Mixing OpenACC and OpenMP4.5

Lixiang (Eric) Luo
IBM Research @ ORNL
Two Kinds of “Mixing”

Co-existence of offloading directives
- Writing two sets of directives in the same source files
- Portability!
- Avoid boilerplate codes

Run-time interoperability
- Two runtimes interoperate in the same process
- Using libraries written in either paradigm (for example, host-side XL OpenMP + OpenACC math libraries)
- Incremental porting from one paradigm to another (and the two live together happily ever after...)
Co-existence of Offloading Directives
Simply writing both – doesn’t work in general

```fortran
!$omp parallel do private(b_start, b_end, b_size, vt)
do ib=1,Nbatch
    b_start=(ib-1)*b_size+1 ; b_end=min(ib*b_size,Nelt)
associate(a_b=>a(:,b_start:b_end),x_b=>x(:,b_start:b_end),y_b=>y(:,b_start:b_end))
!$omp target enter data map(to:a_b,x_b) map(alloc:y_b) depend(out:a_b) nowait
!$omp enter data copyin(a_b,x_b) create(y_b) async(ib)

!$omp target teams distribute private(vt) depend(in:a_b,x_b) depend(out:y_b) nowait
!$omp parallel loop gang private(vt) present(a_b,x_b,y_b) async(ib)
do ie=b_start,b_end
    !$omp parallel
    !$omp do collapse(2)
    !$acc loop vector collapse(2)
do j=1,Np ; do i=1,Np
        vt(i,j) = a(i,j,ie) * x(j,ie)
    end do ; end do
    !$omp do
    !$acc loop vector
    do i=1,Np
        y(i,ie) = sum(vt(i,:))
    end do
    !$omp end parallel
end do
!$omp target exit data map(from:y_b) depend(in:y_b) nowait
!$acc exit data copyout(y_b) async(thread_id)
end associate
end do
!$omp taskwait
!$acc wait
```

OK with XL OpenMP4.5 - OpenACC ignored
$ xlf_r -qsmp=omp -qoffload mix_test.f90

NOT OK with PGI OpenACC - OpenMP not ignored
$ pgfortran -acc -mp mix_test.f90

These device-side directives will be parsed for the host side!
Guarding Device-side Directives

```c
!omp parallel do private(b_start, b_end, b_size, vt)
  do ib=1,Nbatch
    b_start=(ib-1)*b_size+1 ; b_end=min(ib*b_size,Nelt)
    associate(a_b=>a(:,b_start:b_end),x_b=>x(:,b_start:b_end),y_b=>y(:,b_start:b_end))
    !_OMPTGT_(target enter data map(to:a_b,x_b) map(alloc:y_b) depend(out:a_b) nowait)
    !_ACCTGT_(enter data copyin(a_b,x_b) create(y_b) async(ib))
    !_OMPTGT_(target teams distribute private(vt) depend(in:a_b,x_b) depend(out:y_b) nowait)
    !_ACCTGT_(parallel loop gang private(vt) present(a_b,x_b,y_b) async(ib))
    do ie=b_start,b_end
      !_OMPTGT_(parallel)
      !_OMPTGT_(do collapse(2))
      !_ACCTGT_(loop vector collapse(2))
      do j=1,Np ; do i=1,Np
        vt(i,j) = a(i,j,ie) * x(j,ie)
      end do ; end do
      !_OMPTGT_(parallel do)
      !_ACCTGT_(loop vector)
      do i=1,Np
        y(i,ie) = sum(vt(i,:))
      end do
      !_OMPTGT_(end parallel)
    end do
    !_OMPTGT_(target exit data map(from:y_b) depend(in:y_b) nowait)
    !_ACCTGT_(exit data copyout(y_b) async(thread_id))
  end associate
end do
!_OMPTGT_(taskwait)
!_ACCTGT_(wait)
```
Using the Guarding Macros

- Use the guarding macros to guard directives in the source codes
- Host-side OpenMP directives are not guarded
- Device-side directives can be selected at compile time
- An advanced version of guarding macros can be used if macro expansion is needed inside the guarding macros

