Compilers and Libraries

Ilias Katsardis
ikatsardis@cray.com

Don’t forget

- Use the `ftn`, `cc`, and `CC` wrappers
  - The wrappers uses your module environment to get all libraries and include directories for you. You don’t have to know their real location.
- You select the your environment by selecting one of the programming environments `PrgEnv-X` with `X=[cray|intel|gnu]` “`module swap PrgEnv-cray PrgEnv-intel`”
- Check that the craype-ivybridge module is loaded
- Use aprun to start your application on the compute nodes

Cross-compilation

- **If the compute and login nodes have different CPUs, one needs to cross-compile for the compute nodes**
  - The apps compiled for compute nodes might not run on login nodes
- **Use the wrappers ftn, cc and CC to cross-compile**
  - If you really need to run something on the login nodes, switch the module **craype-network-aries** to **craype-network-none**
  - Or use the flag **-target=local_host** or **-target-cpu=**
  - Or use the non-wrapper compiler commands (gfortran, gcc,...)
- **One may run into trouble with GNU automake or cmake**
  - Add the specifier **--host=x86_64-unknown-linux-gnu** for the configure tool
  - With cmake, provide the CMAKE_SYSTEM_NAME and the used compilers in a toolchain file or when invoking cmake, e.g.

```
cmake -DCMAKE_SYSTEM_NAME=Linux \
-DCMAKE_C_COMPILER=cc -DCMAKE_CXX_COMPILER=CC
```

Dynamic vs Static linking (1)

- **Static Linking**
  - The linked places all library code into the final executable
- **Dynamic Linking**
  - The library code is linked into the process at runtime
- **Site preference sets dynamic or static linking as default**
- **However, you can decide how to link,**
  1. You can either set `CRAYPE_LINK_TYPE` to "`static`" or "`dynamic`" (e.g. `export CRAYPE_LINK_TYPE=dynamic`) during compilation
  2. Or pass the `-static` or `-dynamic` option to the linking compiler
- **Features of dynamic linking :**
  - smaller executable, potential automatic use of new libs
  - Might need longer startup time to load and find the libs
  - Runtime loaded `modules` can potentially affect how the application runs (see next slide)

Dynamic vs Static linking (2)

- **Features of static linking :**
  - Larger executable (usually not a problem)
  - Faster startup
  - Application will run the same code every time it runs (independent of environment)

The three styles of dynamic linking

Shared libraries mean applications may use a different versions of a library at runtime than was linked at compile time. On the Cray XC30 there are three ways to control which version is used

1. *Default* – Follow the default Linux policy and at runtime use the system default version of the shared libraries (so may change as and when system is upgraded)
2. *pseudo-static* – Hardcodes the path of each library into the binary at compile time. Runtime will attempt to use this version when the application start (as long as lib is still installed). Set `CRAY_ADD_RPATH=yes` at compile
3. *Dynamic modules* – Allow the currently loaded PE modules to select library version at runtime. App must not be linked with `CRAY_ADD_RPATH=yes` and must add "`export LD_LIBRARY_PATH=$CRAY_LD_LIBRARY_PATH`" to run script

The Cray Compilation Environment (CCE)

CCE Overview

- **Cray technology focused on scientific applications**
  - Takes advantage of automatic vectorization
  - Takes advantage of automatic shared memory parallelization
- **Standard conforming languages and programming models**
  - ANSI/ISO Fortran 2008 standards compliant
  - ANSI/ISO C99 and C++2003 compliant
  - OpenMP 3.1 compliant, working on OpenMP 4.0
  - OpenACC 2.0
- **OpenMP and automatic multithreading fully integrated**
  - Share the same runtime and resource pool
  - Aggressive loop restructuring and scalar optimization done in the presence of OpenMP
  - Consistent interface for managing OpenMP and automatic multithreading
- **PGAS languages (UPC & Fortran coarrays) fully optimized and integrated into the compiler**

