Hewlett Packard Enterprise

# Cray Scientific Libraries

## Comprehensive General LUMI Course

April 23–26, 2024

# Approaches to Accelerate Applications

## Accelerated Libraries

- The easiest solution, just link the library to your application without in-depth knowledge of GPU programming
- Many libraries are optimized by GPU vendors, eg. algebra libraries

## Directive based methods

- Add acceleration to your existing code (C, C++, Fortran)
- Can reach good performance with somehow minimal code changes
- OpenACC, OpenMP

## Programming Languages

- Maximum flexibility, require in-depth knowledge of GPU programming and code rewriting (especially for Fortran)
- Kokkos, RAJA, CUDA, HIP, OpenCL, SYCL

# What are libraries for?

- Building blocks for writing scientific applications
- Historically – allowed the first forms of code re-use
- Later – became ways of running optimized code
- Today the complexity of the hardware is very high
- The Cray PE insulates users from this complexity
  - Cray module environment
  - CCE
  - Performance tools
  - Tuned MPI libraries (+PGAS)
  - Optimized Scientific libraries

Cray Scientific Libraries are designed to provide the maximum possible performance from Cray systems with minimum effort

# Cray Scientific and Maths Libraries (CSML)

- LibSci
  - BLAS (Basic Linear Algebra Subroutines)
  - BLACS (Basic Linear Algebra Communication Subprograms)
  - CBLAS (wrappers providing a C interface to the FORTRAN BLAS library)
  - IRT (Iterative Refinement Toolkit)
  - LAPACK (Linear Algebra Routines)
  - LAPACKE (C interfaces to LAPACK Routines)
  - ScaLAPACK (Scalable LAPACK)
- LibSci_ACC
  - Subset of GPU-optimized GPU routines from LibSci
- FFTW3
  - Fastest Fourier Transforms in the West, release 3
- Data libraries
  - NetCDF and HDF5

# What can we expect from the scientific libraries

1. Node performance
   - Highly tuned routines at the low-level (e.g. BLAS)

2. Network performance
   - Optimized for network performance
   - Overlap between communication and computation
   - Use the best available low-level mechanism
   - Use adaptive parallel algorithms

3. Highly adaptive software
   - Use auto-tuning and adaptation to give the user the known best (or very good) codes at runtime

4. Productivity features
   - Simple interfaces into complex software
   - Support for a range of different compilers
   - Optimized for a variety of hardware targets including recent AMD EPYC targets

# Libraries Usage

- LibSci
  - `module load cray-libsci`
    - CPU routines
  - `module load cray-libsci_acc`
    - GPU routines, requires `cray-libsci` to be loaded first
- FFTW
  - `module load cray-fftw`
- NetCDF, HDF5
  - `module load cray-hdf5 # or cray-hdf5-parallel`
  - `module load cray-netcdf # or cray-parallel-netcdf`
    - Requires `cray-hdf5` to be loaded first
- As usual, the compiler wrappers will do the linking
- Available for all PrgEnv's

# Check you are using the correct library

- Add options to the linker to make sure you have the correct library module loaded.
- `-Wl` adds a command to the linker from the driver
- You can ask for the linker to tell you where an object was resolved from using the –y option.
  - E.g. `–Wl,-ydgemm_`

```
/opt/cray/pe/libsci/22.12.1.1/CRAY/9.0/x86_64/lib/libsci_cray_mpi.so: reference to dgemm_
/opt/cray/pe/libsci/22.12.1.1/CRAY/9.0/x86_64/lib/libsci_cray.so: shared definition of dgemm_
```

- In the case of dynamic linking you can also use the command `ldd` on your executable

# Threading and LibSci

LibSci is compatible with OpenMP

- Control the number of threads to be used in your program using OMP_NUM_THREADS
- e.g., in job script **`export OMP_NUM_THREADS=16`**
- Then run leaving space for threads
  **`-n1 --cpus-per-task=16`**

What behavior you get from the library depends on your code

1. No threading in code
   - The BLAS call will use OMP_NUM_THREADS threads
2. Threaded code, outside parallel regions
   - The BLAS call will use OMP_NUM_THREADS threads
3. Threaded code, inside parallel regions
   - The BLAS call will use a single thread

