Challenges of offloading radiative heating calculations using OpenACC and OpenMP accelerator device directives in the CM2.5 FLOR Earth system model

Raymond Menzel
2016 Multicore 6 Workshop
Boulder, CO
September 2016
Atmospheric and ocean ranks are controlled through the use of "pelists", defined in the mpp layer.

(Nested) OpenMP is leveraged in each component layer through the use of blocking, with each thread pinned to a physical core.

The atmosphere is sub-cycled with a 1200 s timestep, while the ocean uses a 3600 s timestep.

Inside the atmospheric component, OpenMP is used to calculate the radiation and atmospheric dynamics/physics concurrently.

The radiation calculation is typically run every 3 hours (once per every 9 atmosphere timesteps).
  ○ Despite this, model timers indicate that the radiation accounts for ~15% of the time spent for the atmosphere.

Can accelerator directives be used to speed-up the calculation so that the radiation may be computed more often?
# Accelerator Directives Overview

<table>
<thead>
<tr>
<th>OpenACC</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>● “One directive for all architectures”.</td>
<td>● As of OpenMP 4.0, accelerator devices may be “targeted”.</td>
</tr>
<tr>
<td>● “Performance Portability” - a single code should perform well on multiple architectures.</td>
<td>● Separate directives are needed for CPUs and accelerators.</td>
</tr>
<tr>
<td>● Compiler can be used to perform analysis and parallelization.</td>
<td>● The compiler does not perform any analysis. Parallelization is the programmer’s responsibility.</td>
</tr>
</tbody>
</table>
## Accelerator Directives Compiler Support

<table>
<thead>
<tr>
<th>Compiler</th>
<th>OpenACC 2.0 Support (GPU)</th>
<th>OpenACC 2.0 Support (MIC)</th>
<th>OpenMP 4.0 Support (GPU)</th>
<th>OpenMP 4.0 Support (MIC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Version 16</td>
</tr>
<tr>
<td>PGI</td>
<td>&gt;= Version 14</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GNU</td>
<td>&gt;= Version 5</td>
<td>-</td>
<td>-</td>
<td>&gt;= Version 5</td>
</tr>
<tr>
<td>Cray</td>
<td>&gt;= Version 8.2</td>
<td>-</td>
<td>&gt;= Version 8.4</td>
<td>-</td>
</tr>
</tbody>
</table>
OpenACC Performance Considerations (1)

Toy kernel run on Titan (NVIDIA Tesla K20X (Kepler) GPU):

```!
$acc kernels
do i = 1, array_dim
   do j = 1, array_dim
      b(j,i) = 1.0 + a(j,i)
      c(j,i) = b(j,i) - a(j,i)
      b(j,i) = b(j,i) - c(j,i)
      b(j,i) = b(j,i)*b(j,i)
      c(j,i) = b(j,i)/a(j,i)
      b(j,i) = (b(j,i)/(c(j,i)*a(j,i)))*(a(j,i))
   enddo
enddo
$acc end kernels!
```

Compiled with PGI 16.5.0 with flags: 
- fast -acc -ta=tesla 

Profiled with the NVIDIA Visual Profiler.
For NVIDIA GPUs, the overhead for launching a kernel and initiating a data transfer are each of order 10 μs.

For array sizes greater than approximately 10,000 elements, the time taken to transfer a single array starts to greatly exceed the compute time for a simple kernel.

If the `async` clause is not used with the `!$acc kernels` or `!$acc parallel` directives, an implicit cuda stream synchronize (~3 - 5 μs on Titan) is executed for each data transfer and each kernel launch.
OpenMP Performance Considerations (1)

- Currently KNL systems are self-booting, and thus cannot support offloading.

- In “native” mode, a typical KNL system contains 64 physical cores, each supporting up to 4 hyper-threads (256 available OpenMP threads).

- Thread affinity can be controlled through the KMP_AFFINITY environment variable, or through the use of OpenMP runtime routines.