General Cray Compiler Flags

- **Optimisation Options**

-O2	safe flags [enabled by default]
-O3	aggressive optimization
-O ipaN (ftn) or **-hipaN (cc/CC)**	inlining, N=0-5 [default N=3]

- **Create listing files with optimization info**

-hlist=a	creates a listing file with all optimization info
-hlist=m	produces a source listing with loopmark information

- **Parallelization Options**

-h omp	Recognize OpenMP directives [default]
-h threadN	control the compilation and optimization of OpenMP directives, N=0-3 [default N=2]

➔ **More info: man crayftn, man craycc, man crayCC**

Fortran Source Preprocessing

For a source file to be preprocessed automatically, it must have an uppercase extension, either .F (for a file in fixed source form), or .F90, .F95, .F03, .F08, or .FTN (for a file in free source form). To specify preprocessing of source files with other extensions, including lowercase ones, use the -eP or -eZ options

- **-eP**: Performs source preprocessing on Fortran source files, **but does not compile**. Generates file.1, which contains the source code after the preprocessing has been performed and the effects have been applied to the source program.
- **-eZ**: similar to **-eP**, but it also performs compilation on Fortran source files

Unrolling

- **By default, the compiler attempts to unroll all loops, unless the `NOUNROLL` directive is specified for a loop**
  - Generally, unrolling loops increases single processor performance at the cost of increased compile time and code size
- **-hunrollN where N=0,1,2, globally control loop unrolling and changes the assertiveness of the UNROLL directive**
  - **0**: No unrolling (ignore all `UNROLL` directives and do not attempt to unroll other loops)
  - **1**: Attempt to unroll loops if there is proof that the loop will benefit
  - **2**: (Default) Attempt to unroll all loops (includes array syntax implied loops), except those marked with the `NOUNROLL` directive.

Vectorization

- **-hvectorN (cc/CC) where N=0…3, specify the level of automatic vectorizing to be performed. Vectorization results in significant performance improvements with a small increase in object code size. Vectorization directives are unaffected by this option**
  - **0**: No automatic vectorization
  - **1**: Specifies conservative vectorization. Loop nests are restructured. No vectorizations that might create false exceptions are performed. Results may differ slightly from results obtained when N=0 is specified because of vector reductions
  - **2**: (Default) Specifies moderate vectorization. Characteristics include moderate compile time and size. Loop nests are restructured
  - **3**: Specifies aggressive vectorization. Loop nests are restructured. Vectorizations that might create false exceptions in rare cases may be performed.

Floating-Point Optimizations

- **The -hfpN option, where N=0…4, controls the level of floating-point optimizations: N=0 gives the compiler minimum freedom to optimize floating-point operations, while N=4 gives it maximum freedom. The higher the level, the less the floating-point operations conform to the IEEE standard.**
  - **N=0 and N=1**: Use this option only when your code pushes the limits of IEEE accuracy or requires strong IEEE standard conformance. Executable code is slower than higher floating-point optimization levels
  - **N=2**: default value. It performs various generally safe, non-conforming IEEE optimizations
  - **N=3**: This option should be used when performance is more critical than the level of IEEE standard conformance provided by N=2. This is the suggested level of optimization for many applications.
  - **N=4**:You should only use this option if your application uses algorithms which are tolerant of reduced precision.

Floating-Point Optimization Flags Comparison

The **-hfpN** option, where N=0…4, controls the level of floating-point optimizations

