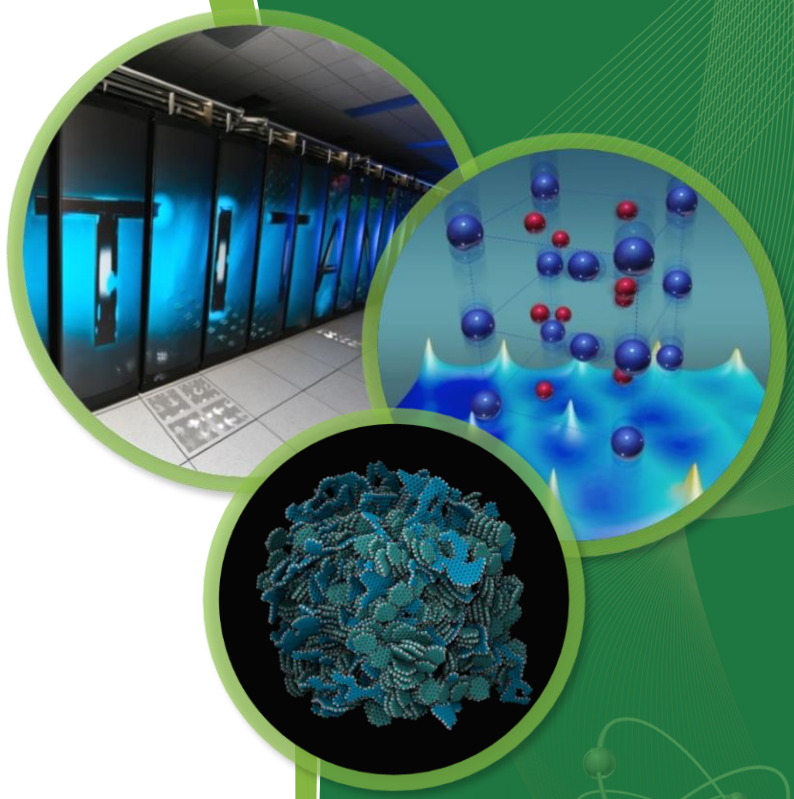
Introduction to OpenACC Directives



Matt Norman

Scientific Computing Group

Oak Ridge National Laboratory

https://mrnorman.github.io

What Is OpenACC?

- A set of directives and a library API for simpler GPU programming
- Directives are *ignored* when the compiler isn't using OpenACC
- Directives in C/C++: #pragma acc [clauses]
- Directives in Fortran: !$acc [clauses]
- A few main purposes for OpenACC directives
  1. Apply directives to loops to thread them on the GPU
  2. Manage data between the CPU main memory and GPU memory
  3. Manage asynchronicity between the CPU and the GPU
- OpenACC is intended for cross-platform portability
  - Realistically, it's geared more toward the GPU

Why Use OpenACC?

- Simpler to use than CUDA
- Source code requires fewer changes
- More portable to other architectures
- Similar to OpenMP (familiarity)
  - Easy transition to OpenMP 4.5 +
- Shields programmers somewhat from hardware complexity
- OpenACC provides some conveniences that CUDA does not

Part 1: Apply directives to loops to thread them on the GPU

Layout of a GPU

- A GPU is a collection of “Streaming Multiprocessors” (SMs)
- SMs do not coordinate with each other and are completely independent

Layout of a GPU

- A GPU is a collection of “Streaming Multiprocessors” (SMs)
- SMs do not coordinate with each other and are completely independent
- Each SM contains some local cache and a lot of cores
- Cores within an SM can synchronize, but there is no easy way to synchronize between SMs.

