C and C++ Reference Manual (S-2179-84)

Contents

The Cray Compiling Environment

The Cray Compiling Environment (CCE) contains both the Cray C and C++ compilers.

Log in either to a login node or a standalone application development system and use the Cray XE, Cray XK, Cray XC or Cray CS series Programming Environment, and related products to create an application which executes on compute nodes. For further information about login nodes and the user environment, see the *Cray Programming Environment User's Guide*.

Throughout this manual, the differences between the Cray C and C++ compilers are noted when appropriate. When there is no difference, the phrase *the compiler* refers to both compilers. All compiler command options apply to Cray C and C++ unless noted.

The Cray C Compiler

The Cray C compiler consists of a preprocessor, a language parser, an optimizer, and a code generator. Invoke the Cray C compiler with the cc compiler driver command. This command and its options are also described in the craycc(1) man page.

The Cray C++ Compiler

The Cray C++ compiler consists of a preprocessor, a language parser, an optimizer, and a code generator. Invoke the Cray C++ compiler with the CC compiler driver command. The CC command is described in and the crayCC(1) man page. See *Command Line Examples*.

Standards

The Cray C compiler conforms to the International Organization of Standards (ISO) standard ISO/IEC 9899:1999 (C99).

The Cray C++ compiler accepts the C++ language as defined by the *ISO/IEC 14882:2003* standard, with some exceptions. The exceptions are noted in *Cray C and C++ Dialect Use*.

The Cray C++ compiler also contains support for the C++11 version of the C++ language as defined by the ISO/IEC 14882:2011 standard, with the exception of the “alignas” type specifier. This support can be enabled with the `-hstd=c++11` command-line flag.

The Cray C compiler supports the UPC 1.3 specification. The UPC 1.3 specification is discussed on the UPC specification website, *http://code.google.com/p/upc-specification*. This support can be enabled with the `-hupc` command-line flag.

Related Publications

The following documents contain additional information that may be helpful:

- `cc(1)` compiler driver man page
- `craycc(1)` man page for the Cray C compiler

- `CC(1)` compiler driver man page
- `crayCC(1)` man page for the Cray C++ compiler
- `intro_directives(7)` man page
- `intro_openacc(7)` man page
- `intro_pgas(7)` man page
- `ftn(1)` compiler driver man page
- `crayftn(1)` man page for the Cray Fortran compiler
- `aprun(1)` man page
- *Cray Fortran Reference Manual*
- *Cray Programming Environments Installation Guide*
- *Cray Programming Environment User's Guide*
- *Using Cray Performance Measurement and Analysis Tools*

Invoke the C and C++ Compilers

The following commands invoke the compilers:

- `CC` invokes the Cray C++ compiler.
- `cc` invokes the Cray C compiler.
- `cpp`, the C language preprocessor, is not part of CCE. The `cpp` command resolves to the GNU cpp command and does not predefine any Cray compiler-specific macros. If the predefinition of the Cray compiler-specific macros is required, then use the `cc` or `CC` command to do the source preprocessing using the `-E` or `-P` option.

A successful compilation creates an executable file, named `a.out` by default, that reflects the contents of the source code and any referenced library functions. Use the `aprun` command to run the executable on the compute nodes.

Simple C compiler invocation and application execution

```
% cc mysource.c; aprun -n 64 ./a.out
```

By default, the CC and cc commands automatically call the linker, which creates an executable file. If only one source file is specified, the object file (`*.o`) is deleted. If more than one source file is specified, the object files are retained.

Simple C compiler invocation using multiple source files

The following command creates and retains object files `file1.o`, `file2.o`, and `file3.o`, and creates the executable file a.out.

```
% cc file1.c file2.c file3.c
```

Simple C compiler invocation using single source file

The following command creates file.o and a.out; file.o is not retained.

```
% cc file1.c
```

CC Command

`CC` [`-c`] [`-C`] [`-D` *macro*[=*def*]] [`-E`] [`-g`] [`-G` *level*] [`-h` *arg*] [`-I` *incldir*] [`-K trap=opt,opt ...`] [`-l` *libfile*] [`-L` *ldir*] [`-M`] [`-nostdinc`] [`-o` *outfile*] [`-O` *level*] [`-P`] [`-S`] [`-U` *macro*] [`-V`] [`-W`*phase*,*"opt ..."*] [`-X` *npes*] [`-Y`*phase*, *dirname*] [`-#`] [`-##`] [`-###`] *sourcefile . . .*

The `CC` command invokes the Cray C++ compiler and accepts C++ source files with the following suffixes:

```
.c
.C
.i
.c++
```

```
.C++
.cc
.cxx
.Cxx
.CXX
.CC
```

The `.i` files are created when the preprocessing compiler command option (`-P`) is used. The CC command also accepts object files with the `.o` suffix, library files with the `.a` suffix, and assembler source files with the `.s` suffix.

cc Command

`cc` [`-c`] [`-C`] [`-D` *macro*[=*def*]] [`-E`] [`-g`] [`-G` *level*] [`-h`*arg*] [`-I` *incldir*] [`-K trap=opt[,opt] ...`] [`-l` *libfile*] [`-L` *ldir*] [`-M`] [`-nostdinc`] [`-o` *outfile*] [`-O` *level*] [`-P`] [`-S`] [`-U` *macro*] [`-V`] [`-W`*phase*,*"opt ..."*] [`-X` *npes*] [`-Y`*phase*,*dirname*] [`-#`] [`-##`] [`-###`] *sourcefile . . .*

The cc command invokes the Cray C compiler. The `cc` command accepts C source files that have the `.c` and `.i` suffixes; object files with the `.o` suffix; library files with the `.a` suffix; and assembler source files with the `.s` suffix.

File Types Used or Created by the Compiler

The compiler uses and creates several types of files during processing:

a.out	Default name of the executable output file. Use the compiler driver command line option `-o` to specify an executable name other than `a.out`.
.c, .C, .c++, .C++, .cc, .cxx, .Cxx, .CXX, .CC	C++ source files
.c	C source files
.i	Files containing output from the source preprocessor
.o	Relocatable object code. During compilation, these relocatable object files are saved in the current directory automatically. If using CrayPat to conduct performance analysis experiments, the object files created during compilation must be kept in order to preserve source-to-executable function mapping. To do so, use the `-h keepfiles` option.
.a	Library files containing external references
.s	Assembly language files. Files with `.s` extensions are assembled and written to the corresponding `.o` file.