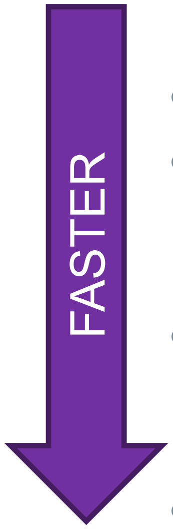
Optimization	fp0	fp1	fp2 (default)	fp3	fp4
Safety	Maximum	High	High	Moderate	Low
Complex divisions	Accurate and slower	Accurate and slower	Fast	Fast	Fast
Exponentiation rewrite	None	None	When benefit is very high	Always	Always
Strength reduction	None	None	Fast	Fast	Fast
Rewrite division as reciprocal equivalent	None	None	Yes	Aggressive	Aggressive
Floating point reductions	Slow	Fast	Fast	Fast	Fast
Expression factoring	None	Yes	Yes	Yes	Yes
Expression tree balancing	None	None	Yes	Yes	Yes
Inline 32-bit operations	No	No	No	Yes	Yes
Fused multiply-add	No	Yes	Yes	Yes	Yes

Why are CCE's results sometimes different?

- **We do expect applications to be conformant to language requirements**
  - This include not over-indexing arrays, no overlap between Fortran subroutine arguments, and so on
  - Applications that violate these rules may lead to incorrect results or segmentation faults
  - Note that languages do not require left-to-right evaluation of arithmetic operations, unless fully parenthesized
    - This can often lead to numeric differences between different compilers
    - Use **-hadd_paren** to add automatically parenthesis to select associative operations (+,–,*). Default is **-hnoadd_paren**
- **We are also fairly aggressive at floating point optimizations that violate IEEE requirements**
  - Use **-hfp[0-4]** flag to control that

About reproducibility

- **Results can vary with the number of ranks or threads**
  - Use **-hflex_mp=option** to control the aggressiveness of optimizations which may affect floating point and complex repeatability when application requirements require identical results when varying the number of ranks or threads.
  - **option** in order from least aggressive to most is:
    - intolerant: has the highest probability of repeatable results, but also has the highest performance penalty
    - strict: uses some safe optimizations, with high probability of repeatable results.
    - conservative: uses more aggressive optimization and yields higher performance than intolerant, but results may not be sufficiently repeatable for some applications
    - default: uses more aggressive optimization and yields higher performance than conservative, but results may not be sufficiently repeatable for some applications
    - tolerant: uses most aggressive optimization and yields highest performance, but results may not be sufficiently repeatable for some applications



Recommended CCE Optimization Options

- **Default optimization levels should be good**
  - It's the equivalent of most other compilers -O3
  - It is also our most thoroughly tested configuration
- **Use -O3,fp3 (or -O3 -hfp3, or some variation) if the application runs cleanly with these options**
  - **-O3** only gives you slightly more than the default **-O2**
  - We also test this thoroughly
  - **-hfp3** gives you more floating point optimization (default is **-hfp2**)
- **If an application is intolerant of floating point reordering, try a lower -hfp number**
  - Try **-hfp1** first, only **-hfp0** if absolutely necessary (**-hfp4** is the maximum)
  - Might be needed for tests that require strict IEEE conformance
  - Or applications that have 'validated' results from a different compiler
  - Higher numbers are not always correlated with better performance

Recommended for bit reproducibility

Start from this set

- -hflex_mp=conservative –hfp1 –hadd_paren

Fortran precision defaults

Use –s option

- **-s real64**
  REAL (64bits), DOUBLE PRECISION (64bits)
  COMPLEX (128bits), DOUBLE COMPEX (128 bits)
- **-s integer64**
  Default integers to 64 bits

- See crayftn manpage for other precision options

Diagnostic Flags

- **-Rb (ftn) or -h bounds (cc/CC)**
  - Fortran: Enables checking of array bounds at runtime
  - C/C++: Enables checking of pointer and array references at runtime. **-h nobounds** disables these checks
- **-eo (ftn) or -hdisplay_opt (cc/CC)**
  - Display the compiler optimization settings currently in force
- **-ei (ftn)**
  - Initialize all undefined local stack, static, and heap variables to an invalid value (signaling NaN)
- **-T (ftn)**
  - Disables the compiler but displays all options currently in effect

