Hewlett Packard Enterprise

Compilers and Parallel Programming Models

Comprehensive General LUMI Course

April 23–26, 2024

Agenda

- Using compilers and linking defaults
- Cray Compiling Environment (CCE) Compilers
  - Fortran compiler
  - C/C++
- Notes on other compilers
- Compiler Suites for GPUs
- Parallel programming models

Reminder on Compiler Usage

Set the context with appropriate modules

- PrgEnv to get compiler family: PrgEnv-cray, PrgEnv-gnu, PrgEnv-aocc, PrgEnv-amd
- Any extra modules for libraries you want to reference
- A craype- architecture module will select the target hardware architecture
  - For example
    - CPU: craype-x86-rome, craype-x86-milan, craype-x86-trento
    - GPU: craype-accel-amd-gfx90a

Backend Compiler Version

- Check the compiler version to know which backend you are using in the compiler wrappers
  - `--version` flag
- E.g.
  - PrgEnv-cray

```
> cc --version
Cray clang version 16.0.1
> ftn --version
Cray Fortran : Version 16.0.1
```

  - PrgEnv-amd

```
> cc --version
AMD clang version 14.0.0 (https://github.com/RadeonOpenCompute/llvm-project roc-5.2.3 22324 d6c88e5a78066d5d7a1e8db6c5e3e9884c6ad10e)
> ftn --version
AMD flang-new version 14.0.0 (https://github.com/RadeonOpenCompute/llvm-project roc-5.2.3 22324 d6c88e5a78066d5d7a1e8db6c5e3e9884c6ad10e)
```

Mixed Compiler Module

- There is a special ‘mixed’ compiler module that can be used to load for example a specific GCC version
  - This is compatible with the Lmod hierarchy and allows PrgEnv-cray along with GCC
  - Available for AOCC, AMD, CCE, GCC (check via `module av mixed`)
- As an example

```
% which gcc ; gcc --version
/usr/bin/gcc
gcc (SUSE Linux) 7.5.0

% module list PrgEnv

Currently Loaded Modules Matching: PrgEnv
  1) PrgEnv-cray/8.3.3
% module load gcc-mixed
% gcc --version
gcc (GCC) 12.2.0 20220819 (Cray Inc.)

% cc --version
Cray clang version 15.0.0  (324a8e7de6a18594c06a0ee5d8c0eda2109c6ac6)
```

CPU Target Module

- The login nodes (UANs) have AMD “Rome” CPUs, while the compute nodes have AMD “Milan” (LUMI-C) and “Trento” (LUMI-G) CPUs
  - By default, the CPU target module is `craype-x86-rome`
  - Note that both ”Milan” and “Trento” are Zen3 cores
  - Need to swap target CPU module: `module load craype-x86-trento`

  ➔ It implies cross-compilation for the compute nodes
- The wrappers ftn, cc and CC will use the corresponding CPU target flag
  - You can use the flag `-craype-verbose` to check which flags are used by the backends, e.g. CCE:

```
> cc -craype-verbose --version
clang -march=znver2 -dynamic –version
…
> module load craype-x86-trento
> cc -craype-verbose --version
clang -march=znver3 -dynamic --version
```

- Important
  - Avoid directly specifying compiler hardware target flags, e.g. `-march=native -mtune=native -mavx2`

GPU Target Module

- No GPU-specific modules are loaded by default
- LUMI-G (Mi250X) requires `craype-accel-amd-gfx90a` module

```
module load craype-accel-amd-gfx90a
```

- By loading this module, you enable:
  - GPU support in the PrgEnv modules, e.g. MPI G2G support
  - Specific OpenMP and OpenACC offload flags, e.g. CCE

```
> module load craype-accel-amd-gfx90a
> ftn -craype-verbose --version
ftn_driver.exe -hcpu=x86-trento -haccel=amdgcn-gfx90a -hnetwork=ofi -hdynamic --version
> cc -craype-verbose --version -fopenmp
clang -march=znver3 -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx90a -dynamic --version -fopenmp
```

- NOTE:
  - OpenACC supported only in PrgEnv-cray Fortran, see `man intro_openacc`
  - OpenMP offload supported in PrgEnv-cray (see `man intro_openmp`) and PrgEnv-amd

Disabling Cross-compilation and cmake

- If you really need to run something on the login nodes, swap the module

```
module swap craype-network-ofi craype-network-none
```