OpenACC kernels directive

- "!$acc kernels" is what you use when you want the compiler to do the parallelization
- I don't recommend using this
  - It often significantly underperforms, and "kernels" is very dissimilar from OpenMP 4.5 +
- Operates over a bracket in C; requires an "!$acc end kernels" in Fortran

```
#pragma acc kernels
{
  for (i=0; i < n; i++) {...}
}
```

```
!$acc kernels
do i = 1 , n
  ...
enddo
!$acc end kernels
```

- All options used with "kernels" is considered a suggestion by the compiler
- Compiler retains the right to do whatever it wants to do

OpenACC parallel directive

- The “!$acc parallel” is prescriptive rather than descriptive (like OpenMP)
- Usually coupled with the “loop” directive as “#pragma acc parallel loop”
- Each “parallel” directive is a single kernel on the GPU

```
#pragma acc parallel loop [options]
for (i=0; i < n; i++) {...}
```

```
!$acc parallel loop [options]
do i = 1 , n
  ...
enddo
```

GPU Loop Threading: Gang, Worker, and Vector

- `!$acc loop` takes optional arguments to say how to distribute the loop's work
- Options are "gang", "worker" and "vector"
  - Gang: Distribute the work across the GPU's SMs (for outer loops)
  - Vector: Distribute the work within a GPU's SM (for inner loops)
  - Worker: Also distribute work within a GPU's SM, but on a coarser scale
- Typical CUDA equivalents (this is in practice, not specified by OpenACC standard)
  - Grid = blockIdx.x ; Worker = threadIdx.y ; Vector = threadIdx.x
- If you only have two levels of parallelism, choose gang & vector

```
!$acc parallel loop gang
do k = 1 , nz
  !$acc loop worker
  do j = 1 , ny
    !$acc loop vector
    do i = 1 , nx
      ...
    enddo
  enddo
enddo
```

GPU Loop Threading: Loop Collapsing

- "Collapsing" loops means "gluing" multiple loops together into a single level of parallelism
- Loop collapsing is often more efficient than explicitly specifying "gang", "worker", and "vector
  - Often, your inner loops aren't optimally sized for vector work
  - Often, you have too many loops to fit OpenACC's three levels of parallelism
  - Often, collapsing allows needed flexibility in your loop dimension sizes

Example: Threading a loop on the GPU

- Example: Diffusion equation solved by central Finite-Differences: $\frac{\partial q}{\partial t} = \kappa\left(\frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2}\right)$
- Fortran version

```
do n = 1 , nsteps
  !$acc parallel loop collapse(2)
  do j = 1 , ny
    do i = 1 , nx
      tend(i,j) = kappa * ( (q(i+1,j)-2*q(i,j)+q(i-1,j)) / (dx*dx) + &
                            (q(i,j+1)-2*q(i,j)+q(i,j-1)) / (dy*dy) )
    enddo
  enddo
  !$acc parallel loop collapse(2)
  do j = 1 , ny
    do i = 1 , nx
      q(i,j) = q(i,j) + dt * tend(i,j)
    enddo
  enddo
enddo
```

Example: Threading a loop on the GPU

- Example: Diffusion equation solved by central Finite-Differences: $\frac{\partial q}{\partial t} = \kappa\left(\frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2}\right)$
- C version

```
for (n=0; n<nsteps; n++) {
  #pragma acc parallel loop collapse(2)
  for (j=0; j<ny; j++) {
    for (i=0; i<nx; i++) {
      tend[j][i] = kappa * ( (q[j][i+1]-2*q[j][i]+q[j][i-1]) / (dx*dx) +
                             (q[j+1][i]-2*q[j][i]+q[j-1][i]) / (dy*dy) );
    }
  }
  #pragma acc parallel loop collapse(2)
  for (j=0; j<ny; j++) {
    for (i=0; i<nx; i++) {
      q[j][i] = q[j][i] + dt * tend[j][i];
    }
  }
}
```

Race Conditions in OpenACC Programming

- OpenACC's seeming simplicity can be deceptive
- Most people are not used to exposing massive numbers of threads
- As you're forced to thread inner loops of your code, you'll likely encounter "race conditions" you're not used to dealing with
- A race condition is when multiple parallel threads access data at the same memory location in a manner that can cause wrong answers