Threading

- **OpenMP is on by default**
  - This is the opposite default behavior that you get from GNU and Intel compilers
  - Optimizations controlled by **-OthreadN (ftn)** or **-hthreadN (cc/CC)**, N=0-3 [default N=2]
  - To shut off use **-O/-h thread0** or **-xomp (ftn)** or **-hnoomp**
- **Autothreading is off by default**
  - **-hautothread** to turn it on
  - Interacts with OpenMP directives
- **If you do not want to use OpenMP and have OMP directives in the code, make sure to shut off OpenMP at compile time**

Other flags in brief

- **-h restrict=[a|f]**
  - C/C++ option which tells the compiler to treat certain classes of pointers as restricted pointers. You can use this option to enhance optimizations (this includes vectorization).
- **-h cacheN**
  - Specifies the levels of automatic cache management to perform. Values for N are between 0 (cache blocking turned off) and 3 (aggressive automatic cache management). Symbols are placed in the cache when the possibility of cache reuse exists. Default value is N=2.

Other flags in brief (cont)

- **-h PIC**
  - Generate position independent code (PIC), which allows a virtual address change from one process to another, as is necessary in the case of shared, dynamically linked objects. The virtual addresses of the instructions and data in PIC code are not known until dynamic link time.

- **-h[system|default]_alloc**
  - The **-hsystem_alloc** option causes the compiler to use the native malloc implementation provided by the OS. By default, the compiler uses a modified malloc implementation which offers better support for Cray memory needs. This is a link-time option.

Loopmark: Compiler Feedback

- –hlist=m …
- Compiler generates an <source file name>.lst file
  - Contains annotated listing of your source code with letter indicating

```
%%%    L o o p m a r k   L e g e n d    %%%
Primary Loop Type       Modifiers
------- ---- ----       ---------
                        a - vector atomic memory operation
A   - Pattern matched   b – blocked
C   - Collapsed         f – fused
D   - Deleted           i – interchanged
E   - Cloned            m - streamed but not partitioned
I   - Inlined           p - conditional, partial and/or computed
M   - Multithreaded     r – unrolled
P   - Parallel/Tasked   s – shortloop
V   - Vectorized        t - array syntax temp used
W   - Unwound           w - unwound
```

Example: Cray loopmark messages

```
29.   b-------<  do i3=2,n3-1
30.   b b-----<     do i2=2,n2-1
31.   b b Vr--<        do i1=1,n1
32.   b b Vr            u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
33.   b b Vr    *         + u(i1,i2,i3-1) + u(i1,i2,i3+1)
34.   b b Vr            u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
35.   b b Vr    *         + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
36.   b b Vr-->        enddo
37.   b b Vr--<        do i1=2,n1-1
38.   b b Vr            r(i1,i2,i3) = v(i1,i2,i3)
39.   b b Vr    *        - a(0) * u(i1,i2,i3)
40.   b b Vr    *        - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )
41.   b b Vr    *        - a(3) * ( u2(i1-1) + u2(i1+1) )
42.   b b Vr-->        enddo
43.   b b----->     enddo
44.   b------->  enddo
```

Example: Cray loopmark messages (cont)

```
 ftn-6289 ftn: VECTOR File = resid.f, Line = 29
   A loop starting at line 29 was not vectorized because a
recurrence was found on "U1" between lines 32 and 38.
ftn-6049 ftn: SCALAR File = resid.f, Line = 29
   A loop starting at line 29 was blocked with block size 4.
ftn-6289 ftn: VECTOR File = resid.f, Line = 30
   A loop starting at line 30 was not vectorized because a
recurrence was found on "U1" between lines 32 and 38.
ftn-6049 ftn: SCALAR File = resid.f, Line = 30
   A loop starting at line 30 was blocked with block size 4.
ftn-6005 ftn: SCALAR File = resid.f, Line = 31
   A loop starting at line 31 was unrolled 4 times.
ftn-6204 ftn: VECTOR File = resid.f, Line = 31
   A loop starting at line 31 was vectorized.
ftn-6005 ftn: SCALAR File = resid.f, Line = 37
   A loop starting at line 37 was unrolled 4 times.
ftn-6204 ftn: VECTOR File = resid.f, Line = 37
   A loop starting at line 37 was vectorized.
```