# Limiting threading in LibSci

There are other ways of limiting thread count aside from using OMP_NUM_THREADS.
In an OpenMP app one could do the following:

```
omp_set_num_threads(nthreads_sci);
libsci_call(); // will use available omp threads set
omp_set_num_threads(nthreads_original);
```

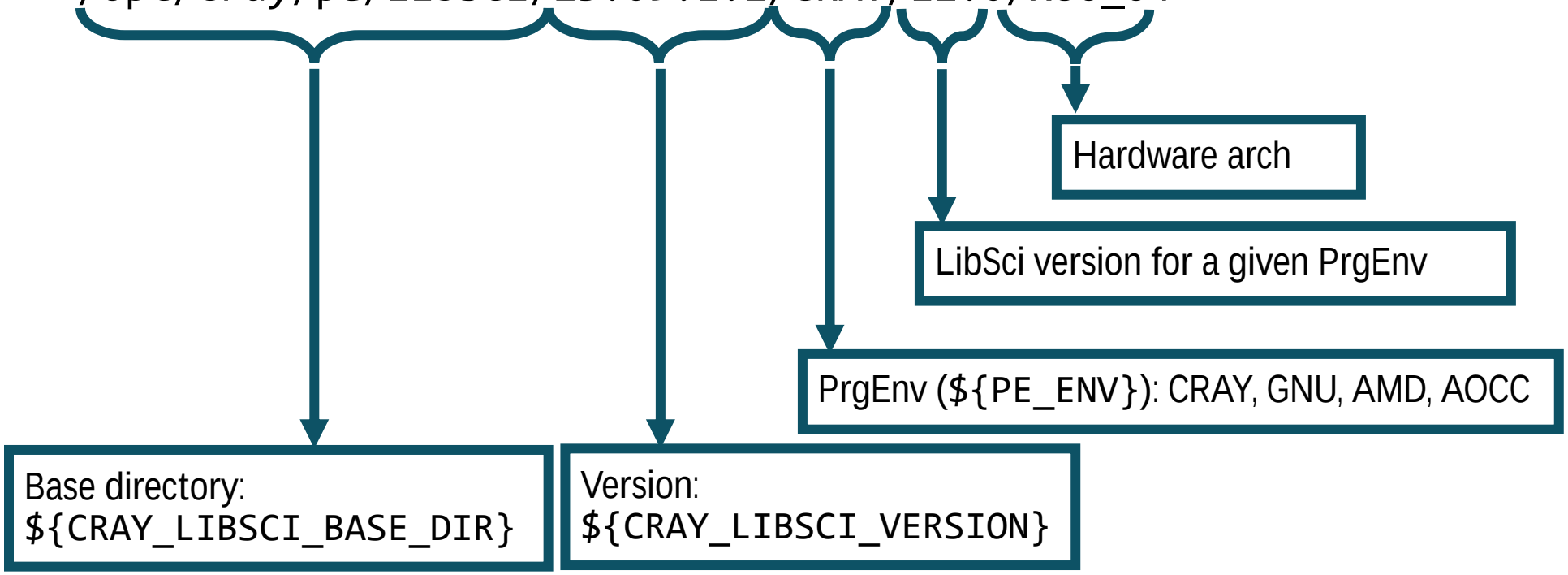
Alternatively, to temporarily use 1 thread one can wrap the library calls as follows…

```
#pragma omp parallel ...
#pragma omp single
{
  libsci_call();
}
```

# Manually linking LibSci libraries (1)

- The PE wrappers automatically inject flags for using LibSci depending on the PrgEnv
  - Current compatible version can be access via ${CRAY_LIBSCI_PREFIX_DIR}, e.g.

```
> echo ${CRAY_LIBSCI_PREFIX_DIR}
/opt/cray/pe/libsci/23.09.1.1/CRAY/12.0/x86_64
```



# Manually linking LibSci libraries (2)

- There are 4 versions of the LibSci library:
  - Serial, e.g. libsci_cray.a and libsci_cray.so
  - OpenMP, e.g. libsci_cray_mp.a and libsci_cray_mp.so
  - MPI, e.g. libsci_cray_mpi.a and libsci_cray_mpi.so
  - MPI + OpenMP: libsci_cray_mpi_mp.a and libsci_cray_mpi_mp.so
- Structured so that routines that must be linked with the MPI library (i.e., ScaLAPACK) are segregated from those that do not require MPI linking (i.e., BLAS and LAPACK)