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>#ifdef <em>OL_OMP</em></td>
<td>#ifdef <em>OL_OMP</em></td>
</tr>
<tr>
<td>#define <em>OMPTGT</em>(x) $omp x</td>
<td>#define <em>OMPTGT</em>(x) _Pragma(omp #x)</td>
</tr>
<tr>
<td>#else</td>
<td>#else</td>
</tr>
<tr>
<td>#define <em>OMPTGT</em>(x) disabled</td>
<td>#define <em>OMPTGT</em>(x) {}</td>
</tr>
<tr>
<td>#endif</td>
<td>#endif</td>
</tr>
<tr>
<td>#ifdef <em>OL_ACC</em></td>
<td>#ifdef <em>OL_ACC</em></td>
</tr>
<tr>
<td>#define <em>ACCTGT</em>(x) $acc x</td>
<td>#define <em>ACCTGT</em>(x) _Pragma(acc #x)</td>
</tr>
<tr>
<td>#else</td>
<td>#else</td>
</tr>
<tr>
<td>#define <em>ACCTGT</em>(x) disabled</td>
<td>#define <em>ACCTGT</em>(x) {}</td>
</tr>
<tr>
<td>#endif</td>
<td>#endif</td>
</tr>
</tbody>
</table>
Using the Guarding Macros

- To compile with PGI OpenACC (with host-side OpenMP)
  
  $ pgfortran -acc -mp -D OL_ACC_ ...
  $ pgcc -acc -mp -D OL_ACC_ ...
  $ pgc++ -acc -mp -D OL_ACC_ ...

- To compile with XL OpenMP4.5
  
  $ xlf_r -qsmp=omp -qoffload -D OL_OMP_ ...
  $ xlc_r -qsmp=omp -qoffload -D OL_OMP_ ...
  $ xlc++_r -qsmp=omp -qoffload -D OL_OMP_ ...
OpenACC/OpenMP4.5
Run-time Interoperability
Motivations

Using libraries written with a different paradigm

• A common example: main program uses host-side XL OpenMP for CPU-only multi-threaded codes, while a dependent library uses device-side PGI OpenACC to offload GPU-optimized algorithms.

• Another example: main program uses two libraries, one written with OpenMP4.5 offloading and the other written with OpenACC offloading.

Incremental porting from one paradigm to another

• Kernels are ported one (or a small batch) at a time, while the whole program remains functional and verifiable after each increment

• If something goes wrong, bugs can be located easily

• If done properly, re-validation may not be necessary
PGI OpenACC Main + XL OpenMP Subroutine

main-pgi.F90

program main_pgi
implicit none
integer, parameter :: N=9
real*8, dimension(N) :: arr
integer :: i

interface
subroutine subomp4(arr,n)
   bind(c,name="subomp4")
   real*8, dimension(N), device :: arr
   integer :: n
end subroutine subomp4
end interface

!$acc data copy(arr)
!$acc parallel loop vector
do i=1,N; arr(i)=i; enddo
call subomp4(arr,N)
!$acc parallel loop vector
do i=1,N; arr(i)=arr(i)+10*i; enddo
!$acc end data

write(*,'(9F5.1)') arr(1:N)
end

subomp4-xl.F90

subroutine subomp4(a,n)
implicit none
real*8, dimension(n) :: a
integer :: n
integer :: i

!$omp target teams distribute parallel do is_device_ptr(a)
do i=1,n; a(i)=a(i)+0.1*i; enddo
end
How to Compile the Code?

Static linking won’t work...

- PGI is used as the OpenACC compiler and host linker (the just-shown example), and XL is used for generating statically-linked OpenMP4.5 binaries. *Not possible*: XL's OpenMP4.5 requires certain supporting binaries to be injected during the linking stage, which PGI has no knowledge of.

- XL is used as the OpenMP4.5 compiler and host linker, and PGI is used for generating statically-linked OpenACC binaries. *Functional*: however, the OpenACC codes must be compiled with the PGI's "nordc" sub-option, severely impacting programmability.
Use Dynamic Linking Instead

- XL generates a shared library “bin-xl.so”, which packaged all the necessary supporting codes required by the OpenMP4.5 runtime.
  - “bin-xl.so” only exposes an ordinary function subomp4(), which can be linked to by any host linker.
- PGI compiles the OpenACC codes, injecting supporting codes required by the OpenACC runtime, and links subomp4() dynamically.
- The OpenACC runtime has no knowledge of the XL-generated device codes.
- Most importantly: “nordc” is not required.