Compiler Message System

- **The `explain` command displays an explanation of any message issued by the compiler. The command takes as an argument, the message number, including the number's prefix (`ftn-` for ftn or `CC-` for cc/CC)**

  Example:

```
% cc bug.c
CC-24 cc: ERROR File = bug.c, Line = 1
An invalid octal constant is used.
int i = 018;
         ^

1 error detected in the compilation of "bug.c".
% explain CC-24
An invalid octal constant is used.
Each digit of an octal constant must be between 0 and 7,
inclusive. One or more digits in the indicated octal
constant are outside of this range. Change each digit in
the octal constant to be within the valid range.
```

➔ **More info: man explain (when PrgEnv-cray loaded)**

Compiler Message System (cont)

- **-h [no]msgs**
  - Enables or disables the writing of optimization messages to `stderr`. Default is **-h nomsgs**
- **-h [no]negmsgs**
  - Enables or disables the writing of messages to `stderr` that indicate why optimizations such as vectorization, inlining, or cloning did not occur in a given instance. Default is -h nonegmsgs
- **-m *n* (ftn) or -h msglevel_*n* (cc/CC)**
  - Specifies the lowest level of severity of messages to be issued. Messages at the specified level and above are issued. Values of ***n*** are:
    - 0: Comment
    - 1: Note
    - 2: Caution
    - 3: Warning (default)
    - 4: Error
- **-M msg*n*[,…] (ftn) or -h nomessage=*n*[:...] (cc/CC)**
  - Suppresses specific messages at the warning, caution, note, and comment levels, where ***n*** is the number of a message to be disabled (multiple numbers are possible)

CCE Directives

- **The Cray compiler supports a full and growing set of directives and pragmas**
  - Fortran:
    - `!dir$ concurrent`
    - `!dir$ ivdep`
    - `!dir$ interchange`
    - `!dir$ unroll`
    - `!dir$ loop_info [max_trips] [cache_na]` ... Many more
    - `!dir$ blockable`
  - For C/C++ replace **`!dir$`** with **`#pragma [_CRI]`**
    - The **`_CRI`** specification is optional; it ensures that the compiler will issue a message concerning any directives that it does not recognize. Diagnostics are not generated for directives that do not contain the **`_CRI`** specification.

➔ More info: man directives
man loop_info

Macros

- **Cray compilers define the following macros:**
  - Fortran: `_CRAYFTN`
  - C/C++: `_CRAYC`

- **For example, the macros can be used to ensures that other compilers will not interpret the directives by encapsulating them inside `#if … #endif`**

```
#if _CRAYC
     #pragma _CRI directive
#endif
```

- **Some compilers diagnose any directives that they do not recognize. The Cray C/C++ compilers diagnose directives that are not recognized only if the _CRI specification is used.**

Cray programming environment: assign

- Associates options with Fortran I/O unit numbers and file names for use during the library open processing, i.e. you can tell the Fortran runtime how to treat a file, without changing your code
  - **assign [assign options] assign_object**

- Interesting assign options

-R	removes all assign options for assign_object
-N <numcon>	specifies foreign numeric conversion

- assign_object used to specify the object of assign options

f:<filename>	applies to filename
u:<unit>	applies to Fortran unit number
g:su	applies to all Fortran sequential unformatted files

How to handle byte-swapped files with CCE

- **Explicit usage of assign**
  - Can control which files are byte-swapped

```
export FILENV=.assign
assign -R
assign -N swap_endian f:aof
aprun a.out
```

- **Link the application with –hbyteswapio**
  - Forces byte-swapping of all input and output files for direct and sequential unformatted I/O
  - This is equivalent to set

```
assign -N swap_endian g:su ←all sequential unformatted
assign -N swap_endian g:du ←all direct unformatted
```

➔ More info: man assign (when PrgEnv-cray loaded)