- Loop scheduling and SIMD vectorization can be controlled/tuned with OpenMP directives. Optimization reports can be outputted with the “-qopt-report” compiler flag.
OpenMP Performance Considerations (2)

Toy kernel run on self-booting Intel KNL (64 cores):

```c
!$omp parallel do collapse(2)
do i = 1,array_dim
    do j = 1,array_dim
        b(j,i) = 1.0 + a(j,i)
        c(j,i) = b(j,i) - a(j,i)
        b(j,i) = b(j,i) - c(j,i)
        b(j,i) = b(j,i)*b(j,i)
        c(j,i) = b(j,i)/a(j,i)
        b(j,i) = (b(j,i)/(c(j,i)*a(j,i)))*(a(j,i))
    enddo
enddo
```

Compiled with Intel 16.0.3.210 with flags:
-axMIC-AVX512 -O3 -qopenmp
Radiation Overview

Initial atmospheric state

Atmospheric dynamics/physics/chemistry
Atmosphere/land flux exchange
Land model update
Ice model update

Combine atmospheric and radiative fluxes and update the atmospheric state to the next time step

Radiation clean-up

Radiation time-varying processes

Block 1
- Aerosol driver
- Cloud driver
- Flux calculation

Block 2
- Aerosol driver
- Cloud driver
- Flux calculation

Block 3
- Aerosol driver
- Cloud driver
- Flux calculation

Blue = 1 thread
Red = dynamics/physics thread
Green = radiation thread
Isolating an Offload Target (1)

- Due to its relative cost (~45% of the total radiation time) and lack of dependency on I/O, the flux calculation was chosen as a target to be offloaded.

- To simplify the offloading process, a tool called KGEN developed by Youngsung Kim and John Dennis of NCAR was used.

- This tool allows for a routine or code section of a Fortran application to be isolated as a standalone, single process program (“kernel”) through the use of a single set of directives.

- Once the application source code is run through KGEN, actual application data may be recorded and outputted to be used with the KGEN generated kernel.
Advantages of using KGEN for offloading include:

- Removing other software dependencies, such as MPI and NetCDF.

- Capturing actual model data collected from user-specified ranks at several different user-specified model timesteps, which can be as input data for the standalone program.

- Providing bitwise and RMS error checking against actual model data to verify results.

- Allowing for easy profiling of the single process program with standard tools such as gprof.
Profile (gprof, Intel 15.0.2.164)

Flux calculation routine

Shortwave

Longwave
Offloading Challenges

- The flux calculation routine contains many children routines (> 20), but a relatively flat profile.
  - The most expensive routine takes about ~20% of the time, but the rest of the time is pretty evenly split between the rest of the routines (~1-5% each)

- In certain places, the call stack can get relatively large (~5 levels).

- Data dependencies are complex.
  - Derived types are allocated/deallocated each time the radiation is calculated.
  - Some routines take in/operate on (> 20 member) derived types and module variables.

- Many loops in each routine are parallelizable, but require refactoring.
Managing Data Transfers

- Each kernel launch typically involves several data transfers between the host and the device.

- These data transfer costs can greatly outweigh the accelerator compute costs.

- The number of data transfers can be minimized by:
  - Keeping data persistent on the device as long as possible.
  - Minimizing updates between the host and the device (preferably once at the beginning and once at the end of the calculation for each time step).
  - Eliminating dynamic data allocation/deallocation if possible.
Static-sized Module Variables

- In the model, all modules are required to have initialization/finalization routines, where module variables are allocated/deallocated and their values are set on the host.

- On initialization of the module, allocate and copy all necessary module variables to the device.
  ```
  !$acc enter data copyin(a,b,c)
  ```

- On finalization of the module, delete and deallocate all appropriate module variables from the device.
  ```
  !$acc exit data delete(a,b,c)
  ```

- Updates between the host/device can be performed, but should be minimized.
  ```
  !$acc update self(a)
  ```

- Here, module variables live on the device for the entire life of the module.
Derived Types

- Deep copy of derived types is not yet supported in OpenACC or OpenMP. We seek a solution that will work for both.

