Using CUDA C , CUDA Fortran, and OpenCL on a Cray XK6

Jeff Larkin
larkin@cray.com

At the end of this talk you should be able to

- Build a CUDA C code
- Build a CUDA Fortran code
- Build an OpenCL code
- Share data between CUDA and libsci_acc
- Share data between OpenACC and CUDA
- Share data between OpenACC and libsci_acc

CUDA/OpenCL PE integration

- As you would expect, much of the complexity of using CUDA/OpenCL on an XK6 has been simplified via compiler wrappers.
- Loading the cuda module
  - Adds nvcc to the path
  - Adds CUDA/OpenCL includes automatically
  - Adds -lcuda automatically
  - Changes to dynamic linking
  - Does not automatically link in -lOpenCL

CUDA/OpenCL PE integration

- Loading the xtpe-accel-nvidia20 module
  - Automatically loads the cuda and libsci_acc modules
  - Enables OpenACC directives in CCE
  - Turns on dynamic linking

CUDA/OpenCL PE integration

- nvcc does not know about MPI headers
  - Simplest solution: isolate CUDA C and MPI codes into separate files
  - More Complicated solution: explicitly include the MPI include directory in the nvcc compile
- Building a .cu file enables C++ name mangling, so
  - C codes will need to be built with the CC compiler or…
  - Add extern “C” to continue using cc compiler

Some Gotchas

- The module versions on chester have been temporarily locked at versions that are not current.
  - Sometimes you will need to swap several modules (even if they're already loaded) to get things to build and link properly.
  - I've coded the Makefiles in the examples to help you with this when they can.
  - These problems will be fixed on the final machine

Code Samples for this talk

- Please copy /ccs/proj/trn001/cray/titan_workshop_examples.tgz

- Please hang on to these examples and slides to refer to when trying to build your codes.

CUDA FOR C

CUDA for C - What?

- CUDA C is a programming model that has been created and is supported by Nvidia.
- It consists of both library calls and language extensions.
  - Only Nvidia's nvcc compiler understands the language extensions
- Lots of tutorials and examples exist online
- Requires explicitly rewriting important parts of your code to
  - Manage accelerator memory
  - Copy data between CPU and accelerator
  - Execute on the accelerator

CUDA C (serial)

- The Plan:
  - Write a CUDA C kernel and a C (or Fortran) main program
  - Build and link with nvcc
  - Launch executable with aprun
- The Code: example1_serial/scaleitC.cu
- Supported PEs: Any
  - Works best with GNU or Cray (with -hgnu flag)

CUDA C (MPI)

- The Plan:
  - Write a CUDA C kernel and a launcher function in a .cu file containing no MPI
  - Write a C (or Fortran) main program with MPI
  - Build .cu nvcc, rest with cc (or ftn)
  - Link via cc (or ftn)
  - Launch executable with aprun
- The Code: example2_mpi/scaleitC*
- Supported PEs: Any
  - Works best with GNU or Cray (with -hgnu flag)
- Gotchas
  - nvcc uses C++ name mangling unless `extern "C"` is used.
  - If CUDA and MPI must exist in the same file, it's necessary to point nvcc to the MPI include directory

CUDA FORTRAN

CUDA Fortran - What?

- CUDA Fortran is a parallel to CUDA for C created by PGI and Nvidia and supported by PGI.
- It is a mixture of library calls and Fortran extensions to support accelerators.
- Requires explicitly rewriting important parts of your code to
  - Manage accelerator memory
  - Copy data between CPU and accelerator
  - Execute on the accelerator

CUDA Fortran (serial)

- The Plan
  - Create a Fortran module containing CUDA kernel and data
  - Create Fortran main, which calls launcher function from above module
  - Build and Link with ftn
  - Run with aprun
- The Code: example1_serial/scaleitF.F90
- Supported PEs: PGI Only
- Gotchas
  - CUDA Fortran requires the use of Fortran modules, if you have pure F77 code, it will need to be updated to F90

CUDA Fortran (mpi)

- The Plan
  - Create a Fortran module containing CUDA kernel and data
  - Create Fortran main, which calls launcher function from above module
  - Build and Link with ftn
  - Run with aprun
- The Code: example2_mpi/scaleitF.F90
- Supported PEs: PGI Only
- Gotchas
  - CUDA Fortran requires the use of Fortran modules, if you have a pure F77 code, it will need to be updated to F90

See example3/

BUILDING A PARALLEL OPENCL CODE

OpenCL - What?

- OpenCL is a set of libraries and C language extensions for generic parallel programming over a variety of devices.
- Industry standard maintained by Kronos Group and supported by multiple vendors.
- Functionally similar to low-level CUDA driver API.
- Requires explicitly rewriting important parts of your code to
  - Manage accelerator memory
  - Copy data between CPU and accelerator
  - Execute on the accelerator

OpenCL

- The Plan:
  - Write an OpenCL kernel and a launcher function in a .c
  - Write a C (or Fortran) main program with MPI
  - Build with cc (and maybe ftn)
  - Link via cc (or ftn) adding -lOpenCL
  - Launch executable with aprun
- The Code: example3/
- Supported PEs: GNU

SHARING DATA BETWEEN CUDA AND LIBSCI

LibSci and CUDA for C

- What: Part of the code relies on LibSci routines and part has been written in CUDA
- The Plan:
  - Build and use CUDA for C as before
  - Use libsci_acc's expert interface to call device kernels with your existing device arrays.
- The Code: example4_cudaC_libsci/
- Supported PEs: GNU & Cray (with -hgnu)