Default Output Formats

- List-directed output depends on the value being written
  - `assign` command can be used to change that
- Let's take this code for example

```
integer :: ia(4)
real    :: ra(4)
ia = 102
ra = 200.10
print *, ' ia=',ia
print *, ' ra=',ra
```

Output

```
ia= 4*102
ra= 4*200.100006
```

By setting

```
export FILENV=FILETMP
assign -U on g:sf
```

and rerunning the code (without recompiling it), the output becomes

```
ia=          102         102         102         102
ra=     200.1000    200.1000    200.1000    200.1000
```

➔ More info: man assign (when PrgEnv-cray loaded)

Brief notes on Intel and GNU Compilers

GNU and Intel compiler flags

- **More or less all optimizations and features provided by CCE are available in Intel and GNU compilers**
- **GNU compiler serves a wide range of users & needs**
  - Default compiler with Linux, some people only test with GNU
  - Defaults are conservative
  - -O3 includes vectorization and most inlining
- **Intel compiler is typically more aggressive in the optimizations**
  - Defaults are more aggressive (e.g -O2), to give better performance "out-of-the-box"
    - Includes vectorization; some loop transformations such as unrolling; inlining within source file
  - Options to scale back optimizations for better floating-point reproducibility, easier debugging, etc.
  - Additional options for optimizations less sure to benefit all applications

Compiler feedback

- **Intel**
  - `ftn/cc -opt-report 3 -vec-report 6`
  - If you want this into a file: add `-opt-report-file=filename`
  - See `ifort --help reports`
- **GNU**
  - `-ftree-vectorizer-verbose=9`

Recommended compiler optimization levels

- **Intel compiler**
  - The default optimization level (equal to -O2) is safe and gives usually good performance
  - Try with **-O3** (verify correctness & performance)
    - If that works still, you may try with `-Ofast`
  - Also setting `-fp-model fast=2` (or =1) may give some additional performance
    - Further relaxed precision with `-fno-prec-div -fno-prec-sqrt`
  - Loop unrolling with `-funroll-loops` or `-unroll-aggressive` may also be beneficial

WARNING – Intel Helper Threads

- **The Intel OpenMP runtime creates more threads than you might expect**
  - It creates an extra helper thread (n+1 threads in total)
  - It also has it’s own method of binding to CPUs (KMP_AFFINITY)

- **Unfortunately both of these options can cause complications with CLE binding**

- **Cray advice…**
  - Don’t use KMP_AFFINITY to bind threads:
    - export KMP_AFFINITY=disabled
    - aprun –cc [numa_node|none] <exe>

Recommended compiler optimization levels

- **GNU compiler**
  - Add `-mavx -march=core-avx-i -mtune=core-avx-i`
  - Almost all HPC applications compile correctly with using `-O3`, so do that instead of the cautious default
    - `-Ofast` may give minor extra performance on top of -O3
  - `-ffast-math` may give some extra performance (but verify results)
  - `-funroll-loops` or `-funroll-all-loops` benefit most applications

Compiler man pages

- **The cc(1), CC(1), and ftn(1) man pages contain information about the compiler driver commands**
- **Cray compiler: man craycc(1), crayCC(1), and crayftn(1)**
- **GNU compiler: gcc(1), g++(1), and gfortran(1)**
- **Intel: `icc(1), icpc(1), ifort(1)`**

- **To verify that you are using the correct version of a compiler, use:**
  - -V option on a cc, CC, or ftn command with CCE
  - --version option on a cc, CC, or ftn command with GNU