$ ml xl
$ xlf_r -qsmp=omp -qoffload -qpic -c subomp4-xl.F90
$ xlf_r -qsmp=omp -qoffload -qmksrobj -o bin-xl.so subomp4-xl.o
$ ml pgi
$ pgf90 -Mcuda -ta=tesla:cc70 -o run_pgi main-pgi.F90 bin-xl.so
Building and Running

To compile the code

```
$ ml xl
$ xlf_r -qsmp=omp -qoffload -qpic -c subomp4-xl.F90
$ xlf_r -qsmp=omp -qoffload -qmkshrobj -o bin-xl.so subomp4-xl.o
$ ml pgi
$ pgf90 -Mcuda -ta=tesla:cc70 -o run_pgi main-pgi.F90 bin-xl.so
```

To run the code in an interactive session

```
$ ml cuda
$ export LD_LIBRARY_PATH=./$LD_LIBRARY_PATH
$ jsrun -n1 -g1 -E LD_LIBRARY_PATH ./run_pgi
11.1 22.2 33.3 44.4 55.5 66.6 77.7 88.8 99.9
```
### XL OpenMP Main + PGI OpenACC Library

<table>
<thead>
<tr>
<th>main.c</th>
<th>shared-data.F90</th>
<th>sub_acc.F90</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;stdio.h&gt;</code></td>
<td><code>module shared_data</code></td>
<td><code>subroutine sub_acc()</code></td>
</tr>
<tr>
<td><code>#include &lt;omp.h&gt;</code></td>
<td><code>implicit none</code></td>
<td><code>use shared_data</code></td>
</tr>
<tr>
<td><code>void sub_acc();</code></td>
<td><code>integer :: mf</code></td>
<td><code>implicit none</code></td>
</tr>
<tr>
<td><code>int main(void)</code></td>
<td><code>real*8, allocatable :: a(:)</code></td>
<td><code>integer :: i mf = 16</code></td>
</tr>
<tr>
<td><code>{</code></td>
<td><code>!$acc declare create(a)</code></td>
<td><code>allocate(a(mf))</code></td>
</tr>
<tr>
<td><code>#pragma omp parallel</code></td>
<td><code>end module shared_data</code></td>
<td><code>!$acc parallel loop gang vector present(a)</code></td>
</tr>
<tr>
<td><code>printf(&quot;thread %d\n&quot;,</code></td>
<td><code>do i=1,mf ; a(i)=i ; end do</code></td>
<td><code>write(*,'(100F5.1)') a(:)</code></td>
</tr>
<tr>
<td><code>omp_get_thread_num());</code></td>
<td></td>
<td><code>end subroutine sub_acc</code></td>
</tr>
<tr>
<td><code>sub_acc();</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- This code emulates the linking of an OpenMP-enabled host C code (compiled using XL) with a math library written with PGI Fortran OpenACC.
- Note the implicit device data allocation for global data “a” in OpenACC.
- Although OpenMP codes have no offloading, the solution is similar.
Building and Running

To compile the code

```
$ ml pgi
$ pgfortran -ta=tesla:cuda9.2,cc70 -fPIC -c shared_data.F90
$ pgfortran -ta=tesla:cuda9.2,cc70 -fPIC -c sub_acc.F90
$ pgfortran -ta=tesla:cuda9.2,cc70 -shared -o bin-pgi.so sub_acc.o shared_data.o
$ ml xl
$ xlc_r -qsmp=omp -o run_xl.x main.c bin-pgi.so # no need to use -qoffload
```

To run the code in an interactive session

```
$ jsrun -n1 -c2 -g1 -E OMP_NUM_THREADS=8 -E LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH ./run_xl.x
  thread 0
  thread 6
  thread 2
  thread 1
  thread 5
  thread 3
  thread 7
  thread 4
  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0  11.0  12.0  13.0  14.0  15.0  16.0
```
Considerations on Data Interoperability

- OpenACC and OpenMP4.5 runtimes do not share the host-device pointer association mappings.
- It is possible to reconstruct the mapping using API calls:

```c
#include <omp.h>

void xlomp4_assoc(double *a_h, double *a_d, int *size) {
    omp_target_associate_ptr(a_h, a_d, *size*sizeof(double), 0, 0);
}

void xlomp4_disassoc(double *a_h) {
    omp_target_disassociate_ptr(a_h, 0);
}
```