  - Network module `craype-network-ofi` is loaded by default
  - `craype-network-none` disable MPI compilation
- Or use the flag `-target-cpu=x86-rome -target-network=none`
- 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
```

Static and Dynamic Linking

- Static Linking
  - The linker 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
- You can decide how to link if a choice is supported
  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

Static and Dynamic Linking

- 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)
- 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)

- On the HPE Cray EX systems only dynamic linking is supported

Dynamic Linking: 3 Styles

Use of shared libraries means applications may use a different versions of a library at runtime than was linked at compile time. On the Cray EX 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=${CRAY_LD_LIBRARY_PATH}:$LD_LIBRARY_PATH
   ```

   to run script environment

   Note that you must do this if you are not using default modules

Hugepages

- COS supports multiple pagesizes, hugepages are larger than the default (4k)
- There can be a performance advantage for application that use large datasets and/or MPI buffers
- The hugetlbfs library is used to map application memory segments into hugepages locations
- Modules are provided to load hugepages support as appropriate

```
richards@uan03:~> module avail craype-hugepages

-------------------------- HPE-Cray PE target modules --------------------------
   craype-hugepages128M    craype-hugepages2G     craype-hugepages512M
   craype-hugepages16M     craype-hugepages2M     craype-hugepages64M
   craype-hugepages1G      craype-hugepages32M    craype-hugepages8M
   craype-hugepages256M    craype-hugepages4M

richards@uan03:~> module load craype-hugepages8M
```

- See `man intro_hugepages`
- Load module at link time, choose page size via module at runtime.

The Cray Compilation Environment (CCE)

CCE Overview

- The default compiler on EX systems
  - Specifically designed for HPC applications
  - Takes advantage of Cray's experience with automatic vectorization and shared memory parallelization

- Standards support for multiple languages and programming models
  - Fortran, C, C++
    - Supports most of Fortran 2018 (ISO/IEC 1539:2018) with some exceptions
    - C/C++ compiler is based on Clang/LLVM
  - **OpenMP (including offload for AMD and NVIDIA GPUs)**
    - Full OpenMP 4.5 and partial OpenMP 5.0 and 5.1, see `man intro_openmp`
  - **OpenACC (Fortran only, AMD and NVIDIA GPUs)**
    - Full OpenACC 2.0 and partial OpenACC 2.x/3.x, see `man intro_openacc`
  - HIP compilation
  - MPI Interfaces

- Full integrated and optimised support for PGAS languages
  - UPC 1.2 and Fortran 2008 coarray support

Notes on subsequent slides

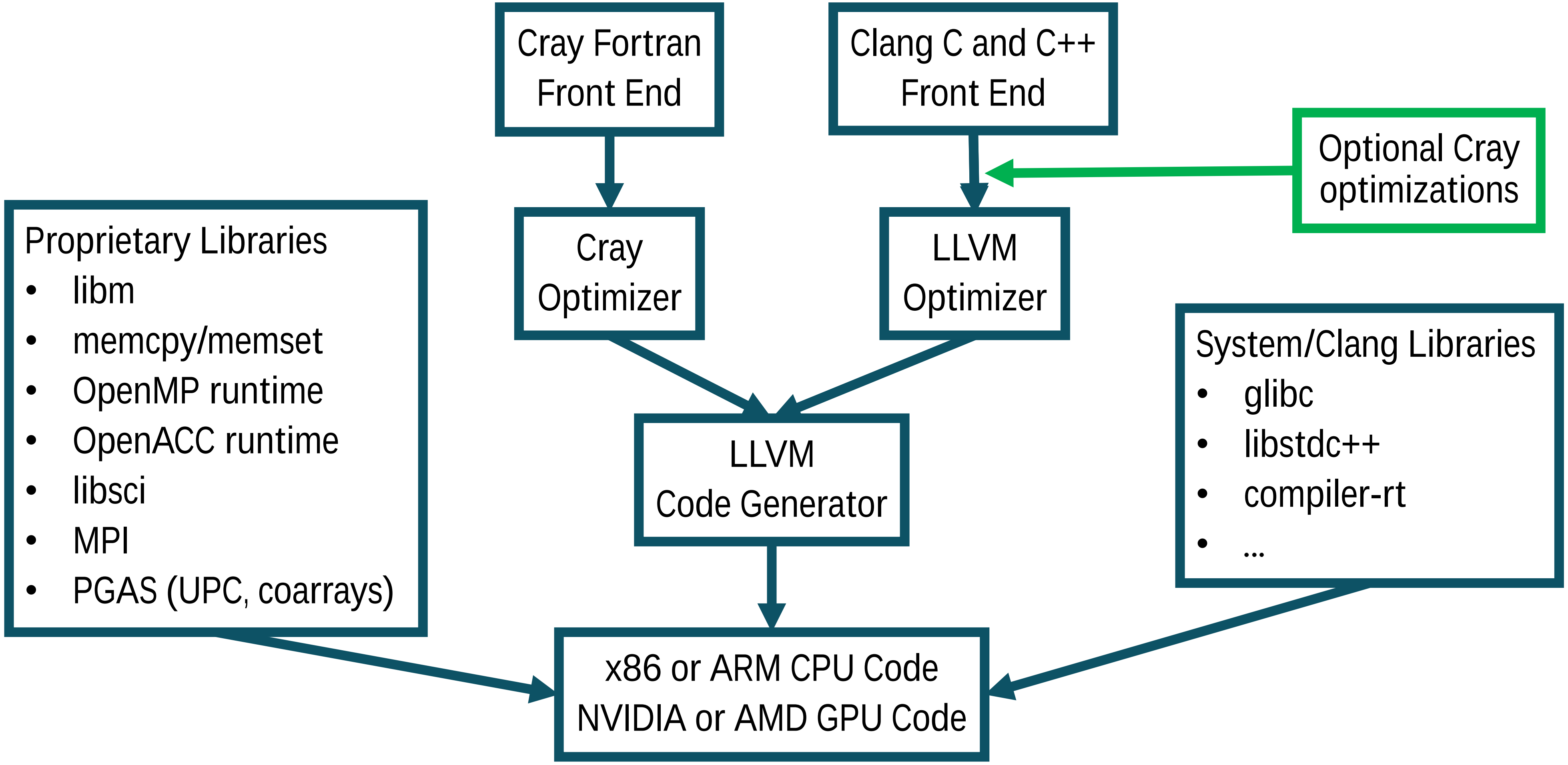
- The CCE compilers transitioned to the use of clang for C and C++ starting with CCE 9
- Cray features/optimizations are added on top of clang
- Only the clang-based compilers are available starting CCE 10
  - ➔ Some old configuration files can still report old CCE flags