- Usually, it's multiple threads doing a read+write of the same memory
  - This most typically happens with array reductions
  - Two arrays might read the same, stale value, update, then write incorrect values

GPU Loop Threading: Atomic Operations

- Sometimes, you must avoid threads updating the same memory location
- Example: Partial reductions

```
!$acc parallel loop collapse(3)
do k=1,nzm
  do i=1,nxp1
    do icrm = 1 , ncrms
      !$acc atomic update
      flux(icrm,k) = flux(icrm,k) + www(icrm,i,k)
    end do
  end do
end do
```

- If all threads updated “flux” at the same time, that would give an error
- With “atomic,” only one thread will do a read/write update at a time
- “Atomic” becomes less efficient, the more threads you have accessing at the same time

How Do I Know When To Use “!$acc atomic”?

Look for an array on the left-hand-side of an “=“ with fewer indices than you have loops

```
!$acc parallel loop collapse(3)
do k=1,nzm
  do i=1,nxp1
    do icrm = 1 , ncrms
      !$acc atomic update
      flux(icrm,k) = flux(icrm,k) + www(icrm,i,k)
    end do
  end do
end do
```

- Notice “flux” has two indices, but we’re threading over three loops
- You’re guaranteed to have a race condition in these circumstances

GPU Loop Threading: Global Reductions

- OpenACC does kernel-wide reductions very efficiently:

```
glob_sum = 0
!$acc parallel loop collapse(3) reduction(+:glob_sum)
do k=1,nz
  do j=1,ny
    do i=1,nx
      glob_sum = glob_sum + www(i,j,k)
    end do
  end do
end do
```

Do I Use Atomic or Reduction?

- Partial reductions (i.e., over some but not all of the loop indices) often perform better with “atomic”
- Global reductions to a scalar always perform better with reductions

```
!$acc parallel loop collapse(3)
do k=1,nzm
  do i=1,nxp1
    do icrm = 1 , ncrms
      !$acc atomic update
      flux(icrm,k) = flux(icrm,k) + www(icrm,i,k)
    end do
  end do
end do
```

```
glob_sum = 0
!$acc parallel loop collapse(3) reduction(+:glob_sum)
do k=1,nz
  do j=1,ny
    do i=1,nx
      glob_sum = glob_sum + www(i,j,k)
    end do
  end do
end do
```

GPU Loop Threading: Private Variables

- Often, every thread will need its own copy of a small array

```
!$acc parallel loop collapse(4) private(oldgrads)
do ie = nets , nete
  do k = 1 , len
    do j = 1 , np
      do i = 1 , np
        oldgrads = grads(i,j,:,k,ie)
        grads(i,j,1,k,ie) = sum(oldgrads(:)*tensorVisc(i,j,1,:,ie))
        grads(i,j,2,k,ie) = sum(oldgrads(:)*tensorVisc(i,j,2,:,ie))
      enddo
    enddo
  enddo
enddo
```

- You know to use private when an array's dimensions are not one of the parallelized loops
- You do not need to privatize scalars, only arrays

GPU Loop Threading: If Statements and Collapsing

- You cannot collapse loops that have intermittent work (even if-statements)

```
do k = 1 , nz
  if (level_active(k)) then
    do j = 1 , ny
      do i = 1 , nx
        tend(i,j) = kappa * ( (q(i+1,j)-2*q(i,j)+q(i-1,j)) / (dx*dx) + &
                              (q(i,j+1)-2*q(i,j)+q(i,j-1)) / (dy*dy) )
      enddo
    enddo
  endif
enddo
```

```
!$acc parallel loop collapse(3)
do k = 1 , nz
  do j = 1 , ny
    do i = 1 , nx
      if (level_active(k)) then
        tend(i,j) = kappa * ( (q(i+1,j)-2*q(i,j)+q(i-1,j)) / (dx*dx) + &
                              (q(i,j+1)-2*q(i,j)+q(i,j-1)) / (dy*dy) )
      endif
    enddo
  enddo
enddo
```