Libsci + CUDA for C

- Use cudaMalloc and cudaFree to manage the device memory
- Use cublasSetMatrix and cublasGetMatrix to copy to/from the device
- Use dgetrf_acc_ with your device pointers to run dgetrf on the device

```
/*  Copy A to the device                          */
cudaMalloc( &d_A, sizeof(double)*lda*M);
cublasSetMatrix( M, N, sizeof(double), A2,
                 lda, d_A, lda);

/* Calling the accelerator API of dgetrf */
dgetrf_acc_( &M, &N, d_A, &lda, ipiv, &info);

/*  Copy A in the device back to the host */
cublasGetMatrix( M, N, sizeof(double), d_A,
                 lda, A, lda);
cudaFree( d_A );
```

LibSci and CUDA Fortran

- What: Part of the code relies on LibSci routines and part has been written in CUDA Fortran
- The Plan:
  - Build and use CUDA Fortran as before
  - Use libsci_acc's expert interface to call device kernels with your existing device arrays.
- The Code: example5_cudaF_libsci/
- Supported PEs: PGI

OpenACC - Fortran

- Use CUDA Fortran to declare and manage device arrays.
- Call LibSCI expert interface to launch kernel on device with your data.

```
! allocatable device arrays
real, device, allocatable, dimension(:,:) ::
   Adev,Bdev,Cdev

! Start data xfer-inclusive timer and allocate
   the device arrays using
! F90 ALLOCATE

allocate( Adev(N,M), Bdev(M,L), Cdev(N,L) )

! Copy A and B to the device using F90 array
   assignments
Adev = A(1:N,1:M)
Bdev = B(1:M,1:L)

! Call LibSCI accelerator Kernel
call sgemm_acc ('N', 'N', N, L, M, 1.0, Adev,
                 N, Bdev, M, 0.0, Cdev, N)

! Ensure Kernel has run
r = cudathreadsynchronize()

! Copy data back from device and deallocate
C(1:N,1:L) = Cdev
deallocate( Adev, Bdev, Cdev )
```

SHARING DATA BETWEEN OPENACC AND CUDA FOR C

OpenACC & CUDA C

- The Plan
  - Write a CUDA C Kernel and a Launcher function that accepts device pointers.
  - Write a C or Fortan main that uses OpenACC directives to manage device arrays
  - Use `acc host_data` pragma/directive to pass device pointer to launcher
  - Build .cu with nvcc and rest per usual
- The Code: example6_openacc_cuda/
- Supported PEs: Cray

OpenACC C-main

- Notice that there is no need to create device pointers
- Use acc data region to allocate device arrays and handle data movement
- Use acc parallel loop to populate device array.
- Use acc host_data region to pass a device pointer for array

```
/* Allocate Array On Host */
  a = (double*)malloc(n*sizeof(double));

/* Allocate device array a. Copy data both to
   and from device. */
#pragma acc data copyout(a[0:n])
  {
#pragma acc parallel loop
    for(i=0; i<n; i++)
    {
      a[i] = i+1;
    }

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* Use device array when calling
   scaleit_launcher */
#pragma acc host_data use_device(a)
    {
      ierr = scaleit_launcher_(a, &n, &rank);
    }
  }
```

OpenACC Fortran-main

- Notice that there is no need to create device pointers
- Use acc data region to allocate device arrays and handle data movement
- Use acc parallel loop to populate device array.
- Use acc host_data region to pass a device pointer for array

```
integer,parameter :: n=16384
real(8) :: a(n)

!$acc data copy(a)
!$acc parallel loop
do i=1,n
  a(i) = i
enddo
!$acc end parallel loop

!$acc host_data use_device(a)
ierr = scaleit_launcher(a, n, rank)
!$acc end host_data
!$acc end data
```

SHARING DATA BETWEEN OPENACC AND LIBSCI

OpenACC and LibSCI

- The Plan:
  - Use OpenACC to manage your data
  - Possible use OpenACC for certain regions of the code
  - Use LibSCI's expert interface to call device routines
- The Code: example7_openacc_libsci
- Supported PEs: Cray

OpenACC with LibSCI - C

- OpenACC data region used to allocate device arrays for A, B, and C and copy data to/from the device.

```
#pragma acc data
   copyin(a[0:lda*k],b[0:n*ldb])
   copy(c[0:ldc*n])
    {
#pragma acc host_data use_device(a,b,c)
       {

   dgemm_acc('n','n',m,n,k,alpha,a,lda,b
   ,ldb,beta,c,ldc);
       }
    }
```

OpenACC with LibSCI - Fortran

- OpenACC data region used to allocate device arrays for A, B, and C and copy data to/from the device.

```
!$acc data copy(a,b,c)
!$acc host_data use_device(a,b,c)
      Call
   dgemm_acc('n','n',m,n,k,alpha,a,lda,b
   ,ldb,beta,c,ldc)
!$acc end host_data
!$acc end data
```

PE Support Summary

	CUDA for C	CUDA Fortran	LibSci_acc	OpenAcc	OpenCL
PrgEnv-cray	(yellow)	(red)	(green)	(green)	(yellow)
PrgEnv-pgi	(yellow)	(green)	(green)	(yellow)	(green)
PrgEnv-gnu	(green)	(red)	(green)	(red)	(green)

Full Support	(green)
Limited/Forthcoming Support	(yellow)
Currently No Support	(red)