Command Line Examples

The following examples illustrate a variety of command lines for the C and C++ compiler commands:

CC -h conform myprog.C

This example compiles `myprog.C`. The `-h conform` option specifies strict conformance to the ISO C++ standard.

```
% CC -h conform myprog.C
```

cc -c -h ipa1 myprog.c subprog.c

This example compiles input files `myprog.c` and `subprog.c`. The `-c` option tells the compiler to create object files `myprog.o` and `subprog.o` but not call the linker. Option `-h ipa1` tells the compiler to inline function calls marked with the `inline_always` pragma.

```
% cc -c -h ipa1 myprog.c subprog.c
```

cc -I. disc.c vend.c

This example specifies that the compiler search the current working directory, represented by a period (.), for `#include` files before searching the default `#include` file locations.

```
% cc -I. disc.c vend.c
```

cc -P -D DEBUG newprog.c

This example specifies that source file `newprog.c` be preprocessed only. Compilation and linking are suppressed. In addition, the macro `DEBUG` is defined.

```
% cc -P -D DEBUG newprog.c
```

cc -c -h report=s mydata1.c

This example compiles `mydata1.c`, creates object file `mydata1.o`, and produces a scalar optimization report to `stdout`.

```
% cc -c -h report=s mydata1.c
```

CC -h ipa5,report=if myfile.C

This example compiles `myfile.C` and tells the compiler to attempt to aggressively inline calls to functions defined within `myfile.C`. An inlining report is directed to `myfile.V`.

```
% CC -h ipa5,report=if myfile.C
```

Compile Time Environment Variables

The following environment variables are used during compilation.

CRAYOLDCPPLIB When set to a nonzero value, enables C++ code to use the following nonstandard Cray C++ headers files:

- `common.h`
- `complex.h`
- `fstream.h`
- `generic.h`
- `iomanip.h`
- `iostream.h`
- `stdiostream.h`
- `stream.h`
- `strstream.h`
- `vector.h`

To use the standard header files, the code may require modification to compile successfully. For more information, see *Cray C and C++ Dialect Use*.

Setting the `CRAYOLDCPPLIB` environment variable disables exception handling, unless compiling with the `-h exceptions` option.

CRI_CC_OPTIONS, CRI_cc_OPTIONS Specifies command line options that are applied to all compilations. Options specified by this environment variable are added following the options specified directly on the command line. This is especially useful for adding options to compilations done with build tools.

Identifies the requirements for native language, local customs, and coded character set with regard to compiler messages.

Controls the format in which compiler messages are received.

Specifies the message system catalogs that should be used.

Specifies the number of processes used for simultaneous compilations The default is 1. When more than one source file is specified on the command line, compilations may be multiprocessed by setting the environment variable `NPROC` to a value greater than 1. `NPROC` can be set to any value; however, large values can overload the system.

Run Time Environment Variables

CRAY_MALLOPT_OFF If set, then the system default `mallopt` parameters are used, instead of the compiler default parameters. For most programs, run time performance is improved by using the compiler defaults, but more memory may be used.

MALLOC_MMAP_MAX_ Specifies the maximum number of memory chunks to allocate with `mmap`. The compiler default value is 0. For most programs, run time performance is improved by using the compiler default, but more memory may be used.

MALLOC_TRIM_THRESHOLD_ Specifies the minimum size of the unused memory region at the top of the heap before the region is returned to the operating system. The compiler default value is 536870912 bytes. For most programs, run time performance is improved by using the compiler default, but more memory may be used.

PGAS_ERROR_FILE
: Specifies the location to which `libpgas` (the library which provides an interface to the internal system network) error messages are written. The default is `stderr`. If `stdout` is specified, errors will be written to standard output.

CRAYLIBS_ARCH_OVERRIDE
: Override the default Cray math library run time selection and specify the library to use by CPU architecture. The valid options are: `haswell`, `ivybridge`, `sandybridge`, `interlagos`, `abudhabi`, `interlagos-cu`, `abudhabi-cu`, `mc8`, `mc12`, `istanbul`, `shanghai`, `barcelona`, `opteron`, or `x86-64`.

  This run time environment variable can be used to specify that a lowest-common-denominator math library be used instead of the default selection, thus ensuring that identical computations produce identical results regardless of the type of compute node CPU actually used. The trade-off is that specifying an older library may affect performance on a newer CPU. For example, if `ivybridge` is specified, the code will run and produce identical results on a `haswell` compute node, but performance may be reduced.

  Default: If not set, the library specific to the type of CPU selected at run time is used.

Compiler Command Line Options

With the use of appropriate options, it is possible to direct the compiler to generate intermediate translations, including relocatable object files (`-c` option), assembly source expansions (`-S` option). In general, it is possible to save the intermediate files and reference them later on another invocation of the compiler, with other files or libraries included as necessary.

Options that are not recognized by the compiler driver are passed to the linking phase.

If the specified options conflict, the option specified last on the command line overrides the previously specified option. Exceptions to this rule are noted in the individual descriptions of the options.

There are many options that start with `-h`. Specify multiple `-h` options using commas to separate the arguments.

Multiple `-h` options

```
cc -h parse_templates, fp0
```

Conflicting options

In this example, `-h fp0` overrides `-h f1`.

```
cc -h fp1,fp0 myfile.c
```

Standard Language Conformance Options

`-h [no]conform`, `-h [no]stdc`

Default: `-h noconform`

Default: `-h nostdc`

`-h conform` specifies strict conformance to the ISO C standard or the ISO C++ standard. The `-h [no]stdc` option specifies strict conformance to the ISO C standard and does not relate to `C++`. The `-h noconform` and `-h nostdc` options specify partial conformance to the standard.

The `-h conform` option enables `-h exceptions`, `-h dep_name`, `-h parse_templates`, and `-h const_string_literals` options in Cray C++.

By default, the compiler calls the Cray mathlib versions of intrinsic functions (`abs`, `cos`, `exp`, for example) which do not set `errno` and do not raise IEEE-754 underflow exceptions. If `-hconform` is specified, the compiler calls the `stdc glibc` versions of the runtime intrinsic functions.

When `-h noc99` is used, C99 language features such as variable-length arrays (VLAs) and restricted pointers that were available as extensions previously to adoption of the C99 standard remain available.

-h [no]gnu

Default: `-h gnu`

Enables the compiler to recognize the GCC extensions to C listed in GCC C Language Extensions. For detailed descriptions of the GCC C and C++ language extensions, see *http://gcc.gnu.org/onlinedocs/*.

-h std=*language_standard*

Determines the C/C++ language standard. *`language_standard`* can be either `c99`, `c89`, `c++03`, or `c++11`. The default, if this option is not specified, is c99/ c++03. This option does not imply strict conformance to the language standard. That is controlled by the `-h conform` and `-h stdc` options.

If both `-h std=c++11` and `-h conform` are specified on a command line, `-h conform` is disabled, as specifying `c++11` implies `-h gnu` by default.

C Specific Language Conformance Options

-h [no]c99

Default: `-h c99`

This option enables or disables language features in the C99 standard. If the previous implementation of the Cray extension differed from the C99 standard, both implementations will be available when the `-h c99` option is enabled. The `-h c99` option is also required for C99 features not previously supported as extensions.