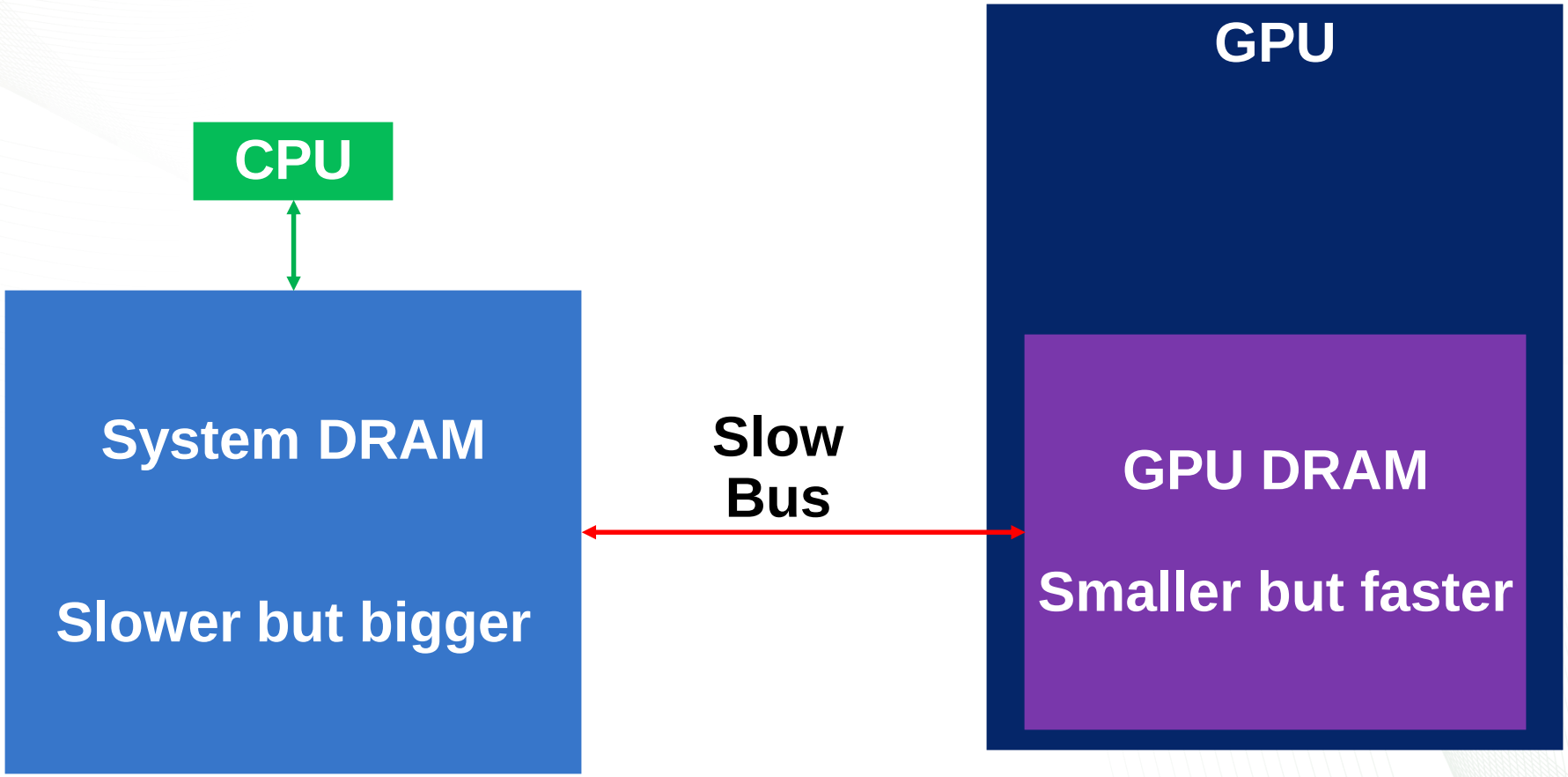
GPU Loop Threading: Function Calls

- Function calls must be declared on the GPU with the “routine” directive
- !$acc routine(name) [seq | vector | worker | gang]

```
!$acc parallel loop collapse(3)
do k = 1 , nz
  do j = 1 , ny
    do i = 1 , nx
      tend(i,j) = compute_tendency(q,i,j,dx,dy)
    enddo
  enddo
enddo
function compute_tendency(q,i,j,dx,dy)  result(tnd)
  !$acc routine(compute_tendency) seq
  implicit none
  real(8), intent(in) :: q(:,:), dx, dy
  integer, intent(in) :: i,j
  real(8)             :: tnd
  tnd = kappa * ( (q(i+1,j)-2*q(i,j)+q(i-1,j)) / (dx*dx) + &
                  (q(i,j+1)-2*q(i,j)+q(i,j-1)) / (dy*dy) )
end function compute_tendency
```

Part 2: Manage data between the CPU main memory and GPU memory

GPU & CPU Have Separate Memory Spaces



In Fortran, Compilers Generate Data Statements For You

- Use PGI's or Cray's OpenACC compiler for this feature
- PGI has a Community Edition you can use for free
- Port your kernels *first*, and then worry about optimizing data movement later
- The compiler will analyze your kernels and generate correct data statements
- It will also report to you which data statements it generated
  - PGI: -Minfo=accel
- You can use this information to write data statements yourself

- In C / C++, you must worry about data as you port the kernels

OpenACC Is Meant To *Mirror* Data Between CPU and GPU

- The typical data mode in OpenACC is for data to exist on both CPU and GPU
- You declare / allocate data on the CPU first
- Then, you create it on the device with directives and update it back and forth