➔ Note that the PrgEnv is part of the name of the library (cray, gnu, amd, aocc)

- The PE wrappers will link the right library:
  - By default links to the serial version of the LibSci library
  - Will link-in the OpenMP version of LibSci when the OpenMP is requested in the compilation and linking
  - Same considerations apply for MPI (assuming that the MPI module is loaded)

# Manually linking LibSci libraries (3)

It is possible to manually link specific serial and parallel cray-libsci libraries

```
-L ${CRAY_LIBSCI_PREFIX_DIR}/lib -l library
```

Where the environment variable `${CRAY_LIBSCI_PREFIX_DIR}` is set when the cray-libsci module is loaded.

For example, for CCE

- Includes: `-I $CRAY_LIBSCI_PREFIX_DIR/include`
- Linking the serial library: `-L $CRAY_LIBSCI_PREFIX_DIR/lib –l sci_cray`

# Threaded LAPACK

- Threaded LAPACK works exactly the same as threaded BLAS
- Anywhere LAPACK uses BLAS, those BLAS can be threaded
- Some LAPACK routines are threaded at the higher level
- No special instructions

➔ `man intro_lapack`

# ScaLAPACK

- ScaLAPACK is optimized for the interconnect
  - New collective communication procedures are added
  - Default topologies are changed to use the new optimizations
  - Much better strong scaling
- It also benefits from the optimizations in the BLAS library

➔ `man intro_scalapack`

# Iterative Refinement Toolkit (IRT)

- IRT is a suite of tools to help exploit single precision, obtaining solutions accurate to double precision
  - An automatic framework to use mixed precision under the covers
  - A maximum theoretical speed-up of 2x over the full precision methods may be obtained. In practice, the condition number of the user's matrices strongly affect performance of IRT, and the obtained speed-up is more likely to be in the range 1.2 to 1.7
- Used for serial and parallel LU, Cholesky and QR
- Either set IRT_USE_SOLVERS to 1 or use the advanced API

➔ `man intro_irt`

# FFTW

- Cray's main FFT library is FFTW from MIT
  - Some additional optimizations for HPE hardware
    - It will quickly generate highly optimized plans for some problem types.
    - This feature is enabled by default, but may be disabled by setting the environment variable FFTW_CRAY_FASTPLAN=0

- Usage is simple
  - Load the module (`fftw`)
    - By default, no version is loaded
  - In the code, call an FFTW plan

- Threading version is linked if OpenMP is requested

➔ `man intro_fftw3`

# A note about Intel MKL

- This can sometimes achieve better performance for an individual application
- The use of the Intel developer is not supportable on the Cray EX architecture with AMD processors
- We had a recipe to link with MKL instead of LibSci on previous generations
- Some sites have downloaded MKL and used it so please ask about this
- And report any cases to us where LibSci is not performing well

# LibSci for Accelerators: LibSci_ACC

- Provide basic libraries for accelerators
  - BLAS, LAPACK, ScaLAPACK
- Multiple use case support
  - Get the base use of accelerators with no code change
  - Get extreme performance of GPU with or without code change (use existing vendor GPU optimized libraries)
  - Extra tools for support of complex code
- Must be independent to OpenACC/OpenMP, but fully compatible
- Incorporate the existing GPU libraries into LibSci
  - Provide additional performance and usability
  - Maintain the Standard APIs where possible!

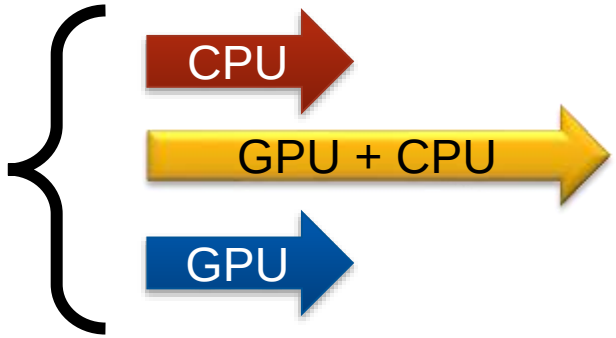
➔ Check `man intro_libsci_acc` for more details, including the list of supported functions