**XL OpenMP4.5 device pointer association**

**PGI CUDA Fortran wrapper for device pointer association**

```fortran
module omp4_wrappers

Interface

subroutine xlomp4_assoc(a_h, a_d, nv)
    bind(c)
    real*8, dimension(*) :: a_h
    real*8, dimension(*), device :: a_d
    integer :: nv
end subroutine

subroutine xlomp4_disassoc(a_h)
    bind(c)
    real*8, dimension(*) :: a_h
end subroutine

end interface

end module omp4_wrappers
```
With Manual Association

```
main-pgi.F90

program main_pgi
  use omp4_wrappers
  implicit none
  integer, parameter :: N=9
  real*8, dimension(N) :: arr
  integer :: i

interface
  subroutine subomp4(arr,n) bind(c,name="subomp4")
    real*8, dimension(N), device :: arr
    integer :: n
  end subroutine subomp4
end interface

!$acc data copy(arr)
!$acc parallel loop vector
do i=1,N; arr(i)=i; enddo
!$acc end data
write(*,'(9F5.1)') arr(1:N)
end

subomp4-xl.F90

!$omp target teams distribute parallel do
do i=1,n; a(i)=a(i)+0.1*i; enddo

eend
```
Using ATS for Data Interoperability

**With explicitly data management**

The secondary paradigm can either do a manual host-device pointer association or rely on the device pointer clause. Both requires additional programming.

**With CUDA Managed Memory**

All heap data are directly interoperable, but global data and stack data remains challenging.

**With ATS (Address Translation Service) on Summit**

All data (except pure device pointers) are interoperable and cache-coherent, from both CPU and GPU POV.
With ATS...

- OpenACC/OpenMP4.5 GPU kernels now work with any address in UM
  - Use device pointer clause or XLSMPPOPTS=TARGETMEM=UIMPLICIT (OpenMP4.5)
  - If address backed by device memory: as good as an ordinary device pointer
  - If address backed by host memory: zero-copy host memory, no implicit host-device data transfer. It is as fast as pinned host memory after pinning using cudaHostRegister(), otherwise slightly slower than pinned host memory.

- Biggest bonus: OpenACC/OpenMP4.5 kernels can be written without any map clause, and retains best possible performance for all memory
  - ATS allows most of the benefits of CUDA Managed Memory, even if changing the memory allocation source codes is not an option.
  - Pinning and prefetching can be done through CUDA API calls
  - ATS is particularly attractive for writing libraries, since it imposes no prerequisites on how the memory is allocated outside the library.
XLSMPOPTS=TARGETMEM=UIMPLICIT

This works with device pointers/managed memory/ATS, but not explicitly mapped “a” (unless changing the source codes):

```c
#pragma omp target teams distribute parallel for is_device_ptr(a)
for(i=0;i<N;i++) a[i]=i;
```

This will work without source code changes:

```c
#pragma omp target teams distribute parallel for
for(i=0;i<N;i++) a[i]=i;
```

• If using device pointers/managed/ATS, set XLSMPOPTS=TARGETMEM=UIMPLICIT
• If explicitly mapped already, do nothing – it’s assumed mapped here
Some Migration Considerations

- OpenMP4.5 has a little more freedom in expressing parallelism inside teams
  - A team handles multiple iterations in a “distribute” construct.
  - Several “parallel do” constructs inside a team construct can be merged into a single “parallel” construct encompassing multiple “do” constructs.
  - Explicit thread barriers.

- OpenMP4.5 lacks automatic optimization for Fortran transformational intrinsic functions (MATMUL, DOT_PRODUCT, ...) and array assignments.
  - Use plain loops instead

- Different data dependency resolution approach

- XL OpenMP4.5 now has good optimizations using CUDA shared memory
  - More optimizations using shared memory for team-private arrays are coming

- Automatic CUDA managed and host-pinned variables in Fortran now supported in XL OpenMP4.5
  - Pointers allocated this way is interoperable.

* Some of the topics above have been covered in March Summit Training slides
Thank you!