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

ORNL is managed by UT-Battelle LLC for the US Department of Energy
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

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)

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

```c
use_device_addr
```

```c
use_device_ptr
```

Repository containing examples in this tutorial:
https://github.com/olcf/openmp-gpu-library
Illustrative Fortran Example - Main program

```
program MatrixMultiply_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_real164 ) !-- 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 MatrixMultiply_Example
```
OpenMP Offload Subroutine

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
#define use_device_addr use_device_ptr

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

```c
// include and headers elided
#define __cplusplus
extern "C"
{
#endif
void MatrixMultiply_HIP_Launch
{ double *d_A, double *d_B, double *d_C, int nV);
#endif
#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 ( );
}
```
Using GPU Libraries (hipblas / cublas) - Fortran Interfaces

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
1 subroutine MatrixMultiply_GPU.Library ( A, B, C )
2   real ( real64 ), dimension ( :, : ), intent ( in ), target :: &
3     A, B
4   real ( real64 ), dimension ( :, : ), intent ( inout ), target :: &
5     C
6
7   integer ( c_int ) :: &
8     nV, &
9     Status
10   real ( c_double ) :: &
11     Alpha = 1.0 c_double, &
12     Beta = 0.0 c_double
13   type ( c_ptr ) :: &
14     Handle
15
16   Status = MM_LibCreate ( Handle )
17
18   nV = size ( A, dim = 1 )
19
20   !$OMP target data use device addr ( A, B, C )
21   Status = MM_LibDgemm ( Handle, BLAS_OP_N, BLAS_OP_N, nV, nV, nV, &
22      Alpha, c_loc ( A ), nV, c_loc ( B ), nV, &
23      Beta, c_loc ( C ), nV )
24   !$OMP end target data
25
26   Status = MM_LibDestroy ( Handle )
27
28 end subroutine MatrixMultiply_GPU.Library
```

Calling hipblas / 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.
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)

Buffer: host variable
D_Buffer: 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 \n./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