IMPORTANT: Declaring Array Extent in C/C++

- In general, all arrays in C/C++ OpenACC data statements should have data bounds
  - Because the compiler doesn't know how big the array is
  - Fortran uses "array descriptors" under the hood and therefore does not need this
- However, OpenACC uses a very awkward syntax for array slicing

  #pragma acc data copyin(dat1[*beg*:***size***])

- So if you want to copy an array from index 3 to index 10, you would say:

  #pragma acc data copyin(dat1[*3*:*8*])

- If you don't specify a first index, 0 is assumed

Structured Data Statements

- All data statements are made from the perspective of the GPU
- All data statements are prepended by “present or” by default
  - **Important: If the data is already present on the GPU, the data statement is ignored!**
  - This allows you to easily nest data statements to make sure it’s handled correctly
- `!$acc data [ copy(…) | copyin (…) | copyout (…) | create (…) ]`
  …
  `!$acc end data`
  - “copyin”: If the data isn’t already present, allocate it on the GPU and copy from CPU to GPU at the beginning, then deallocate on GPU at the end
  - “copyout”: If the data isn’t already present, allocate it on the GPU at the beginning, then copy the data from GPU to CPU and deallocate it on the GPU at the end
  - “copy” = If the data isn’t already present, allocate it on the GPU and copy from CPU to GPU at the beginning, then copy the data from GPU to CPU and deallocate it on the GPU at the end
  - “create” = If the data isn’t already present, allocate it on the GPU at the beginning and deallocate it at the end
- Structured data statements cannot be done asynchronously