- In subsequent slides we will explain usage of the Fortran compiler
  - We concentrate on controls relevant to porting and performance

- We mention some details of the Intel compiler which may be useful for porting but note that this compiler is not supported on Cray EX AMD platforms

- A later section will address Cray-specific changes to the clang/LLVM infrastructure

Current CCE Architecture (2018 to present)



HPE Cray Programming Environment Fortran Compiler

General CCE Fortran Compiler Flags

- Optimisation Options
  - -O2 safe flags [enabled by default]
  - -O3 aggressive optimization
  - -O ipaN 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
  - -f openmp Recognize OpenMP directives
  - -h threadN control the compilation and optimization of OpenMP directives, N=0-3 [default N=2]
  - -h acc Enables or disables the compiler recognition of OpenACC accelerator directives

➔More info: man crayftn

➔https://support.hpe.com/hpesc/public/docDisplay?docId=dp00003391en_us

Fortran Source Processing

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

Inlining with CCE

- Inlining is enabled by default
  - Command line option -O ipaN where N=0...5, provides a set of choices for inlining behavior (default is N=3)

- By default, all inlining candidates come from the current source file
  - The -O ipafrom=source[:source] option instructs the compiler to look for inlining candidates from other source files, or a directory of source files, e.g.

    – `ftn -Oipafrom=b.f a.f` tells the compiler to look for inlining candidates within b.f when compiling a.f

    – `ftn -Oipafrom=./dir a.f` tells the compiler to look for inlining candidates in all the valid source files that exist in the directory ./dir when compiling a.f

Whole-Program Compilation

- The Program Library (PL) feature allows the user to specify a repository of compiler information for an application build
  - This repository provides the framework for future productivity features such as
    - Whole program static error detection
    - Incremental recompilation
    - Provide support for the future Cray interactive whole program performance analysis and tuning assistant Reveal (to be covered in a later talk)
- Two command line options control the Program Library functionality
  - -h pl = <PL_path> specifies the repository
    - -hpl=./PL.1 tells the compiler to either update the Program Library "./PL.1" if it exists, or create it if it does not exist
    - <PL_path> should specify a single location to be used for entire application build. If a makefile changes directories during a build, an absolute path might be necessary
  - -h wp enables whole-program mode
  - Both options must be specified on all compilation command lines as well as on the link line

Other Common Optimizations

- Unrolling: -hunrollN where N=0,1,2
  - By default, the compiler attempts to unroll all loops (N=2), unless the `NOUNROLL` directive is specified for a loop
- Vectorization: -hvectorN where N=0...3
  - Specify the level of automatic vectorizing to be performed (default is N=2). Vectorization results in significant performance improvements with a small increase in object code size
- Aggressive optimization: -h [no]aggress
  - Causes the compiler to treat a subroutine, function, or main program as a single optimization region. Doing so can improve the optimization of large program units but also increases compile time and size. Default is noaggress
- Cache optimization: -h cacheN where N=0...3
  - Specifies the levels of automatic cache management to perform. Symbols are placed in the cache when the possibility of cache reuse exists. Default value is N=2
- Loop trips: -h loop_trips=[tiny|small|medium|large|huge]
  - Specifies runtime loop trip counts for all loops in a compiled source file. This information is used to better tune optimizations to the runtime characteristics of the application