When `-h noc99` is used, C99 language features such as variable-length arrays (VLAs) and restricted pointers that were available as extensions previously to adoption of the C99 standard remain available.

-h [no]tolerant

Default: `-h notolerant`

The `-h tolerant` option allows older, non-standard C constructs. This option is useful when porting code written for previous C compilers. Errors involving comparisons or assignments of pointers and integers become warnings. The compiler generates casts so that the types agree. With `-h notolerant`, the compiler is intolerant of the older constructs.

The `-h tolerant` option causes the compiler to tolerate accessing an object with one type through a pointer to an entirely different type. For example, a pointer to a long might be used to access an object declared with type double. Such references violate the C standard and should be eliminated if possible. They can reduce the effectiveness of alias analysis and inhibit optimization.

C++ Specific Language Conformance Options

-h cfront

Default: `off`

Accept or reject constructs that were accepted by previous cfront-based compilers (such as Cray C++ 1.0) but which are not accepted in the C++ standard. The `-h anachronisms` option is implied when `-h cfront` is specified.

-h [no]parse_templates

Default: `-h noparse_templates`

Defines templates using previous versions of the Cray Standard Template Library (STL) (before Programming Environment 3.6) to compile successfully with the `-h conform` option. Use the `-h noparse_templates` option to compile existing code without having to use the Cray C++ STL. Also, the compiler defaults to this mode when the `-h dep_name` option is used. To have the compiler verify that the code uses the Cray C++ STL properly, use the `-h parse_templates` option.

-h [no]dep_name

Default: `-h nodep_name`

Enables or disables dependent name processing - the separate lookup of names in templates when the template is parsed and when it is instantiated. The `-h dep_name` option cannot be used with the `-h noparse_templates` option.

-h [no]exceptions

Default: `-h exceptions`

Enables support for exception handling. The `-h noexceptions` option issues an error whenever an exception construct, a try block, a throw expression, or a throw specification on a function declaration is encountered. If the `CRAYOLDCPPLIB` environment variable is set to a nonzero value, the default is `-h noexceptions`.

The `-h exceptions` option is enabled by `-h conform`.

-h [no]new_for_init

Default: `-h new_for_init`

Enables or disables the new scoping rules for a declaration in a *for-init-statement*. This means that the new standard-conforming rules are in effect; the entire for statement is wrapped in its own implicitly generated scope. The `-h new_for_init` option is implied by the `-h conform` option.

The result of the scoping rule:

```
  .
  .
  .
  for (int i = 0; i < n; i++)  {
    .
    .
    .
  } // scope of i ends here for -h new_for_init
  .
  .
  .
}  // scope of i ends here for -h nonew_for_init
```

-h [no]const_string_literals

Default: `-h noconst_string_literals`

Controls whether string literals are `const` (as required by the standard) or not (as was true in earlier versions of the C++ language).

-h [no]anachronisms

Default: `-h noanachronisms`

Disables or enables anachronisms in Cray C++. This option is overridden by `-h conform`.

When anachronisms are enabled, the following anachronisms are accepted:

- `overload` is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized by using the default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array can be specified in an array delete operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name can be omitted in a base class initializer if there is only one immediate base class.
- Assignment to the `this` pointer in constructors and destructors is allowed. This is only allowed if anachronisms are enabled and the assignment to this configuration parameter is enabled.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a non-nested class name if no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-`const` type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-`const` class type may be initialized from an rvalue of the class type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and can participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when checking for compatibility, therefore, the following statements declare the overloading of two functions named `f`:

  ```
  int f(int);
  int f(x) char x; { return x; }
  ```

  In C, this code is legal, but has a different meaning. A tentative declaration of `f` is followed by its definition.

Miscellaneous C++ Specific Options

-h forcevtbl

Forces the definition of virtual function tables in cases where the heuristic methods used by the compiler to decide on definition of virtual function tables provide no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline, non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity). The `-h forcevtbl` option differs from the default behavior in that it does not force the definition to be local.

-h suppressvtbl

`CC` command line option that suppresses the definition of virtual function tables in cases where the heuristic methods used by the compiler to decide on definition of virtual function tables provide no guidance.

-h keep=*file*

When the `-h keep=file` option is specified, the static constructor/destructor object (.o) file is retained as *`file`*. This option is useful when linking .o files on a system that does not have a C++ compiler. The use of this option requires that the `main` function must be compiled by C++ and the static constructor/destructor function must be included in the link. With these precautions, mixed object files (files with .o suffixes) from C and C++ compilations can be linked into executables by using the linker command instead of the `CC` command.

General Language Options

-h [no]calchars

Default: `-h nocalchars`

The `-h calchars` option allows the use of the `$` character in identifier names. This option is useful for porting code containing identifiers that include this character. With `-h nocalchars`, this character is not allowed in identifier names. Use this option with extreme care, because identifiers with this character are within CNL name space and are included in many library identifiers, internal compiler labels, objects, and functions. Prevent conflicts between identifiers within CNL name space and the code; any such conflict is an error.

-h restrict=*args*

The `-h restrict=`*`args`* option globally tells the compiler to treat certain classes of pointers as restricted pointers. Use this option to enhance optimizations, including vectorization.

Classes of affected pointers are determined by the value contained in args, as follows:

Table 1. -h restrict option arguments

args	Description
`a`	All pointers to object and incomplete types are considered restricted pointers, regardless of where they appear in the source code. This

args	**Description**
	includes pointers in class, struct, and union declarations, type casts, function prototypes, and so on. Do not specify `restrict=a` if, during execution of any function, an object is modified and that object is referenced through either two different pointers or through the declared name of the object and a pointer. Undefined behavior may result.
`f`	All function parameters that are pointers to objects or incomplete types can be treated as restricted pointers. Do not specify `restrict=f` if, during execution of any function, an object is modified and that object is referenced through either two different pointer function parameters or through the declared name of the object and a pointer function parameter. Undefined behavior may result.
`t`	All parameters that are this pointers can be treated as restricted pointers (Cray C++ only). Do not specify `restrict=t` if, during execution of any function, an object is modified and that object is referenced through the declared name of the object and a this pointer. Undefined behavior may result.

The args arguments tell the compiler to assume that, in the current compilation unit, each pointer (`=a`), each pointer that is a function parameter (`=f`), or each this pointer (`=t`) points to a unique object. This assumption eliminates those pointers as sources of potential aliasing, and may allow additional vectorization or other optimizations. These options cause only data dependencies from pointer aliasing to be ignored, rather than all data dependencies.

The arguments make assertions about the program that, if incorrect, can introduce undefined behavior. Do not use `-h restrict=a` if, during the execution of any function, an object is modified and that object is referenced through either of the following: Two different pointers The declared name of the object and a pointer The `-h restrict=f` and `-h restrict=t` options are subject to the analogous restriction, with "function parameter pointer" replacing "pointer."