- Derived type members must individually be allocated/copied to and from the device. This can also be done with pointers.

- Interface changes are necessary for subroutines/functions that take in entire derived types, but will be run on a device.

```fortran
subroutine foo(n,atype)
  !$acc routine
  integer, intent(in) :: n
  type(bar), intent(inout) :: atype
  integer :: i
  do i = 1,n
    atype%mem1(i) = 1.0
    atype%mem2(i) = 2.0
  enddo
end subroutine foo

subroutine foo(n,mem1,mem2)
  !$acc routine
  integer, intent(in) :: n
  real, dimension(n), intent(inout) :: mem1
  real, dimension(n), intent(inout) :: mem2
  integer :: i
  do i = 1,n
    mem1(i) = 1.0
    mem2(i) = 2.0
  enddo
end subroutine foo
```
Identifying Parallel Loops (1)

- For each frequency band, heating rates and fluxes are calculated at each (lon, lat, vertical) grid point.

- Due to the way the arrays have been defined, the lat-lon (i and j) loops are typically inside the vertical loop (k).

- Each vertical “column” (i and j loop nest) is independent.

- Due to the MPI domain decomposition and atmospheric array blocking, typical i, j, and k dimension sizes are of order 15, 15, and 32, depending on the model resolution.

Example routine:

```fortran
subroutine foo(bar)
  ...
  do k = ks, ke               ! Vertical dimension
    do j = js, je              ! Latitude dimension
      do i = is, ie            ! Longitude dimension
        ...
        bar(i, j, k) = some_func(k, k+1, ...)
      enddo
    enddo
  enddo
  ...
end subroutine foo
```
Identifying Parallel Loops (2)

- Although the i-j loop nests are parallelizable, offloading them as individual accelerator kernels is not effective due to their small size of ~200 elements (~3 μs compute vs. ~ 10 μs launch overhead for an NVIDIA GPU per kernel).

- To hide the overhead, we need larger kernels that get launched less often. This can be achieved by:
  - Fusing small loops together.
  - Offloading multiple independent loops/loop nests in a single launch.
  - Reordering the loops so that the k loop is run sequentially inside the i-j loops.
  - Using multiple streams to possibly overlap two independent kernels (OpenACC only).
Exploiting Higher Level Parallelism

- In addition to the independence of the columns on a single rank, the radiation calculation performed by each atmosphere rank itself is independent.

- Like parallel I/O, data can be accumulated on a subset of “radiation ranks”, then offloaded in larger sizes to increase the computation intensity of each kernel launch.
Current Progress

- OpenACC and OpenMP directives are being systematically implemented in each routine in the longwave driver part of the flux calculation:
  
  - Routines are being refactored to increase kernel size and limit the number of kernel launches.
  
  - !$acc parallel async and !$acc loop directives are being implemented to limit overhead.
  
  - Data transfers are being pushed to the beginning and end of the flux calculation, so that they may only be performed once per timestep.
  
  - Module variables are transferred to the device upon initialization, and removed when the module is finalized.
  
  - Where possible, a second stream is being used to attempt to overlap independent kernels.
Available OpenACC Compute Directives

$\textbf{!$acc kernels}$

- Compiler analyzes all loops.
- For each parallelizable loop/loop nest, an accelerator kernel is generated.
  - This ensures that all loop/loop nest run to completion before the next one is started.
- Data is automatically created on/copied to and from the device as necessary.

$\textbf{!$acc parallel}$

- A number of gangs are launched, which redundantly execute the code in parallel.
- Loops decorated with the $\textbf{!$acc loop}$ directive are work-shared between gangs.
- Gangs cannot be synchronized.
Examples (OpenACC)

```c
!$acc kernels
do i = 1,n
   a(i) = b(i) + c(i)
enddo
do j = 1, m
   d(j) = e(j)*f(j)
enddo
!$acc end kernels
```

Here two accelerator kernels will be created, and then launched in order. Thus the first loop is guaranteed to be completed before second loop is started.

```c
!$acc parallel
!$acc loop
do i = 1,n
   a(i) = b(i) + c(i)
enddo
!$acc loop
do j = 1, m
   d(j) = e(j)*f(j)
enddo
!$acc end parallel
```

Here only one accelerator kernel will be created. There is no guarantee that the first loop is finished before the second loop starts.
Available OpenACC Data Directives

**!$acc data**

- Defines a local data region, where data can be copied to/from and allocated on the device.
- Local regions may contain any number of **!$acc kernels** and/or **!$acc parallel** regions.
- The existence of other data that has already been transferred to the device can be checked with the **present** clause.