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 Flag Comparison

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
Inline 32-bit operations	No	No	No	Yes	Yes

Why can results from CCE be different

- We do expect application source to conform with 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

Concerning 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:

FASTER

  - intolerant: has the highest probability of repeatable results, but also has the highest performance penalty
  - rigorous: compromise between intolerant and strict
  - 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

Recommendation 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

Diagnostics Flags

- -Rb or -h bounds
  - Fortran: Enables checking of array bounds at runtime
- -eo or -hdisplay_opt
  - Display the compiler optimization settings currently in force
- -T
  - Disables the compiler but displays all options currently in effect
- -h zero
  - Initializes all undefined local stack variables to 0 (zero). Disabled by default.
- -ez
  - Initialize all memory allocated by Fortran ALLOCATE statements to zero. Disabled by default.

Memory Allocator

-hsystem_alloc, -htcmalloc

- This option specifies the malloc implementation to be used.
  By default -htcmalloc option is used for Fortran. The -hsystem_alloc causes the compiler to use the native malloc implementation provided by the OS and this is the default for C/C++.
- This is a link-time option
- Deprecated in CCE 17+

-h[no]heap_allocate

- This option forces all variable-size local arrays and temporary arrays to be allocated on the heap. Default is noheap_allocate.

Loopmark

- This is a feature to generate a listing file containing compiler loop annotations (-h list=m/a)

```
%%%    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: 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: loopmark messages

```
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

  Example:

```
% ftn bad.f90
 do l = 0.0,9.0
         ^
ftn-1569 crayftn: WARNING $MAIN, File = bad.f90, Line = 1, Column = 10
  A DO loop variable or expression of type default real or double precision real is a deleted
feature of the Fortran standard.
% explain ftn-1569

Warning : A DO loop variable or expression of type default real or double
precision real is a deleted feature of the Fortran standard.

A real or double precision real variable or DO loop expression is a deleted
feature.

A DO variable or expression of type integer should be used in place of the DO
loop expression or variable that is of type default real or double precision
real.
```

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

Error Message Example

```
program iread
  character(len=:), allocatable :: var
  integer i(2)
  i=0 ; var='20’
  read(var,"(i10:/i10)",iostat=iostat)i
  if(iostat<0) print *,"read failed, iostat=",iostat
  print *,i
  read(var,"(i10:/i10)")i
end program iread
```

```
 % ./iread
 read failed, iostat= -4005
 20,  0

lib-4005 : UNRECOVERABLE library error
  A READ operation on an internal file tried to read past the end-of-file.

Encountered during a sequential formatted READ from an internal file (character variable)
Aborted
```

Compiler Message System

```
% explain lib-4005

A READ operation on an internal file tried to read past the end-of-file.

A Fortran READ operation tried to read beyond the end of the internal file,
and neither an END nor an IOSTAT specifier was included on the internal READ
statement.

Either 1) add an END=s specifier (s is a statement label) and/or an IOSTAT=i
specifier (i is an integer variable) to the READ statement, or 2) modify the
program so that it does not read beyond the end of the internal file.

Because this is an end-of-file condition, the negative of this error number
is returned in the IOSTAT variable, if specified.

For more information, see the description of internal records and files in
your Fortran reference manual.

The error class is UNRECOVERABLE (issued by the run time library).
```

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

Compiler Message System

- -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*
  - 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*[,...]
  - 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
- !dir$ optimize

More info: https://support.hpe.com/hpesc/public/docDisplay?docId=dp00003391en_us

Macros

- Cray compilers define the following macros:
  - Fortran: **`_CRAYFTN`**
  - C/C++ (clang): **`__cray__`**

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

- Need to set FILENV envar to point to an accessible file

How to handle byte-swapped files with assign

Explicit usage of assign (no need to recompile)

- Can control which files are byte-swapped

```
export FILENV=.assign # store assign info in the file .assign
assign –R # remove all previous flags
assign -N swap_endian f:aof # swap_endian applies only to aof file
srun a.out
```
(or equivalent launch command)

Link the application with –hbyteswapio (need to recompile)