# Why LibSci_ACC?

- Code modification is required to use specific APIs of optimized (vendor) GPU libraries
  - E.g.: rocBLAS/hipBLAS, cuBLAS
- No compatibility to Legacy APIs
  - e.g. `hipblasDgemm(…)` VS `dgemm(…)`
- Portability across AMD and NVIDIA GPUs
- Includes Fortran API support
  - Other libraries require specific data types, primitives, and functions in order to interface to Fortran
- Includes optimized functions
  - Especially for distributed functions (ScaLAPACK)

# Three interfaces for three use cases

- Automatic interface
  - CPU pointers: dgetrf(M, N, h_A, LDA, IPIV, INFO)




  - GPU pointers: dgetrf(M, N, d_A, LDA, IPIV, INFO)


- Device interface
  - GPU pointers: dgetrf_acc(M, N, d_A, LDA, IPIV, INFO)


- CPU interface
  - CPU pointers: dgetrf_cpu(M, N, h_A, LDA, IPIV, INFO)


# Automatic interface

- You can pass either host pointers or device pointers to simple interface
  - No API changes!

- Host memory pointers
  - The library will automatically offload computational tasks to the GPU at runtime if it determines performance will be enhanced by a nontrivial amount (heuristic checking)
    - Move data to the GPU, compute on the GPU, results are transferred back to the host
  - If LibSci_ACC determines that the overhead of moving data to the GPU is greater than the benefit of executing computations on the GPU, the traditional LibSci routine will execute on the host CPU(s)
    - Can be controlled via environment variables
  - Possibility of hybrid computation CPU + GPU

- Device memory pointers
  - Performs operation on GPU
  - All data on the GPU

# Automatic interface – Environment variables control

- The avoid the possible overhead of the heuristic checking, each BLAS function (for all precisions and levels) may be individually specified to bypass the overhead of the automatic interface and go directly to the version indicated by the value of an environment variable

- E.g. for dgemm:
  - `LIBSCI_ACC_BYPASS_DGEMM=0` ➔ (this is the default value), the automatic version of the function is called
  - `LIBSCI_ACC_BYPASS_DGEMM=1` ➔ the GPU version of the function is directly called
  - `LIBSCI_ACC_BYPASS_DGEMM=2` ➔ the CPU version of the function is directly called
  - `LIBSCI_ACC_BYPASS_DGEMM=3` ➔ the hybrid version of the function is directly called

- For convenience, environment variables `LIBSCI_ACC_BYPASS_BLAS1`, `LIBSCI_ACC_BYPASS_BLAS2`, and `LIBSCI_ACC_BYPASS_BLAS3` are provided and will assign the defined value to all functions for the specified BLAS level
  - Combinations are possible, e.g. `LIBSCI_ACC_BYPASS_BLAS1=2` and `LIBSCI_ACC_BYPASS_DDOT=0` ➔ all BLAS1 calls, with the exception of `ddot`, will execute the CPU version. `ddot` will use the automatic version

# Device interface

- Forces use of the accelerated routine on the accelerator only, regardless of environment variable settings

- Requires that you have already allocated and copied your data to the device memory
  - For better performance, use pinned memory

- API
  - Every routine in libsci has a version with `_acc` suffix
  - E.g. `dgetrf_acc`
  - This resembles standard API except for the suffix and the device pointers

# Memory management

- Cray LibSci-Acc provides several native memory management routines so the users can manage host and device memory spaces without calling vendor runtime routines

- Currently available routines are
  - Allocate/free page-locked memory on the host: `libsci_acc_HostAlloc` and `libsci_acc_HostFree`
  - Allocate/free memory on the device: `libsci_acc_DeviceAlloc` and `libsci_acc_DeviceFree`
  - Register an existing host memory range so it can be accessed from device: `libsci_acc_HostRegister` and `libsci_acc_HostUnregister`
  - Memory copy between host and device: `libsci_acc_Memcpy`

# CPU Interface

- Forces use of the standard Cray LibSci routine on host processors only, regardless of environment variable settings
  - Need to preserve GPU memory
  - Don't want to incur transfer cost for a small operation

- Can force any operation to occur on CPU with `_cpu` version
  - Every routine has a `_cpu` entry-point
  - API is exactly standard otherwise

# Usage - Basics

- Usual modules (PrgEnv, craype-accel-amd-gfx90a, rocm)
- `module load cray-libsci_acc` (not loaded by default)