**!$acc enter data / !$acc exit data**

- Transfers/removes specified data to/from the device.
- Does not create a local data region. The data stays on the device until it is removed by the programmer.
- Necessary to handle global and non-local data (like module variables).
Examples (OpenACC)

```plaintext
!$acc data copyout(a) copyin(b,c)  
...  
!$acc end data

When the !$acc data directive is reached, the “a” array is allocated on the device and the “b” and “c” arrays are copied. When the !$acc end data directive is reached, the value of a is copied from the device to the host, and then the a, b, and c arrays are freed on the device. The !$acc data and !$acc end data must exist in the same scope.
```

```plaintext
!$acc enter data copyin(a) create(b)  
...  
...  
!$acc exit data delete(a)  
...  
!$acc exit data delete(b)

When the !$acc enter data directive is reached, the “a” array is copied to the device and the “b” array is allocated on the device. This data will exist on the device until the !$acc exit data directive is reached. The !$acc enter data and !$acc exit data directives do not need to exist in the same scope.
```
Available OpenMP Device Compute Directives

**!$omp target**

- Code enclosed in a target region is executed by the target device.
- Data must be mapped (created, copied) to and from the device.
- A team of threads may be spawned on the device using the **!$omp parallel** directive.
- Loops decorated with the **!$omp do** directive are work-shared amongst threads in a team.

**!$omp teams**

- Inside a **!$omp target** region, a “league” of thread “teams” may be spawned.
- The master threads of each team redundantly execute the code in parallel.
- Loops decorated with the **!$omp distribute** directive are work-shared amongst the master threads of each team in the league.
- The **!$omp parallel do** directive can be used to work-share loop iterations amongst threads in a team.
Examples (OpenMP)

```c
!$omp target
!$omp parallel
!$omp do
  do i = 1,n
    a(i) = b(i) + c(i)
  enddo
!$omp do
  do j = 1, m
    d(j) = e(j)*f(j)
  enddo
!$omp end parallel
!$omp end target

!$omp target
!$omp teams num_teams(2)
!$omp distribute
  do i = 1,n
    !$omp parallel do
      do j = 1,m
        a(j,i) = b(j,i) + c(j,i)
      enddo
    enddo
!$omp end teams
!$omp end target
```
Available OpenMP Data Directives

**!$omp target data**

- Defines a local data region, where data can be mapped to/from and allocated on the device.

- Local regions may contain any number of !$omp target regions.

- The !$omp target update directive may be used to copy data between the host and target.

**!$omp declare target**

- Specifies that data/procedures are mapped to the target device.

- Does not create a local data region.

- Necessary to handle global and non-local data (like module variables).

- !$omp target enter data and !$omp target exit data directives have been added to OpenMP 4.5.
Examples (OpenMP)

The !$omp target data directive is used to allocate an array on the target and map existing arrays.

```
!$omp target data map(from: a) map(to:b,c)
...
!$omp end target data
```

When the !$omp target data directive is reached, the “a” array is allocated on the target and the “b” and “c” arrays are mapped to the target. When the !$omp end target data directive is reached, the value of a is copied from the target to the host, and then the a, b, and c arrays are freed on the target. The !$omp target data and !$omp end target data directives must exist in the same scope.

```
module foo
!$omp declare target (a)
real,dimension(10) :: a
...
end module foo
```

Here the !$omp declare target directive specifies that the array “a” should be mapped to the target. A local data region is not created.