`-h [no]signedshifts`

Default: `-h nosignedshifts`

For the expression `e1 >> e2`, where `e1` has a signed type, when `-h signedshifts` is in effect, the vacated bits are filled with the sign bit of `e1`. When `-h nosignedshifts` is in effect, the vacated bits are filled with zeros, identical to the behavior when `e1` has an unsigned type. Also, see *Integers* about the effects of this option when shifting integers.

`-h list=list_opt`

The values for `list_opt`:

a Use all list options.

d Decompile (translate) the intermediate representation of the compiler into listings that resemble the format of the source code. This is performed twice, resulting in two output files, at different points during the

optimization process. Use these files to examine the restructuring and optimization changes made by the compiler, which can lead to insights about changes made to the source code to improve its performance. The compiler produces two decompilation listing files with these extensions per specified source file: `.opt` and `.cg`. The compiler generates the `.opt` file after applying most high-level loop nest transformations to the code. The code structure of this listing most resembles the source code and is readable by most users. In some cases, because of optimizations, the structure of the loops and conditionals will be significantly different than the structure in the source file. The `.cg` file contains a much lower level of decompilation. It is quite close to what will be produced as assembly output. This version displays the intermediate text after all vector translation and other optimizations have been performed. An initmate knowledge of the hardware architecture of the system is helpful to understanding this listing. The `.opt` and `.cg` files are intended as a tool for performance analysis and are not vaild source code. The format and contents of the files can be expected to change from release to release.

e Expand included files. This option is off by default.

i Intersperse optimization messages within the loopmark listing rather than at the end.

m Produce loopmark listing. To provide a more complete report, this option automatically enables the `-0 negmsgs` option to show why loops were not optimized. If this information is not required, use the `-0 nonegmsgs` option on the same command line. Loopmark information will not be displayed if the `-d B` option has been specified.

o Show options used by the compiler at compile time in listing.

s Create a complete source listing (include files not expanded).

General Optimization Options

-h [no]add_paren

Default: `-h noadd_paren`

The `-h [no]add_paren` option automatically adds parenthesis to select associative operations (+,-,*) to encourage left to right evaluation of floating point and complex expressions. For more information, see the crayftn(1) man page. Left to right evaluation is not required by the language standards, but some applications may expect it.

-h [no]aggress

Default: `noaggress`

Provides greater opportunity to optimize loops that would otherwise by inhibited from optimization due to an internal compiler size limitation. `noaggress` leaves this size limitation in effect. With `aggress`, internal compiler tables are expanded to accommodate larger loop bodies. This option can increase the compilation's time and memory size.

-h [no]autoprefetch

Default: `autoprefetch`

This option controls autoprefetch optimization. This option does not affect the `loop_info noprefetch` or `prefetch` directives.

`-h [no]autothread`

Default: `-h noautothread`

The `-h [no]autothread` option enables or disables autothreading.

`-h display_opt`

The `-h display_opt` option displays the compiler optimization settings currently in force.

`-h flex_mp=`*`level`*

Default: `-h flex_mp=default`

The `-h flex_mp=level` option controls 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.

The values for level are:

intolerant	Has the highest probability of repeatable results, but also the highest performance penalty.
strict	Uses some safe optimizations and yields higher performance than intolerant, with a 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.

`-h [no]func_trace`

Default: `-h nofunc_trace`

The `-h func_trace` option is for use only with CrayPat. If this option is specified, the compiler inserts CrayPat trace entry points into each function in the compiled source file. The names of the trace entry points are:

- **`__pat_tp_func_entry`**
- **`__pat_tp_func_return`**

These are resolved by CrayPat when the program is instrumented using the `pat_build` command. When the instrumented program is executed and it encounters either of these trace entry points, CrayPat captures the address of the current function and its return address.

`-h fusion`*`n`*

Default: `fusion2`

Loop fusion can improve the performance of loops, although in rare cases it may degrade performance. The *`n`* argument allows loop fusion to be turned on or off and determine where fusion should occur.

Loop fusion is disabled when *n* is set to 0.

The values for *n* are:

0 No fusion. Ignore all `fusion` directives and do not attempt to fuse other loops.

1 Attempt to fuse loops that are marked by the `fusion` directive.

2 Attempt to fuse all loops (includes array syntax implied loops), except those marked with the `nofusion` directive.

-h [no]intrinsics

Default: `-h intrinsics`

Allow the use of intrinsic hardware functions, which allow direct access to some hardware instructions or generate inline code for some functions. This option has no effect on specially handled library functions

-h [no]msgs

Default: `-h nomsgs`

The `-h msgs` option causes the compiler to write optimization messages to `stderr`.

-h [no]negmsgs

Default: `-h nonegmsgs`

The `-h negmsgs` option causes the compiler to write messages to `stderr` that indicate why optimizations such as vectorization, inlining, or cloning did not occur. The `-h negmsgs` option enables the `-h msgs` option. The `-h list=a` option enables the `-h negmsgs` option.

-h [no]omp_trace

Default: `-h noomp_trace`

The `-h omp_trace` option turns the insertion of the CrayPat OpenMP tracing calls on or off.

-h [no]overindex

Default: `nooverindex`

The `overindex` option declares that there are array subscripts that index a dimension of an array that is outside the declared bounds of that array. The `nooverindex` option declares that there are no array subscripts that index a dimension of an array that is outside the declared bounds of that array.

-h [no]pattern

Default: `pattern`

Globally enables pattern matching. When the compiler recognizes certain patterns in the source code, it replaces the construct with a call to an optimized library routine. A loop or statement that has been pattern matched and replaced with a call to a library routine is indicated with an A in the loopmark listing. The `nopattern` option globally disables pattern matching and causes the compiler to ignore the PATTERN and NOPATTERN directives.

Pattern matching is not always worthwhile. If there is a small amount of work in the pattern-matched construct, the call overhead may outweigh the time saved by using the optimized library routine. When compiling using the

default optimization settings, the compiler attempts to determine whether each given candidate for pattern matching will in fact yield improved performance.

-h pl=*program_library*

Create and use a persistent repository of compiler information specified by *program_library*. When used with `-hwp`, this option provides application-wide, cross-file, automatic inlining. The *program_library* repository is implemented as a directory and the information contained in program library is built up with each compiler invocation. Any compilation that does not have the `-hpl` option will not add information to this repository. Because of the persistence of *program_library*, it is the user's responsibility to manage it. For example, `rm -r` *program_library* might be added to the make clean target in an application makefile. Because *program_library* is a directory, use `rm -r` to remove it. If an application makefile works by creating files in multiple directories during a single build, the program_library should be an absolute path, otherwise multiple and incomplete program library repositories will be created. For example, avoid `-hpl=./PL.1` and use `-hpl=/fullpath/builddir/PL.1` instead.