Cray, Intel and GNU compiler flags

Feature	Cray	Intel	GNU
Listing	-hlist=a	-opt-report3	-fdump-tree-all
Free format (ftn)	-f free	-free	-ffree-form
Vectorization	By default at -O1 and above	By default at -O2 and above	By default at -O3 or using -ftree-vectorize
Inter-Procedural Optimization	-hwp	-ipo	-flto (note: link-time optimization)
Floating-point optimizations	-hfpN, N=0...4	-fp-model [fast|fast=2|precise| except|strict]	-f[no-]fast-math or -funsafe-math-optimizations
Suggested Optimization	(default)	-O2 -xAVX	-O2 -mavx -ftree-vectorize -ffast-math -funroll-loops
Aggressive Optimization	-O3 -hfp3	-fast	-Ofast -mavx -funroll-loops
OpenMP recognition	(default)	-fopenmp	-fopenmp
Variables size (ftn)	-s real64 -s integer64	-real-size 64 -integer-size 64	-freal-4-real-8 -finteger-4-integer-8

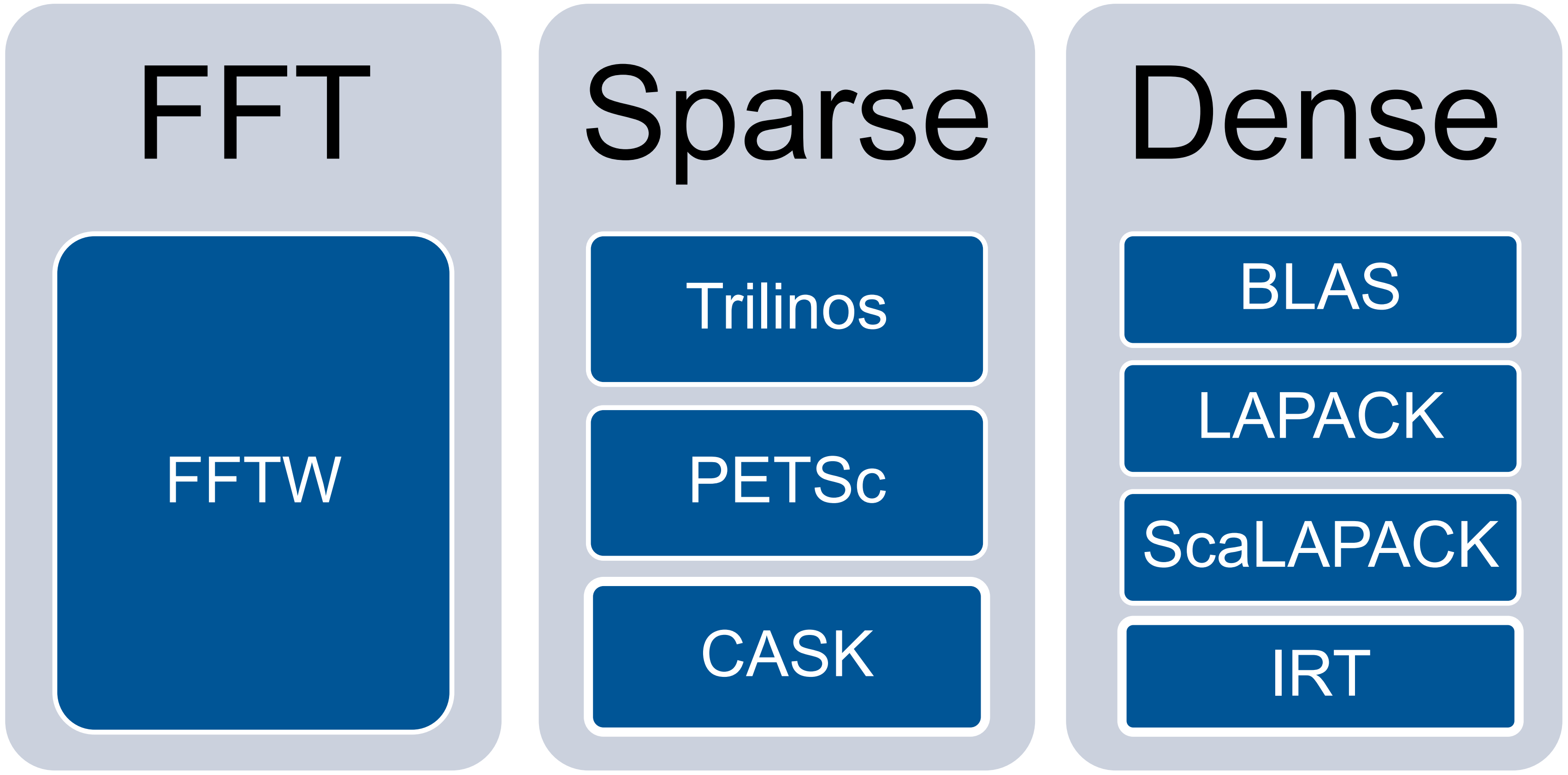
Summary - Compilers

- **Three compiler environments available: Cray, Intel, GNU**
  - All of them accessed through the wrappers ftn, cc and CC – just do module swap to change a compiler!
- **There is no universally fastest compiler – but performance depends on the application, even input**
  - We try however to excel with the Cray Compiler Environment
  - If you see a case where some other compiler yields better performance, let us know!
- **Compiler flags do matter – be ready to spend some effort for finding the best ones for your application**

Cray Scientific Libraries

Overview

Scientific libraries on XC – functional view



What makes Cray libraries special

1. **Node performance**
   - Highly tuned routines at the low-level (ex. 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

LibSci

- **LibSci**
  - The compiler wrappers should do it all for you – no need to explicitly link
  - For threads, set OMP_NUM_THREADS
    - Threading is used within LibSci
    - If you call within a parallel region, single thread used
- **FFTW**
  - `module load fftw` (there are also wisdom files available)
- **PETSc**
  - `module load petsc` (or module load petsc-complex)
  - Use as you would your normal PETSc build
- **Trilinos**