Structured Data Statements

- Try to “glue together” your data statements to avoid shuffling data back and forth

```
!$acc data copyin(level_active) copyout(hsum) copy(q) create(tend)
do n = 1 , nsteps
  !$acc parallel loop collapse(3)
  do k = 1 , nz
    do j = 1 , ny
      do i = 1 , nx
        if (level_active(k)) then
          tend(i,j,k) = kappa * ( (q(i+1,j,k)-2*q(i,j,k)+q(i-1,j,k)) / (dx*dx) + &
                                  (q(i,j+1,k)-2*q(i,j,k)+q(i,j-1,k)) / (dy*dy) )
          if (i==1 .and. j==1) hsum(k) = 0
        endif
      enddo
    enddo
  enddo
  !$acc parallel loop collapse(2)
  do k = 1 , nz
    do j = 1 , ny
      do i = 1 , nx
        q(i,j,k) = q(i,j,k) + dt * tend(i,j,k)
        !$acc atomic update
        hsum(k) = hsum(k) + q(i,j,k)
      enddo
    enddo
  enddo
enddo
!$acc end data
```

Intermittent Updates

- If you need to move data every time the statement is reached, whether the data exists on the GPU or not, use `!$acc update(…)`
- !$acc update host(a): Copy array “a” from the GPU to the CPU
- !$acc update device(a): Copy array “a” from the CPU to the GPU

Unstructured Data Statements

- Use unstructured data statements when:
  - You want to transfer data asynchronously (more on this later)
  - Your data flow is more complicated and cannot be done in strictly structured blocks
- `!$acc enter data [create(…)|copyin(…)]`

  …

  `!$acc exit data [delete(…)|copyout(…)]`
- exit data does not need to be in the same routine as enter data
- You do not need a 1:1 correspondence for the data in enter and exit
- You do not need an exit for every enter (though this is technically a memory leak)

Other Data Statements

- Telling the compiler data is already present (not usually needed)
  - `!$acc parallel loop` **`present(…)`**
  - This tells the compiler the data is definitely already there, so don't generate data statements
- Data statements on a per-kernel basis
  - `!$acc parallel loop` **`[copyin(…) | copyout(…) | copy(…)]`**

Managed Memory

- Managed memory makes things much easier to deal with in OpenACC
- CUDA Managed Memory will page data to and from the GPU for you when you use it on the CPU and then the GPU and vice versa
  - You cannot, however, access it on the CPU and GPU at the same time
- You have no need of data statements in this case
- PGI exposes this with the "-ta=nvidia,managed" compiler flag
  - This replaces Fortran "allocate" and C "malloc" with "cudaMallocManaged" under the hood
- **This makes C OpenACC programming much less stressful**
- cudaMallocManaged is very expensive. Thus PGI uses a "pool allocator" to make it less expensive
- Improve runtime performance by "prefetching" (allocating on the GPU immediately)
  - cudaMemPrefetchAsync

Part 3: Manage asynchronicity between the CPU and the GPU

Why Do Things Asynchronously?

- Overlap independent tasks on the CPU and GPU
  - Off-node data transfer may be overlapped with independent kernel execution
  - Run one model on the CPU and a more expensive one on the GPU
- Handle workflows with strong dependencies
  - Pipelines such as streaming large chunks of work to and from the GPU
  - Coarse-grained task dependency maps
- Reduce launch over head for many small kernels
- Hide GPU allocation costs by overlapping with kernel execution

The `async(id)` clause

- !$acc [clause] async(n)
  - Causes the given OpenACC directive to return immediately back to the CPU
  - Queues the OpenACC activity in the “n”th queue (think CUDA “stream”)
  - Anything in the “n”th queue will not start until the previous action in that queue has completed
  - “n” is called the “async id”. It is not necessarily the actual CUDA stream number
- Actions that can be performed asynchronously
  - Kernels, parallel loop, enter data, exit data, update, wait
- Synchronizing the CPU with a given OpenACC “async id”
  - `!$acc wait(n)`
    - Creates a barrier that doesn’t continue until the “n”th OpenACC queue is empty
  - `!$acc wait`
    - Causes the CPU code to wait on all OpenACC queues to finish

Hands-On Exercise

https://github.com/mrnorman/miniWeather/blob/master/petascale_institute.md