- Forces byte-swapping of all input and output files for direct and sequential unformatted I/O
- This is equivalent to setting (i.e. no recompilation needed)

```
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)

HPE Cray Programming Environment
C and C++ Compilers

HPE Cray Compilation Environment (CCE): C and C++ Compilers

- C and C++ compilers are based on Clang/LLVM
  - CCE major versioning follows LLVM versioning (currently v16)
- All LLVM features are implemented in the CCE Clang
  - C17/C++17, partially C++20
  - OpenMP 4.5, partially OpenMP 5.0/5.1
  - The latest version of the full documentation for the Clang compiler is provided at https://releases.llvm.org/16.0.0/tools/clang/docs/ReleaseNotes.html
- Some features differs from the LLVM source
  - We specifically focus on those
- CCE Clang supports compiling the C, C++, and UPC languages and the OpenMP parallel programming model for targets available on supported systems
- The CCE Clang C and C++ compilers should be invoked via **`cc`** and **`CC`** as usual
  - This will set the target based on the loaded **`craype-`** arch module and link with the usual Cray libraries, including the Cray-optimized math functions, **`memcpy`**, and OpenMP runtime

➔ https://support.hpe.com/hpesc/public/docDisplay?docId=dp00003396en_us&docLocale=en_US

General Enhancements

- CCE Clang/LLVM provides better optimized code and provide additional features with respect to the standard Clang/LLVM
  - In general, performance improvements are enabled by default, but features must be requested by an option
- The compiler predefines the macro `__cray__` in addition to all the usual Clang predefined macros
- Flags `-fcray`, `-fno-cray` can be used to enable (default) or disable Cray enhancements
  - `-fno-cray` is intended to help diagnose whether a problem is caused by a Cray enhancement or is present in the base Clang/LLVM distribution. Either way, the problem should be reported to Cray to receive the fastest response.

Performance Options

- Standard optimisation options
  - Clang does not apply optimizations unless they are requested
    - O0 Means "no optimization": this level compiles the fastest and generates the most debuggable code
    - O1 Somewhere between -O0 and -O2
    - O2 Moderate level of optimization which enables most optimizations
    - O3 Like -O2, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster)
- Aggressive optimizations
  - Recommended only for applications that are not sensitive to floating-point optimizations (otherwise use –O3)
    - **`Ofast`** enables all the optimizations from -O3 along with other aggressive optimizations that may violate strict compliance with language standards
    - **`flto`** enables link time optimizations
    - **`fast`** implies **`–Ofast`** and **`–flto`**
- **`-fcray-mallopt, -fno-cray-mallopt`**
  - Optimize malloc by using Cray's custom mallopt parameters, which for most programs improves performance but may cause higher memory usage. This is a link-time option. The default is `-fcray-mallopt`.

Floating-point Math Optimizations

- **`-ffp=`***level*
  - Select a level for Cray floating-point math optimizations and math library functions
  - Requesting the lowest level, **`-ffp=0`**, will generate code with the highest precision and grants the compiler minimal freedom to optimize floating-point operations, whereas requesting the highest level, **`-ffp=4`**, will grant the compiler maximal freedom to aggressively optimize but likely will result in lower precision
  - Requesting levels 1 through 4 will flush denormals to zero and imply **`-funsafe-math-optimizations`** and **`-fno-math-errno`**; if those options are subsequently changed, then this option may not work as expected
  - With **`-fcray`**, **`-ffp=3`** is implied by **`-ffast-math`** or **`-Ofast`**
  - Using **`-ffp=0`** will prevent the use of Cray math libraries and disable all Cray floating-point optimizations.

Loop Optimizations

- **`-flocal-restrict, -fno-local-restrict`**
  - Honor restrict-qualified pointers declared in a block scope by assuming that they do not alias with other restrict-qualified pointers declared in the same block scope. The default is -flocal-restrict.

- **`-floop-trips=`***scale*
  - Optimize assuming loops with statically unknown trip counts have trip counts at the scale of scale.
  - A valid value for scale is **`huge`**, which assume loops have trip counts large enough such that referenced data will not fit in the cache

Feature Options

- **`-fsave-decompile`**
  - Generate decompile (.dc) and IR (.ll) files prior to optimization, vectorization, and code generation, as well as after LTO. A decompile is a higher-level presentation of the IR that looks similar to C source code, but cannot be compiled. Use the decompile to gain insight about restructuring and optimization changes made by the compiler.
- **`-fsave-loopmark`**
  - Generate a loopmark listing file (.lst) that shows which optimizations were applied to which parts of the source code
- **`-finstrument-loops`**
  - Instrument loops to gather profile data to use with CrayPAT
- **`-finstrument-openmp`**
  - Turns the insertion of the CrayPat OpenMP and accelerator tracing calls on and off.

Loopmark Example

- **`-fsave-loopmark`**
  - Compiler generates an <source file name>.lst file
  - Contains annotated listing of your source code with letter indicating important optimizations

```
Legend:

  A - recognized idiom
  D - deleted loop
  I - callee inlined here
  L - interleaved loop
  M - multithreaded
  P - peeled loop
  S - distributed loop
  U - completely unrolled loop
  u - partially unrolled loop
  V - vectorized loop
  X - loop interchanged with enclosing loop
  Z - versioned loop for LICM
  + - additional messages below
```

Loopmark Example

