Using HIP and GPU Libraries with OpenMP

OLCF Preparing for Frontier Training Series

Reuben D. Budiardja
Swaroop Pophale
Wael Elwasif
Suzanne Parete-Koon

Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory

The Question

GPU libraries and HIP kernels often expect **pointers** to **GPU memory addresses** as arguments.

How do we get those from OpenMP?

Introducing “`use_device_addr`” clause

Potentially common use case for this clause is to call GPU libraries / routines from host.

`use_device_addr ( var1, [ var2, … ] )`

[…] references to the list item in […] are converted into references to the corresponding storage [in the device data environment].

- item should already be *mapped*
- if not mapped, assumed to be accessible from the device (i.e. GPU)
- For C/C++, since dynamically allocated array is likely a pointer, use `use_device_ptr` instead
- In OpenMP 4.5, only `use_device_ptr` exists
  - use this with compiler supporting OpenMP 4.5 (e.g. IBM XL on Summit)

use_device_addr / use_device_ptr simple example

```fortran
call print_addr ( A )  !-- print host addr

!$OMP target enter data map ( to: A )

!$OMP target data use_device_addr ( A )
call print_addr ( A )  !-- print device addr
!$OMP end target data

call print_addr ( A )  !-- print host addr

```

```c
floart *A = (float *) malloc ( sizeof ( float ) * 10 );

printf("%p\n", A); // print A device addr

#pragma omp target enter data map ( to: A[0:10] )

#pragma omp target data use_device_ptr ( A )
{ printf("%p\n", A); } // print A device addr

printf("%p\n", A);  // print A host addr


```

Repository containing examples in this tutorial:
https://github.com/olcf/openmp-gpu-library

Illustrative Fortran Example - Main program

```fortran
program MatrixMultipy_Example

!-- Initialize matrices
allocate ( Matrix_A ( N_VALUES, N_VALUES ), &
           Matrix_B ( N_VALUES, N_VALUES ), &
           Matrix_C ( N_VALUES, N_VALUES ) )

call random_number ( Matrix_A )
call random_number ( Matrix_B )
Matrix_C = huge ( 1.0_real64 )  !-- make error more obvious

!$OMP target enter data map ( to: Matrix_A, Matrix_B, Matrix_C )

Matrix_C_Ref = matmul ( Matrix_A, Matrix_B )

call MatrixMultiply_OpenMP ( Matrix_A, Matrix_B, Matrix_C )
!$OMP target update from ( Matrix_C )

!-- [Verification and resetting of Matrix_C]


call MatrixMultiply_HIP ( Matrix_A, Matrix_B, Matrix_C )
!$OMP target update from ( Matrix_C )

!-- [Verification and resetting of Matrix_C]


call MatrixMultiply_GPU_Library ( Matrix_A, Matrix_B, Matrix_C )
!$OMP target update from ( Matrix_C )

!-- [Verification and resetting of Matrix_C]

end program MatrixMultipy_Example
```

Initializations on host

Mapping to device; OpenMP runtime creates corresponding data storage on device

Multiple ways to do matrix multiplication:

- Using Fortran intrinsic matmul() on host
- Using subroutine with OpenMP offload
- **Using subroutine with HIP kernel**
- **Using subroutine with GPU library (hipblas)**

OpenMP Offload Subroutine

```fortran
module MM_OpenMP

  use iso_fortran_env

  implicit none
  private

  public :: &
    MatrixMultiply_OpenMP

contains

  subroutine MatrixMultiply_OpenMP ( A, B, C )

    real ( real64 ), dimension ( :, : ), intent ( in ) :: &
      A, B
    real ( real64 ), dimension ( :, : ), intent ( out ) :: &
      C

    integer :: &
      iV, jV, kV

    !$OMP target teams distribute parallel do collapse ( 2 )
    do iV = 1, size ( A, dim = 1 )
      do jV = 1, size ( B, dim = 2 )
        C ( iV, jV ) = 0.0_real64
        do kV = 1, size ( A, dim = 2 )
          C ( iV, jV ) = C ( iV, jV )  +  A ( iV, kV ) * B ( kV, jV )
        end do
      end do
    end do

  end subroutine MatrixMultiply_OpenMP

end module MM_OpenMP

```

Computation if offloaded to GPU;
No data movement because of data mappings are already done previously in the main program.