-h profile_generate

The `-h profile_generate` option directs that the source code be instrumented for gathering profile information. The compiler inserts calls and data-gathering instructions to allow CrayPat to gather information about the loops in a compilation unit. If using this option, CrayPat must be run on the resulting executable so the CrayPat data-gathering routines are linked in. For information about CrayPat and profile information, see the *Using Cray Performance Measurement and Analysis Tools* guide.

-h thread*n*

Default: `-h thread2`

The `-h thread`*n* option controls the optimization of both OpenMP and automatic threading.

The values for *n*:

0 No autothreading or OMP threading. The `thread0` option is similar to `-h noomp` , but `-h noomp` disables OpenMP only and does not affect autothreading.

1 Specifies strict compliance with the OpenMP standard for directive compilation. Strict compliance is defined as no extra optimizations in or around OpenMP constructs. In other words, the compiler performs only the requested optimizations.

2 OpenMP parallel regions are subjected to some optimizations; that is, some parallel region expansion. Parallel region expansion is an optimization that merges two adjacent parallel regions in a compilation unit into a single parallel region.

3 Full optimization: loop restructuring, including modifying iteration space for static schedules (breaking standard compliance). Reduction results may not be repeatable.

-h unroll*n*

Default: `unroll2`

The `-h unroll`*n* option globally controls loop unrolling and changes the assertiveness of the `UNROLL` directive. By default, the compiler attempts to unroll loops, unless the `NOUNROLL` directive is specified of a loop. Generally, unrolling loops increases single processor performance at the cost of increased compile time and code size. Loop unrolling is disabled when the scalar level is set to 0.

The values for *n*:

0 No unrolling (ignore all unroll pragmas and do not attempt to unroll other loops).

1 Attempt to unroll loops that are marked by the **unroll** pragma.

2 Unroll loops when performance is expected to improve. Loops marked with the **unroll** or **nounroll** pragma override automatic unrolling.

-h wp

Default: off

Enables the whole program mode.

This option causes the compiler backend (IPA, optimizer, codegenerator) to be invoked at application link time, enabling whole program automatic inlining/cloning and future whole program interprocedural analysis (IPA) optimizations. Since the `-hwp` option provides automatic application-wide inlining, the `-Oipafrom` option is no longer needed for cross-file inlining and using these two options together is not permitted. Requires that `-h pl=`*program_library* is also specified.

The options `-hpl=` and `-hwp` should be specified on all compiler invocations and on the compiler link invocation. Since `-hwp` delays the compiler optimization step until link time, `-c` compiles will take less time and the link step will take longer. Normally, this is just a time shift from one build phase to another with roughly the same overall compile time. In some cases increased inlining may cause an increase in overall compile time. Using `-hwp` allows the compiler backend to be invoked in parallel during a build. Setting the environment variable `NPROC` controls the number of concurrent compiler backend invocations and this parallelism may reduce overall compile time.

-O*level*

Default: `-O2`

Specify a general level of optimization that includes vectorization, scalar optimization, cache management, and inlining. Generally, as the optimization level increases, compilation time increases and execution time decreases.

The `-O`*level* specifications do not directly correspond to the numeric optimization levels for scalar optimization, vectorization, and inlining. For example, specifying `-O 3` does not necessarily enable `-h vector3`. Cray reserves the right to alter the specific optimizations performed at these levels from release to release. Use the `-h display_opt` option to display the optimization options used during compilation.

The values for *level*:

0 Disable optimization, including floating point optimizations. Low compile time, small compile size, no global scalar optimization. Vectorize most array syntax statements, but disable all other vectorizations. Implies `-h fp0`.

1 Conservative optimization: moderate compile time and size, global scalar optimizations, and loop nest restructuring. Results may differ from the results obtained when `-O 0` is specified because of operator reassociation. No optimizations will be performed that might create false exceptions. Only array syntax statements and inner loops are vectorized and the system does not perform some vector reductions. User tasking is enabled, so OpenMP directives are recognized.

2 Moderate optimization: moderate compile time and size, global scalar optimizations, pattern matching, and loop nest restructuring. Results may differ from results obtained when `-O 1` is specified because of vector

reductions. The `-O 2` option enables automatic vectorization of array syntax and entire loop nests. This is the default level of optimization.

3 Aggressive optimization: potentially larger compile time and size, global scalar optimizations, possible loop nest restructuring, and pattern matching. The optimizations performed might create false exceptions in rare instances. Results may differ from results obtained when `-O 1` is specified because of vector reductions.

Cache Management Options

`-h cache`*`level`*

Default: `cache2`

Specifiy the levels of automatic cache management to perform. Automatic cache management can be overridden by the use of the cache directives (`cache` , `cache_nt` , and `loop_info`).

The values for *`level`*:

0 Cache blocking (including directive-based blocking) is turned off. This level is compatible with all scalar and vector optimization levels.

1 Conservative automatic cache management. Characteristics include moderate compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the predicted cache footprint of the symbol in isolation is small enough to experience the reuse.

2 Moderately aggressive automatic cache management. Characteristics include moderate compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the predicted state of the cache model is such that the symbol will experience the reuse.

3 Aggressive automatic cache management. Characteristics include potentially high compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the allocation of the symbol to the cache is predicted to increase the number of cache hits.

Vector Optimization Options

`-h concurrent`

The `-h concurrent` option indicates that no data dependence exists between array references of the same loop, for every loop in the file. This can be useful for vectorization optimizations. Equivalent to adding a `CONCURRENT` pragma before every loop in the file, including loops created from array syntax.

`-h vector`*`n`*

Default: `vector2`

The `vector`*n* option specifies 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. The values for *n*:

0 No automatic vectorization. Characteristics include low compile time and small compile size. This option is compatible with all scalar optimization levels.

1 Specifies conservative vectorization. Characteristics include moderate compile time and size. Loop nests are restructured. No vectorizations that might create false exceptions are performed. Results may differ slightly from results obtained when `-h vector0` is specified because of vector reductions. The `-h vector1` option is compatible with `-h scalar1`, `-h scalar2`, and `-h scalar3`.

2 Specifies moderate vectorization. Characteristics include moderate compile time and size. Loop nests are restructured. Compatible with `-h scalar2` and `-h scalar3`.

3 Specifies aggressive vectorization. Characteristics include potentially high compile time and size. Loop nests are restructured. Vectorizations that might create false exceptions in rare cases may be performed.

`-h [no]preferred_vector_width[=[64 | 128 | 256 | 512]]`

Specifies the preferred vector width to use when vectorizing loops. This option does not guarantee that the specified vector width will be used, only that it is preferred. The optimizer is still free to choose a smaller width if it is expected to perform better. As the set of acceptable width is target-sensitive and fairly complicated, the optimizer diagnoses any illegal values.

A value is not required when specifying `nopreferred_vector_width`.

Interprocedural Analysis (IPA) Optimization Options

Interprocedural analysis includes a set of compiler techniques that examine an entire program, as opposed to a single block of code, to increase the opportunity for optimization, particularly when a program uses many, frequently called functions.