```
51.                       // Multiplying matrix a and b and storing in array mult.
52.      0----------<     for(i = 0; i < r1; ++i)
53. +    0 1--------<         for(j = 0; j < c2; ++j)
54. +    0 1 Vu-----<             for(k = 0; k < c1; ++k)
55.      0 1 Vu                   {
56. +    0 1 Vu                       mult[i][j] += a[i][k] * b[k][j];
57.      0 1 Vu--->>>             }

54. loop not distributed: memory operations are safe for vectorization
54. loop not distributed: use -Rpass-analysis=loop-distribute for more info
54. sinking zext
54. the cost-model indicates that interleaving is not beneficial
54. unrolled loop by a factor of 4 with run-time trip count
54. vectorized loop (vectorization width: 8, interleaved count: 1)
```

Additional Information Resources

- When the CCE module is loaded, the following online help is available
  - **`man craycc and crayCC`** return CCE specific information along with information on some clang options
  - **`man clang`** returns the same
  - **`clang --help`** returns a summary of the command line options and arguments
- Please, always refer to the man page for the latest updates

- CLANG online documentation (v16)
  - https://releases.llvm.org/16.0.0/tools/clang/docs/ReleaseNotes.html

Other Compilers

GNU, Intel and AMD compilers...

- Many optimizations and features provided by CCE are available in GNU, AMD, and Intel 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

AMD Compiler

- Two AMD compilers:
  - AOCC, namely “AMD Optimizing C/C++ Compiler”: AMD CPU compiler
    - PrgEnv-aocc
    - AOCC v3.2 based on LLVM 13.0
  - AMD: AMD compiler with GPU support
    - PrgEnv-amd
    - Based on the LLVM 14.0 bundled in the ROCm 5.2.3 module
- Support Flang as the default Fortran front-end compiler
  - Improved Flang Fortran front-end added with F2008 features and bug fixes
- Highly optimized C, C++ and Fortran compiler for x86 targets especially for Zen-based AMD processors
  - -O2 optimizations (default) provide good performance
    - More specific optimizations with –O3 proved by AMD when compared to the base LLVM version, eg. handling of indirect calls, advanced vectorization etc

Recommended Compiler Optimization Levels (AOCC/AMD)

- AOCC/AMD compiler
  - Use **`-O3`** to enable specific optimizations introduced by AMD on top of the standard LLVM
  - **`-ffast-math`** and **`-freciprocal-math`** may give some extra performance (but verify results)
  - **`-funroll-loops`** benefit most applications
    - More aggressive loop optimizations: **`-enable-loop-versioning-licm`**, **`-enable-partial-unswitch`**, **`-unroll-aggressive`**
  - Other aggressive specific optimizations for vectorization
    - **`-mllvm -mllvm -enable-strided-vectorization`**: enables effective use of gather and scatter kind of instruction patterns

Recommended Compiler Optimization Levels (Intel)

- 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

Recommended Compiler Optimization Levels (GNU)

- GNU compiler
  - 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

Cray, AOCC/AMD, Intel, and GNU Compiler Flags

Feature	Cray	AOCC/AMD	Intel	GNU
Listing	-fsave-loopmark (C/C++) -hlist=a (ftn)	-Rpass=<value>	-opt-report3	-fdump-tree-all
Free format (ftn)	-f free	-ffree-form	-free	-ffree-form
Vectorization	By default at -O1 and above	By default at -O2 and above	By default at -O2 and above	By default at -O3 or using -ftree-vectorize
Inter-Procedural Optimization	-flto (C/C++) -hwp (ftn)	-flto	-ipo	-flto (note: link-time optimization)
Floating-point optimizations	-ffp=N, N=0...4 (C/C++) -hfpN, N=0...4 (ftn)	-ffp-model= [precise| strict|fast]	-fp-model [fast|fast=2|precise| except|strict]	-f[no-]fast-math or -funsafe-math-optimizations
Suggested Optimization	(default)	-O3	-O2	-O2 -mavx2 -mfma -ftree-vectorize -ffast-math -funroll-loops
Aggressive Optimization	-Ofast -ffp=3 (C/C++) -O3 -hfp3 (ftn)	-Ofast	-fast	-Ofast -mavx2 -mfma -funroll-loops
OpenMP recognition	-fopenmp	-fopenmp	-fopenmp	-fopenmp
Variable sizes (ftn)	-s real64 -s integer64	-fdefault-real-8 -fdefault-integer-8	-real-size 64 -integer-size 64	-freal-4-real-8 -finteger-4-integer-8

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
  - `-fdump-tree-all`
- AOCC/AMD
  - Based on LLVM flags to emit optimization reports
    - When the pass makes a transformation: `-Rpass=<value>`
    - When the pass fails to make a transformation: `-Rpass-missed=<value>`
    - When the pass determines whether or not to make a transformation: `-Rpass-analysis=<value>`
  - `<value>` can be any valid POSIX regular expression
    - Standard optimizations: inlining, vectorization, loop optimizations
    - Eg. `-Rpass=inline` to have remarks on inlined function
    - `-Rpass=.*` to request a report from every optimization pass

Compiler Suites for GPUs

ROCm Module

- The ROCm module sets the environment to use ROCm