- Fortran and C/C++ interfaces (column-major assumed)
- Compile as normal

- To enable threading in the CPU library, set `OMP_NUM_THREADS`
- Consider the proper GPU binding
- Execute your code as normal

# LibSci_ACC DGEMM example

- Starting with a code that relies on dgemm
- The library will check the parameters at runtime
- If the size of the matrix multiply is large enough, the library will run it on the GPU, handling all data movement behind the scenes
- NOTE: Input and Output data are in the CPU memory

```
call dgemm('n','n',m,n,k,alpha,&
           a,lda,b,ldb,beta,c,ldc)
```

# LibSci_ACC DGEMM example – Interaction with OpenACC

- If the rest of the code uses OpenACC, it is possible to use the library with directives
- All data management performed by OpenACC
- Class the device version of dgemm
- All data is in CPU memory before and after data region
- Same considerations applies for OpenMP offload, HIP, CUDA

```
!$acc data copy(a,b,c)

!$acc parallel
!Do Something
!$acc end parallel

!$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
```

# LibSci_ACC DGEMM example – Interaction with OpenACC

- LibSci_ACC checks where the data pointers are pointing to (host or GPU memory)
- Since ‘a,’ ‘b’, and ‘c’ are device arrays, the library knows it should run on the device
- So just dgemm is sufficient

```
!$acc data copy(a,b,c)

!$acc parallel
!Do Something
!$acc end parallel

!$acc host_data use_device(a,b,c)

call dgemm     ('n','n',m,n,k,&
                alpha,a,lda,&
                b,ldb,beta,c,ldc)

!$acc end host_data
!$acc end data
```

# Final remarks on LibSci_ACC

- The environment variable setting `MPICH_GPU_SUPPORT_ENABLED=1` is required to enable GPU-resident computation for PBLAS functions

- We recommend enclosing a user program that will access LibSci_ACC routines between function calls to `libsci_acc_init()` and `libsci_acc_finalize()` to ensure optimal performance enhancement

- LibSci_ACC BLAS and LAPACK routines are thread safe
  - Both manual and automatic modes, can be called concurrently from multiple threads such as OpenMP's parallel region
- Any other LibSci_ACC routine is not thread safe

# Other remarks on BLAS-LibSci

- In case of errors:
  - Cray LibSci allocates internal buffers onto the stack and therefore expects an unlimited stack size
    - Try to use `ulimit -s unlimited` if you get a segfaults
    - Alternatively, set the environment variable `CRAYBLAS_ALLOC_TYPE` to 2
  - `CRAYBLAS_LEVEL1_LEGACY, CRAYBLAS_LEVEL2_LEGACY, CRAYBLAS_LEVEL3_LEGACY`
    - When set these environment variables to 1, they entirely disable the CrayBLAS framework for the corresponding BLAS Level calls, resulting in all BLAS calls to fallback to the basis code

- Environment variables
  - `CRAYBLAS_ABORT_ON_ERROR`
    - By default, the library prints an error message to the stdout and aborts if an invalid call to a BLAS routine is made
    - Setting `CRAYBLAS_ABORT_ON_ERROR=0` will cause the library only to report an error to the stdout stream and return from the BLAS call
  - `CRAYBLAS_PROFILING_VERBOSITY`
    - When `CRAYBLAS_PROFILING_VERBOSITY=1`, the routine name and argument values and printed to stdout
    - Set `CRAYBLAS_PROFILING_VERBOSITY=2` to also print the runtime in seconds.

# Take home message concerning libraries

- Do not re-invent the wheel but use scientific libraries wherever you can!
  - All the most widely used library families and frameworks readily available as optimized versions
  - Make sure you use the optimized version provided by the system instead of a reference implementation
- Access to LibSci routines is simple
  - No need to explicitly link - Programming Environment drivers (cc, ftn, CC) do this for you after loading the module
- Libsci_ACC allows to run routines on GPU with minimal or no change to your code
  - Just need to load the module and recompile
  - Simple interface available to enable hybrid, CPU or GPU execution of a routine depending on where memory pointers reside and problem size
  - Interface for advanced control is also available

- Documentation on cray-libsci via man pages:
  - libsci or intro_libsci
  - intro_blacs, intro_blas1, intro_blas2, intro_blas3, intro_cblas
  - intro_irt, intro_lapack, intro_scalapack
- Documentation on cray-libsci_acc via man page
  - `man intro_libsci_acc`