  - `module load trilinos`
- **Cray Adaptive Sparse Kernels (CASK)**
  - You get optimizations for free

Third-party libraries

- **The modules cray-trilinos and cray-petsc / cray-petsc-complex contain the popular Trilinos and PETSc packages**
  - These will automatically employ the Cray Adaptive Sparse Kernels
- **The module cray-tpsl contains ready builds of some other quite common libraries and solvers:**
  - MUMPS, ParMetis, SuperLU, SuperLU_DIST, Hypre, Scotch, Sundials
  - These are for your convenience (i.e. no need to build the library yourself) but do not feature Cray-specific modifications

A Note on Intel MKL

- **The Intel Math Kernel Libraries (MKL) is an alternative to LibSci**
  - Features tuned performance for Intel CPUs as well
- **Linking quite complicated, but the Intel MKL Link Line Advisor can tell you what to add to your link line**
  - http://software.intel.com/sites/products/mkl/
- **Using MKL together with the Intel compilers (PrgEnv-intel) is usually straightforward**

Linking with MKL and PrgEnv-cray

- **PrgEnv-cray compatible with sequential, not threaded, MKL**
- **Examples assume you have loaded the intel module (to define the env var INTEL_PATH)**
  - Typical case: You want to use MKL BLAS and/or LAPACK

```
    -L ${INTEL_PATH}/mkl/lib/intel64/ \
    -Wl,--start-group \
    -lmkl_intel_lp64 -lmkl_sequential -lmkl_core \
    -Wl,--end-group
```

  - Another typical case: You want to use MKL serial FFTs/DFTs

```
    Same as above (need more for FFTW interface)
```

  - A less typical case: You want to use MKL distributed FFTs

```
    -L ${INTEL_PATH}/mkl/lib/intel64/ \
    -Wl,--start-group \
    -lmkl_cdft_core -lmkl_intel_lp64 -lmkl_sequential \
    -lmkl_core -lmkl_blacs_intelmpi_lp64 \
    -Wl,--end-group
```

Take-home messages regarding 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 XC optimized versions**
  - And if the cornerstone library of your application is still missing, let us know about it!
- **Make sure you use the optimized version provided by the system instead of a reference implementation**