```
> module av rocm
-------------------------- HPE-Cray PE modules ---------------------------
rocm/5.2.3 (D)
> module load rocm/5.2.3
```

- The module add paths to LD_LIBRARY_PATH and set ROCM_PATH env variable

```
> echo $ROCM_PATH
/opt/rocm
> echo $LD_LIBRARY_PATH
/opt/rocm/lib64:/opt/rocm/lib:/opt/rocm/rocprofiler/lib:/opt/rocm/rocprofiler/
tool:/opt/rocm/roctracer/lib:/opt/rocm/roctracer/tool:/opt/rocm/hip/lib:…
> echo $PATH
/opt/rocm/bin:/opt/rocm/rocprofiler/bin:/opt/rocm/hip/bin:…
```

Newer ROCm Module (not part of the PE)

- Samuel Antao (AMD) is providing ROCm modules
  - ➢ `module use /pfs/lustrep2/projappl/project_462000125/samantao-public/mymodules`
  - ➢ `module spider rocm`

```
…
rocm/5.4.3
rocm/5.5.0
rocm/5.5.3
rocm/5.6.0
rocm/5.7.1
…
```

  - ➢ `module load rocm/5.6.0`
  - ➢ `hipcc --version`

```
HIP version: 5.6.31061-8c743ae5d
AMD clang version 16.0.0 (https://github.com/RadeonOpenCompute/llvm-project roc-5.6.0
23243 be997b2f3651a41597d7a41441fff8ade4ac59ac)
```

- New ROCm modules provided by LUST via Easybuild
  - https://lumi-supercomputer.github.io/LUMI-EasyBuild-docs/r/rocm/

- Note that the driver install on the system is still based on ROCM 5.2.3
  - Mostly of the functionality are supposed to work with newer ROCM versions
  - Remember that it is not officially supported on LUMI!

Build for GPUs: Cray Compiler

```
> module load PrgEnv-cray
> module load craype-x86-trento
> module load craype-accel-amd-gfx90a
> module load rocm/5.2.3
```

- Load the ROCm module as well as the accelerator module for the target GPU

```
## Fortran OpenACC (enabled by default)
> ftn -hacc saxpy_acc_mpi.f90 -o saxpy_acc_mpi.x
## Fortran OpenMP
> ftn -fopenmp gemv-omp-target-many-matrices.f90 -o gemv-omp-target-many-matrices.x
## C/C++ OpenMP
> CC -fopenmp -DOMP main.cpp omp/OMPStream.cpp -I. -Iomp -DOMP_TARGET_GPU –o omp.x
## C/C++ HIP
> CC -x hip -I. -Ihip -std=c++11 -DHIP main.cpp hip/HIPStream.cpp –o hip.x
```

- No OpenACC support for C/C++
- Check `CRAY_ACC_DEBUG` in `man intro_openacc` and `man intro_openmp`
- Generate compiler listings: `-hlist=a` for Fortran and `-fsave-loopmark` for C/C++

Build for GPUs: AMD Compiler

```
> module load PrgEnv-amd
> module load craype-x86-trento
> module load craype-accel-amd-gfx90a
> module load rocm/5.2.3
```

- Load the ROCm module as well as the accelerator module for the target GPU

```
## C/C++ OpenMP
> CC -fopenmp -DOMP main.cpp omp/OMPStream.cpp -I. -Iomp -DOMP_TARGET_GPU –o omp.x
## C/C++ HIP
> CC -x hip -I. -Ihip -std=c++11 -DHIP main.cpp hip/HIPStream.cpp –o hip.x
```

- Only C/C++ offload support

Summary – How to Compile for GPUs

- Load the `PrgEnv-xxx` module
- Load the CPU target module `craype-x86-trento`
- Load the GPU target module `craype-accel-amd-gfx90a`
- Load the ROCm module `rocm`
- Use the compiler wrappers

Mixing GPU programming models: caveats (1)

- Mixing OpenMP and HIP in the same C/C++ compilation unit
  - No OpenMP offload supported (only CPU execution), the order of the flags matters: `-fopenmp –xhip`

  ➔Use different compilation units for OpenMP offload and HIP
    - Linking with `-fopenmp` only, don't use `-xhip`

- The GNU compilers cannot be used to compile HIP code, so all HIP kernels must be separated from CPU code
  - HIP kernels must be compiled with hipcc
  - All non-HIP code must be compiled with the wrappers (ftn/cc/CC), then linking must be performed with the wrappers
  - Note about OpenMP library: GNU is using libgomp library, while HIP (LLVM) is using libomp library
    - Cannot mix the two, e.g. do not use OpenMP in the HIP code

Mixing GPU programming models: caveats (2)

- HIP code can be compiled with hipcc (see AMD presentations for more details)
  - hipcc is a wrapper around clang, coming from ROCm installation (set via PrgEnv-amd or rocm module)