Using HIP Kernel - Fortran Calling Subroutine

```fortran
#ifdef POWER_XL
#define use_device_addr use_device_ptr
#endif

module MM_HIP

  use iso_fortran_env
  use iso_c_binding

  implicit none

  interface

    subroutine MatrixMultiply_HIP_Launch ( A, B, C, nValues ) &
               bind ( C, name = 'MatrixMultiply_HIP_Launch' )
      use iso_c_binding
      implicit none
      type ( c_ptr ), value :: &
        A, B, C
      integer ( c_int ), value :: &
        nValues
    end subroutine MatrixMultiply_HIP_Launch

  end interface

contains

  subroutine MatrixMultiply_HIP ( A, B, C )

    real ( real64 ), dimension ( :, : ), intent ( in ), target :: &
      A, B
    real ( real64 ), dimension ( :, : ), intent ( inout ), target :: &
      C

    !$OMP target data use_device_addr ( A, B, C )
    call MatrixMultiply_HIP_Launch &
           ( c_loc ( A ), c_loc ( B ), c_loc ( C ), size ( A, dim = 1 ) )
    !$OMP end target data

  end subroutine MatrixMultiply_HIP

end module MM_HIP
```

Preprocessor replaces “`use_device_addr`” to “`use_device_ptr`” for IBM XL compiler on Summit since it provides OpenMP 4.5.

Interface for the C helper subroutine (i.e. function returning void) using the Fortran - C interoperability standard.
`c_ptr` points to GPU memory addresses.

`use_device_addr` tells OpenMP to use the corresponding addresses on device. `c_loc()` returns the c_ptr type compatible to `void *` in C.

Using HIP Kernel - C & HIP Subroutines

```
// Include and headers ellided

#ifdef __cplusplus
extern "C"
  {
#endif
void MatrixMultiply_HIP_Launch
       ( double *d_A, double *d_B, double *d_C, int nV );
#ifdef __cplusplus
  }
#endif

__global__ void matrix_multiply(double *a, double *b, double *c, int N)
{
    int row    = blockDim.x * blockIdx.x + threadIdx.x;
    int column = blockDim.y * blockIdx.y + threadIdx.y;
    if (row < N && column < N)
    {
        double element = 0.0;
        for(int i=0; i<N; i++){
          element += a[row + N * i] * b[i + N * column];
        }
        c[row + N * column] = element;
    }
}

void MatrixMultiply_HIP_Launch  ( double *d_A, double *d_B, double *d_C, int nV )
  {
  dim3 threads_per_block ( 16, 16, 1 );
  dim3 blocks_in_grid ( ceil( float(nV) / threads_per_block.x ),
                        ceil( float(nV) / threads_per_block.y ), 1 );

  // Launch kernel
  double start = omp_get_wtime ( );
  hipLaunchKernelGGL ( matrix_multiply, blocks_in_grid, threads_per_block, 0,
                      0, d_A, d_B, d_C, nV );
  hipDeviceSynchronize ( );
  }

```

Create un-mangled C symbol available for linkage with the corresponding Fortran interface.

HIP / CUDA kernel for matrix multiplication, expecting the pointers to GPU memory addresses as argument.

C helper function to be called from Fortran.

Using GPU Libraries (hipblas / cublas) - Fortran Interfaces

```fortran
#ifdef POWER_XL
#define use_device_addr use_device_ptr
#define __mm_lib_create__  'cublasCreate_v2'
#define __mm_lib_dgemm__   'cublasDgemm_v2'
#define __mm_lib_destroy__ 'cublasDestroy_v2'
...
#endif

#ifdef Cray_CCE
#define __mm_lib_create__  'hipblasCreate'
#define __mm_lib_dgemm__   'hipblasDgemm'
#define __mm_lib_destroy__ 'hipblasDestroy'
...
#endif

module MM_GPU_Library
  use iso_fortran_env
  use iso_c_binding

  implicit none

  enum, bind ( c )  !-- From cublas_api.h
    enumerator :: BLAS_OP_N = __blas_op_n__
    ...
  end enum

  interface

    function MM_LibCreate ( Handle ) &
                bind ( C, name = __mm_lib_create__ ) result ( Status )
      use iso_c_binding
      implicit none
      type ( c_ptr ) :: &
        Handle
      ...
    end function MM_LibCreate

    function MM_LibDgemm ( Handle, Trans_A, Trans_B, M, N, K, Alpha, &
                           d_A, LDA, d_B, LDB, Beta, d_C, LDC ) &
               bind ( c, name = __mm_lib_dgemm__ ) result ( Status )
      use iso_c_binding
      implicit none
      type ( c_ptr ), value :: &
        Handle
     ...
    end function MM_LibDgemm
```

Preprocessor to switch between HIP or CUDA version based on defined macros (set in Makefile)

Note: On Frontier / Crusher, these Fortran interfaces are provided by hipfort:

- `module load hipfort` on terminal
- `use hipfort` in Fortran

https://github.com/ROCmSoftwarePlatform/hipfort

Interfaces to the hipblas / cublas functions; name replacements are done via preprocessor above.