Inlining and cloning transform code in ways that increase the opportunity for interprocedural (IPA) optimizations. The user controls inlining and cloning through the use of command line options alone or command line options in combination with directives placed within the code. By default, the compiler will attempt inline optimizations where appropriate, but not cloning. Inlining and cloning may increase object code size.

Inlining

The compiler supports the following inlining modes through the indicated options:

- Automatic inlining allows the compiler to automatically select which functions to inline. This occurs with the `-h ipa2` or greater option. When `-h ipa` is used alone, the candidates for expansion are all those functions that are present in the input file to the compile step.
- Explicit inlining allows for the explicit indication of which procedures the compiler should attempt to inline and occurs with the `-h ipafrom=` option alone.
- Combined inlining allows potential targets for inline expansion to be specified, while applying the selected level of inlining heuristics. If `-h ipa` is used in conjunction with `-h ipafrom=` the candidates for expansion are those functions present in source and `-h ipa` selects level of heuristics.

Cloning

Automatic cloning is enabled at `-hipa5`. The compiler first attempts to inline a call site. If inlining the call site fails, the compiler attempts to clone the procedure for the specific call site.

`-h ipa` *level*

Default: `-h ipa3`

Specifies the level of interprocedural optimization (IPA). `level` may be one of the following values, and includes functionality of previous non-zero levels:

0 Disable interprocedural analysis and optimizations. All inlining and cloning compiler directives are ignored.

1 Directive IPA. Inlining/cloning is attempted for call sites and routines that are under the control of a compiler directive. See *Inline and Clone Directives*.

2 Inlining. Inline a call site to an arbitrary depth as long as the expansion does not exceed some compiler-determined threshold. The call site must flatten for any expansion to occur. The call site is said to "flatten" when there are no calls present in the expanded code. The call site must reside within the body of a loop and the entire loop body must flatten. A loop body is said to "flatten" when all call sites within the body of the loop are flattened.

3 Constant actual argument inlining and tiny routine inlining. Default level for inlining. Any call site that contains a constant actual argument. Additionally, any call nest (regardless of location) that is below some small compiler-determined threshold will be inlined provided that call nest completely flattens. Cloning directives are recognized.

4 Aggressive inlining. A call site does not have to reside in a loop body to inline nor does the call site have to necessarily flatten. With `-h ipa4`, C++ codes will likely see improved execution performance. However, compile time may also increase.

5 Routine cloning is attempted if inlining fails at a given call site.

`-h ipafrom=`*source[:source] ...*

Explicitly indicate the procedures to consider for inline expansion or cloning. The source arguments identify each file or directory that contains the routines to consider for inlining or cloning. Spaces are not allowed on either side of the equal sign.

Use one or more of the following in the *source* argument:

- C or C++ sourcefiles with extension: .C, .c++, .C++, .cc, .cxx, .Cxx, . CXX, or .CC
- Directory containing filetypes mentioned above

Scalar Optimization Options

`-h [no]interchange`

Default: `interchange`

This option allows the compiler to attempt to interchange all loops - a technique that is used to improve performance by having the compiler swap an inner loop with an outer loop. The compiler attempts the interchange only if the interchange will increase performance. Loop interchange is performed only at scalar optimization level 2 or higher.

To disable interchange of loops individually, use the `nointerchange` directive prior to the loop.

`nointerchange` inhibits the compiler's attempts to interchange loops.

`-h scalar`*`n`*

Default: `scalar2`

Specifies the level of automatic scalar optimization to be performed. Scalar optimization directives are unaffected by this option. The values for n are:

0 Minimal automatic scalar optimization. Implies `-h zeroinc`.

1 Conservative automatic scalar optimization. Implies `-h nozeroinc`.

2 Aggressive automatic scalar optimization. The scalar optimizations that provide the best application performance are used, with some limitations imposed to allow for faster compilation times.

3 Very aggressive optimization; compilation times may increase significantly.

`-h [no]zeroinc`

Default: `-h nozeroinc`

The `-h nozeroinc` option improves run time performance by causing the compiler to assume that *constant increment variables* (CIVs) in loops are not incremented by expressions with a value of 0.

The `-h zeroinc` option causes the compiler to assume that some *constant increment variables* (CIVs) in loops might be incremented by 0 for each pass through the loop, preventing generation of optimized code.

`-h zeroinc` can cause less strength reduction to occur in loops that have variable increments.

A CIV is a variable that is incremented only by a loop invariant value. For example, in a loop with variable *`J`*, the statement `J = J + K`, where *`K`* can be equal to zero, *`J`* is a CIV.

Math Options

`-h fp`*`level`*

Default: `-h fp2`

The `-h fp` option controls the level of floating-point optimizations. The *`level`* argument controls the level of allowable optimization; 0 gives the compiler minimum freedom to optimize floating-point operations, while 4 gives it maximum freedom. The higher the level, the less the floating-point operations conform to the IEEE standard.

Each *`level`*, 1-4, includes the optimizations at the previous *`level`*.

The values for *`level`*:

0 Use this level only when the code pushes the limits of IEEE accuracy or requires strong IEEE standard conformance. The resulting executable code conforms more closely to the IEEE floating-point standard than the default mode (`-h fp2`). Many identity optimizations are disabled. Vectorization of floating-point and complex reductions are disabled. Executable code is slower than higher floating-point optimization levels.

1 Use this option only when the code pushes the limits of IEEE accuracy or requires strong IEEE standard conformance. This option performs generally safe, non-conforming IEEE optimizations, such as folding `a == a` to true, where `a` is a floating point object. At this level, a scaled complex divide mechanism is enabled that increases the range of complex values that can be handled without producing an underflow. Rewrite of division into multiplication by reciprocal is inhibited.

2 Default.

3 Use when performance is more critical than the level of IEEE standard conformance provided by `fp2`. The `-h fp3` option is an acceptable level of optimization for many applications.

4 Use if the application uses algorithms which are tolerant of reduced precision.

Table 2. Floating-point Optimization Levels

Optimization Type	**`fp0`**	**`fp1`**	**`fp2` (default)**	**`fp3`**	**`fp4`**
Safety	Maximum	High	High	Moderate	Low
Complex divisions	Accurate and slower	Accurate and slower	Fast[1]	Fast[1]	Fast[1]
Exponentiation rewrite	None	None	When optimization benefit is very high[2]	Always[2,3]	Always[2,3]
Strength reduction	None	None	Fast	Fast	Fast
Rewrite division as reciprocal equivalent[4]	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

[1] Algebraically correct but may lack precision in boundary cases.
[2] Rewriting values raised to a constant power into an algebraically equivalent series of multiplications and/or square roots.
[3] Rewriting exponentiations (ab) not previously optimized into the algebraically equivalent form exp(b * ln(a)).
[4] For example, x/y is transformed to x * 1.0/y.