```
> hipcc --version
HIP version: 5.2.21153-02187ecf
AMD clang version 14.0.0 (https://github.com/RadeonOpenCompute/llvm-project roc-5.2.3
22324 d6c88e5a78066d5d7a1e8db6c5e3e9884c6ad10e)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /opt/rocm/llvm/bin
```

  - Need to specify all offload flags by hand, i.e. `-D__HIP_PLATFORM_AMD__ --offload-arch=gfx90a`
  - All non-HIP code must be compiled with the wrappers, then linking must be performed with the wrappers
  - Possible incompatibility between HIP-clang and CCE-clang
    - Suggested to use Cray compiler for linking
  - Note about OpenMP library: CCE is using libcraymp library, while HIP (LLVM) is using libomp library
    - Cannot mix the two, e.g. do not use OpenMP in the HIP code

HIPCC Compilation with MPI Support

- Need to set CXXFLAGS, LDFFLAGS, and LIBS flags (otherwise injected by PE wrappers via modules)

<table>
<tr><th>PE Wrappers</th><th>HIPCC</th></tr>
<tr><td><pre>module load PrgEnv-cray # or PrgEnv-amd
module load craype-accel-amd-gfx90a
module load rocm/5.2.3
export CXX=“CC -xhip -craype-verbose”

export LD=“CC -craype-verbose”</pre></td><td><pre>module load PrgEnv-cray # or PrgEnv-amd
module load craype-accel-amd-gfx90a
module load rocm/5.2.3
export CXX=“hipcc”
export CXXFLAGS=“-D__HIP_PLATFORM_AMD__ \
       --offload-arch=gfx90a \
       -I${CRAY_MPICH_DIR}/include”
export LD=“hipcc”
export LDFLAGS=“${CXXFLAGS} \
       -L${CRAY_MPICH_DIR}/lib \
       ${PE_MPICH_GTL_DIR_amd_gfx90a}”
export LIBS="-lmpi \
       ${PE_MPICH_GTL_LIBS_amd_gfx90a}"</pre></td></tr>
</table>

- The flag `-craype-verbose` to confirm the architecture flags
- Can be extended to include other libraries provided by modules (e.g. cray-libsci)
- Same considerations apply if you want to directly use the compiler drivers (e.g. crayCC)

Summary – Compilers

- Multiple compiler environments are available
  - All of them accessed through the wrappers ftn, cc and CC – just do module swap to change a compiler!
  - Load the proper modules
- 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

Parallel Programming Models

Parallel Programming Models: Short Overview

- Multi-processes
  - Suitable for distributed-memory systems (e.g. on multi-nodes)
  - Message Passing Model: MPI
  - Partitioned Global Address Space (PGAS): SHMEM, Coarrays, UPC, Chapel, GASPI

- Multi-threaded
  - Suitable for shared-memory systems (e.g. on multi-cores)
  - Direct parallelism: Pthreads, std::threads
  - Directive based: OpenMP
  - Programming languages: C++ Parallel STL
  - Via parallel libraries: HPX, TBB, C++ Parallel STL Executors

- Accelerator execution (via GPUs)
  - Code regions are offloaded from a host CPU to be computed on an accelerator

- Hybrid model: mixing all the above

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

(Some) GPU Programming Models Portability

Some Comparison References

- *An Evaluative Comparison of Performance Portability across GPU Programming Models*, J. H. Davis et al, 2024 (https://arxiv.org/pdf/2402.08950.pdf)
- *Many Cores, Many Models: GPU Programming Model vs. Vendor Compatibility Overview*, P3HPC Workshop 2023 (https://arxiv.org/abs/2309.05445)
- *Heterogeneous Programming for the Homogeneous Majority*, T. Deakin et al, 2022
  - https://research-information.bris.ac.uk/ws/portalfiles/portal/334944980/p3hpc_sc22.pdf
- *Tracking Performance Portability on the Yellow Brick Road to Exascale*, T. Deakin et al, 2021
  - https://research-information.bris.ac.uk/ws/portalfiles/portal/250082095/p3hpc_sc20_review_submission_3.pdf
- *A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application*, A. Poenaru et al, 2021
  - https://research-information.bris.ac.uk/ws/portalfiles/portal/284239108/A_Performance_Analysis_of_Modern_Parallel_Programming_Models.pdf
- *A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware*, M. Breyer et al, 2022 (https://dl.acm.org/doi/10.1145/3529538.3529980)

Questions?