Using GPU Libraries (hipblas / cublas) - Fortran Call

```fortran
subroutine MatrixMultiply_GPU_Library ( A, B, C )

  real ( real64 ), dimension ( :, : ), intent ( in ), target :: &
    A, B
  real ( real64 ), dimension ( :, : ), intent ( inout ), target :: &
    C

  integer ( c_int ) :: &
    nV, &
    Status
  real ( c_double ) :: &
    Alpha = 1.0_c_double, &
    Beta  = 0.0_c_double
  type ( c_ptr ) :: &
    Handle

  Status = MM_LibCreate ( Handle )

  nV = size ( A, dim = 1 )
  !$OMP target data use_device_addr ( A, B, C )
  Status = MM_LibDgemm ( Handle, BLAS_OP_N, BLAS_OP_N, nV, nV, nV, &
                         Alpha, c_loc ( A ), nV, c_loc ( B ), nV, &
                         Beta, c_loc ( C ), nV )
  !$OMP end target data

  Status = MM_LibDestroy ( Handle )

end subroutine MatrixMultiply_GPU_Library
```

Calling hibblas / cublas

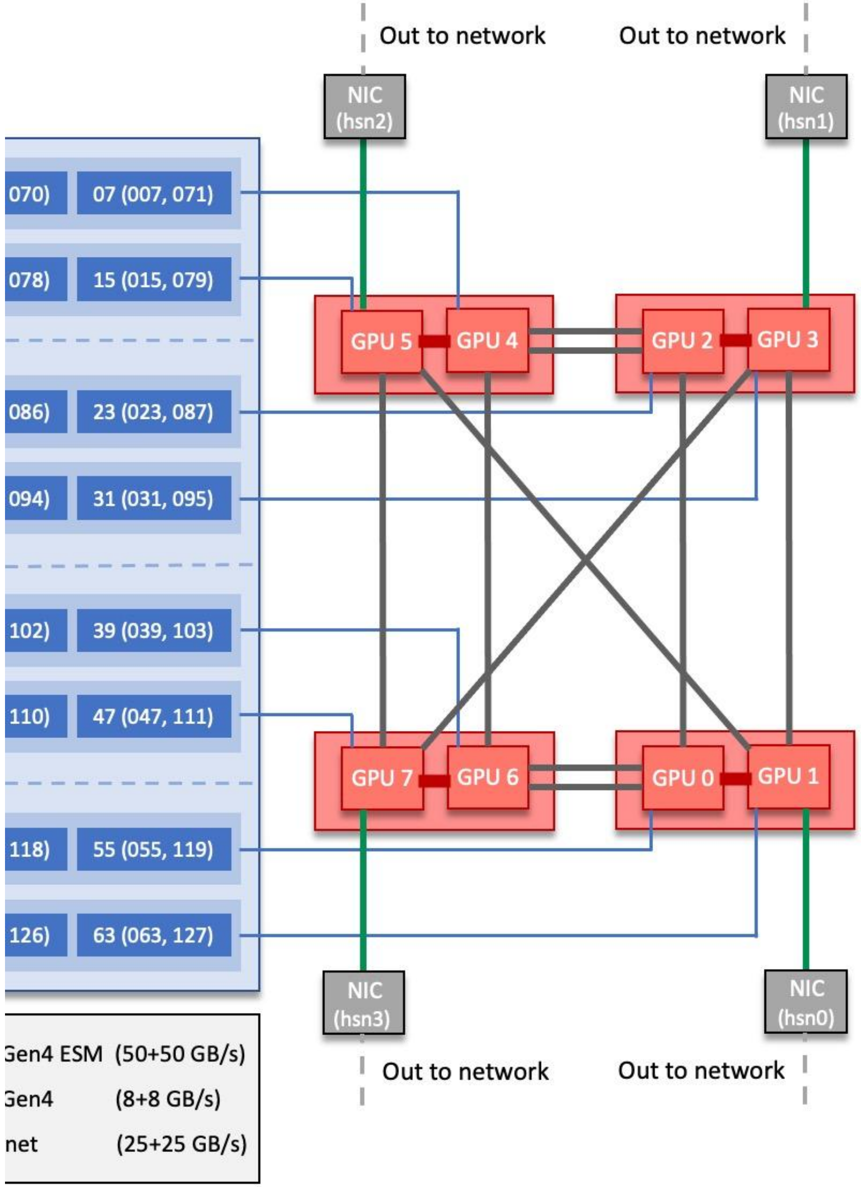
GPU-Aware MPI

GPU-aware MPI library allows GPU memory addresses to be used as arguments to MPI subroutine calls:

- avoids staging (i.e. copying) data to host first
- allows optimized / faster GPU-to-GPU communications

In the next few slides we look at examples on how to do this with OpenMP-managed data.