Optimization Type	fp0	fp1	fp2 (default)	fp3	fp4
Inline 32-bit operations[5]	No	No	No	Yes	Yes
Fused multiply-add[6]	No	Yes	Yes	Yes	Yes

Debug Options

-G *debug_lvl*, -g

Enable the generation of debugging information used by symbolic debuggers. Allow debugging with breakpoints. Better debugging information comes at the cost of inhibiting certain optimization techniques, so choose the *debug_lvl* that best fits the debugging needs of any particular source file in an application.

The -g option is equivalent to -G0. The -g option is included for compatibility with earlier versions of the compiler and many other UNIX systems; the -G option is the preferred specification.

These options recognize OpenMP directives and disable all optimizations. They imply -hthread1 -homp -hfp0. The debugging options take precedence over any conflicting options that appear on the command line. If more than one debugging option appears, the last one specified overrides the others.

The -g or -G options can be specified on a per-file basis, so that only part of an application pays the price for improved debugging.

debug_lvl	Optimization	Breakpoints Allowed On	Debugging	Execution Speed
0	None	Every executable statement	Best	Limited
1	Partial	Block boundaries	Medium	Medium
2	Full	Function entry and exit	Limited	Best
3	Full	Every executable statement	Requires fast-track debugger	Best

-h [no]bounds (cc)

Default: -h nobounds

-h bounds provides checking of pointer and array references to ensure that they are within acceptable boundaries. -h nobounds disables these checks. For each dimension, the checks verify that the subscript is greater than or equal to 0 and less than the upper bound. For pointers, the upper bound is computed based on

[5] 32-bit division, square root, and reciprocal square root use very fast but less precise code sequences.
[6] Uses fused multiply-add instructions on architectures that support it.

the amount of the memory on the node. This amount is scaled at runtime by the number of UPC threads in the job for UPC pointers-to-shared with definite blocksize. For arrays, the declared (possibly implicit) upper bound of the dimension is used. If the dimension is the `THREADS`-scaled dimension of a UPC shared array with definite blocksize, the upper bound for the check is computed at runtime based on the number of UPC threads in the job.

`-h develop`

Default: off

Reduce compile time at the expense of optimization. This option is intended to be used when a program is under development and compiled frequently. This option is different from and independent of the `-O` option. For example, `-O0` disables all optimizations, but sometimes can increase compile time because certain optimizations reduce code size, which allow other phases of the compiler to deal with less code.

`-h dir_check`

Default: off

Enables directive checking at run time. Errors detected at compile time are reported during compilation and so are not reported at run time. The following directives are checked: `collapse`, and the `loop_info` clauses `min_trips` and `max_trips`. Violation of a run time check results in an immediate fatal error diagnostic.

Optimization of enclosing and adjacent loops is degraded when run time directive checking is enabled. This capability, though useful for debugging, is not recommended for production runs.

`-h gasp[=`*opt*`[:`*opt*`]]`

Default: `disabled`

Request GASP (Global Address Space Performance Analysis) instrumentation. Requests instrumentation of events generated by shared local accesses. Instrumenting these events can add runtime overhead to the application.

`#pragma pupc [on|off]` has no effect in the current GASP implementation.

When *opt* is specified, the compiler provides additional instrumentation as follows:

local	Requests instrumentation of events generated by shared local accesses. Instrumenting these events can add runtime overhead to the application
functions	Enables function instrumentation. Sets `-hipa0`

`-h nodwarf`

Disables DWARF generation during compilation. By default, DWARF source line information is generated to support traceback analysis. `-hdwarf` is deprecated. This option has no affect if `-g` or `-G` is specified.

`-h zero`

Default: disabled

Cause stack-allocated memory to be initialized to all zeros.

Message Options

-h [no]message=*n*[:*n*...]

Default: Determined by `-h msglevel_`*n*

Enables or disables specified compiler messages, where n is the number of a message to be enabled or disabled. More than one message number can be specified; multiple numbers must be separated by a colon with no intervening spaces. For example, to disable messages `CC-174` and `CC-9`, specify: `-h nomessage=174:9`.

The `-h [no]message=`*n* option overrides `-h msglevel_`*n* for the specified messages. If n is not a valid message number, it is ignored. Any compiler message except `ERROR`, `INTERNAL`, and `LIMIT` messages can be disabled; attempts to disable these messages by using the `-h nomessage` option are ignored.

-h msglevel_*n*

Default: `-h msglevel_3`

Specify the lowest level of severity of messages to be issued. Messages at the specified level and above are issued. Values for *n* are:

0	Comment
1	Note
2	Caution
3	Warning
4	Error

-h report=*args*

Generates report messages specified in *args* and allows the direction of the specified messages to a file. No spaces are allowed around the equal sign (=) or any of the args codes. The `-h msgs` option also provides optimization messages.

The *args* field can be any combination of the following options:

f

Writes specified messages to *file*`.V`, where file is the source file specified on the command line. If the f option is not specified, messages are written to stderr.

i

Generates inlining optimization messages.

s

Generates scalar optimization messages.

v

Generates vector optimization messages.

Print inlining and scalar optimization messages for myfile.c:

```
cc -h report=is myfile.c
```

-h [no]abort

Default: `-h noabort`

Controls whether a compilation aborts if an error is detected.

-h errorlimit[=*max_err_msgs*]

Default: `-h errorlimit=100`

Specify the maximum number of error messages the compiler prints before it exits, where *max_error_msgs* is a positive integer. Specifying `-h errorlimit=0` disables exiting on the basis of the number of errors. Specifying `-h errorlimit` with no qualifier is the same as setting *max_error_msgs* to 1.

-h error_on_warning

The `-h error_on_warning` option changes the message level of all warning messages to error. This option is off by default.

Default: off

Compilation Phase Options

-E

Directs the compiler to execute only the preprocessor phase of the compiler. Directs the output to `stdout` and inserts appropriate `#line` *linenumber* preprocessing directives. Takes precedence over the `-h feonly`, `-S`, and `-c` options.

Equivalent to `-P`, except `-E` directs output to `stdout` and inserts appropriate `#line` *linenumber* preprocessing directives. When both the `-E` and `-P` options are specified, the last one specified takes precedence.

-P

The `-P` option directs the compiler to execute only the preprocessor phase of the compiler for each source file specified. The preprocessed output for each source file is written to a file with a name that corresponds to the name of the source file and has a `.i` suffix substituted for the suffix of the source file.

Similar to the `-E` option. When both options are specified, the last one specified takes precedence.

-h feonly

Limits the compiler to syntax checking. The optimizer and code generator are not executed. This option takes precedence over `-S` and `-c`.

-S

Compiles the named source files and leaves their assembly language output in the corresponding files suffixed with a `.s`. If this option is used with `-G` or `-g`, debugging information is not generated. This option takes precedence over `-c`.

-c

Create a relocatable object file for each named source file but does not link the object files. The relocatable object file name corresponds to the name of the source file. The `.o` suffix is substituted for the suffix of the source file.