Frontier Node Diagram (partial)



Calling GPU-aware MPI routine

```
program MPI_Send_Recv_Example

  use mpi

  implicit none

  real ( real64 ), allocatable, dimension ( : ) :: &
    Buffer
  ! [other type definitions]

  ...

  !$OMP target data use_device_addr ( Buffer )
  call MPI_SEND &
         ( Buffer, nV, MPI_DOUBLE_PRECISION, 1, 0, MPI_COMM_WORLD, &
           Error )
  !$OMP end target data

  ...

end program MPI_Send_Recv_Example
```

Bonus: "`omp_target_associate_ptr`" routine*

`omp_target_associate_ptr(host_ptr, device_ptr, size, device_offset, device_num)`

… maps a device pointer to a host pointer.

This routine tells OpenMP to use the `device_ptr` when `host_ptr` appear in target region and subsequent map clause.

- `device_ptr` may be returned from runtime routine (e.g. `omp_target_alloc()`, `hipMalloc() / cudaMalloc()`)
- OpenMP will not create its own device storage for the host_ptr (i.e. host variables)

**Note: omp_target_associate_ptr() is not needed for GPU-aware MPI, but useful to know.*

Calling GPU-aware MPI routine (2)

```fortran
program MPI_Send_Recv_Example
  use mpi
  use omp_lib
  use iso_c_binding
  use iso_fortran_env
  implicit none

  real ( real64 ), allocatable, dimension ( : ), target :: &
    Buffer
  real ( real64 ), dimension ( : ), pointer :: &
    F_Buffer
  type ( c_ptr ) :: &
    D_Buffer
  !-- [other type definitions]

  allocate ( Buffer ( nV ) )

  !-- allocate memory on GPU, then associate it with host variable
  associate ( Size => c_sizeof ( 1.0_c_double ) * nV )
  D_Buffer = omp_target_alloc ( c_sizeof ( 1.0_c_double ) * nV, 0 )
  Error = omp_target_associate_ptr ( c_loc ( Buffer ), D_Buffer, Size, &
              0_c_size_t, 0 )
  end associate

  !-- [ MPI Initializations and set values]

  if ( MPI_Rank == 0 ) then

    !$OMP target
    Buffer = 2.0_Real64  !-- Set value on device
    !$OMP end target

    call c_f_pointer ( D_Buffer, F_Buffer, [ nV ] )
    call MPI_SEND &
           ( F_Buffer, nV, MPI_DOUBLE_PRECISION, 1, 0, MPI_COMM_WORLD, &
             Error )
  end if
```

`Buffer`: host variable
`D_Fuffer`: device_ptr to a GPU address
`F_Buffer`: Fortran pointer on the host

`Buffer` is allocated; `D_Buffer` points to GPU-memory allocated to the same size. Both are associated with `omp_target_associate_ptr()`

Values are set on addressed pointed by `D_Buffer`

`D_Buffer` is dereferenced to `F_Buffer`, which is now a rank-2 real arrays backed by GPU-memory addressed. `F_Buffer` can be passed to GPU-aware MPI routines.

Building and Running Matrix Multiplication Examples

On Summit

```
module load xl
module load cuda
make clean
make MACHINE=POWER_XL

jsrun -n 1 -g 1 -c 1 \
./MatrixMultiply_POWER_XL

     OpenMP Verification :  PASSED
        HIP Verification :  PASSED
GPU Library Verification :  PASSED
```

On Crusher / Frontier

```
module load PrgEnv-cray
module load craype-accel-amd-gfx90a
module load rocm
make clean
make MACHINE=Cray_CCE

srun -n1 ./MatrixMultiply_Cray_CCE

     OpenMP Verification :  PASSED
        HIP Verification :  PASSED
GPU Library Verification :  PASSED
```

Repository containing examples in this tutorial:
https://github.com/olcf/openmp-gpu-library

Conclusion

- We learned how to use GPU libraries with OpenMP code
  - use_device_addr clause to get the corresponding device address of a host variable
  - omp_target_associate_ptr to map a generic device pointer to a host variable for OpenMP runtime
- Complete examples code with README to run on Summit and Crusher / Frontier is available at
  [REPO INFO]
- Questions & comments: reubendb@ornl.gov