-#, -##, -###

The `-#` option produces output indicating each phase of the compilation as it is executed. Each succeeding output line overwrites the previous line.

The `-##` option produces output indicating each phase of the compilation as it is executed.

The `-###` option is the same as `-##`, except the compilation phases are not executed.

-W *phase*, "*args*"

Passes *args* directly to a *phase* of the compiling system. *phase* indicates a compiling system phase as shown:

Phase	Compiling System Phase	Command
0 (zero)	Compiler	`CC,cc`
a	Assember	`as`
c	CUDA linker	nvlink
l	linker	`ld`
x	PTX assember	`ptxas`

args to be passed to compiler system phases can be entered in either of two styles:

- *args* enclosed in double quotes. Spaces appear within *args*.
- *args* not enclosed in double quotes. No spaces appear within *args*. Commas can appear wherever spaces normally appear; an option and its argument can be either separated by a comma or not separated.

If a comma is part of an argument, it must be preceded by the \ character.

For example, any of the following command lines would send `-e name` to the linker.

```
%cc -Wl,"-e name" file.c
```

```
%cc -Wl,-e,name file.c
```

```
% cc -Wl,-ename file.c
```

Because the preprocessor is built into the compiler, `-Wp` and `-W0` are equivalent.

The `-Wl,-rpath` *ldir* option changes the runtime library search algorithm to look for files in directory *ldir* at link time. To request more than one library directory, specify multiple `-rpath` options. At link time, all *ldir* arguments are added to the executable.

A library may be found at link time with a `-L` option, but may not be found at run time if a corresponding `-Wl,-rpath` option was not supplied. Also, note that the compiler driver does not pass the `-rpath` option to the linker. Explicitly specify `-Wl,-rpath` when using `-L`.

The dynamic linker will search all *ldir* paths first for shared dynamic libraries at runtime, with one exception. The Linux environment variable `LD_LIBRARY_PATH` precedes all other search paths for shared dynamically linked libraries. The use of `LD_LIBRARY_PATH` is discouraged! Setting `LD_LIBRARY_PATH` changes the shared dynamically linked library search paths for all executable files in the environment.

The `-Wx,` *args* option can be used to pass command line arguments to the PTX assembler for OpenACC applications.

The `-Wc,` *args* option can be used to pass command line arguments to the CUDA linker for OpenACC applications.

`-Y` *phase*, *dirname*

Specifies a new directory, *dirname*, from which the designated *phase* should be executed. *phase* indicates a compiling system phase as shown:

Phase	Compiling System Phase	Command
`0` (zero)	Compiler	`CC` , `cc`
`a`	Assembler	`as`
`l`	Linker	`ld`

Because there is no separate preprocessor, `-Yp` and `-Y0` are equivalent.

Preprocess Options

`-C`

Retain all comments in the preprocessed source code, except those on preprocessor directive lines. By default, the preprocessor phase strips comments from the source code. This option is useful in combination with the `-P` option or `-E` option.

`-D` *macro*[=*def*]

Define macro as if it were defined by a `#define` directive. If no =*def* argument is specified, macro is defined as 1. Predefined macros also exist; these are described in *Predefined Macro Use*. Any predefined macro except those required by the standard (see *Macros Required by the C and C++ Standards*) can be redefined by the `-D` option. The `-U` option overrides the `-D` option when the same macro name is specified, regardless of the order of options on the command line.

`-h [no]pragma[=`*name*`[:`*name* ...`]]`

Default: `-h pragma` (no pragmas disabled)

Enable or disable the processing of specified directives in the source code, where *name* can be the name of a directive or a word to specify a group of directives. More than one name can be specified. Multiple names must be separated by a colon and have no intervening spaces.

Name	Group	Directives Affected
`all`	All	All
`allinline`	Inlining	`inline_*`
`allscalar`	Scalar optimization	`blockable, blockingsize, noblocking, nointerchange, suppress, unroll/nounroll`
`allvector`	Vectorization	`concurrent, novector, loop_info, hand_tuned, ivdep, nopattern, novector, permutation, pipeline/nopipeline, prefervector, safe_address, safe_conditional`
`omp`	OpenMP	All OpenMP directives
`acc`	OpenACC	All OpenACC directives

When using this option to enable or disable individual directives, note that some directives must occur in pairs. For these directives, disable both directives to disable either; otherwise, the disabling of one of the directives may cause errors when the other directive is (or is not) present in the compilation unit.

`-I` *incldir*

Specifies a directory for files named in `#include` directives when the `#include` file names do not have a specified path. Each directory specified must be specified by a separate `-I` option. The order in which directories are searched for files named on `#include` directives is determined by enclosing the file name in either quotation marks (" ") or angle brackets (< and >).

Directories for `#include "file"` are searched in the following order:

1. Directory of the input file
2. Directories named in `-I` options, in command-line order
3. Site-specific and compiler release-specific include files directories
4. Directory `/usr/include`

Directories for `#include <`*file*`>` are searched in the following order:

1. Directories named in `-I` options, in command-line order
2. Site-specific and compiler release-specific include files directories
3. Directory `/usr/include`

If the `-I` option specifies a directory name that does not begin with a slash (`/`), the directory is interpreted as relative to the current working directory and not relative to the directory of the input file, if different from the current working directory.

```
%cc -I. -I yourdir mydir/b.c
```

Search order:

1. `mydir` (#include "file" only)
2. Current working directory, specified by `-I.`
3. `yourdir` (relative to the current working directory), specified by `-I yourdir`.

4. Site-specific and compiler release-specific include files directories
5. Directory `/usr/include`

-M

Provides information about recompilation dependencies that the source file invokes on `#include` files and other source files. This information is printed in the form expected by make. Such dependencies are introduced by the `#include` directive. The output is directed to `stdout`.

-nostdinc

Stops the preprocessor from searching for include files in the standard directories `/usr/include`, and `/usr/include/c++`.

-U *macro*

Removes any initial definition of *macro*. Any predefined macro except those required by the standard (see *Macros Required by the C and C++ Standards*) can be undefined by the `-U` option. The `-U` option overrides the `-D` option when the same macro name is specified, regardless of the order of options on the command line. Macros defined in the system headers are not predefined macros and are not affected by the `-U` option.

Linker Options

-h [system|default]_alloc

Default: `-h default_alloc`

By default, the compiler uses a modified malloc implementation that offers better support for memory needs. The `-h system_alloc` option directs the compiler to link in the native malloc provided by the OS instead of the modified implementation.

-h [no]pgas_runtime

Default: `-h pgas_runtime`

The `-h pgas_runtime` option directs the compiler driver to link with the runtime libraries required when linking programs that use UPC, or coarrays, which is default. In general, `aprun` must be used to launch the resulting executable. The `-hnopgas_runtime` option prevents this runtime library environment from being added to the link line. Use the `-hnopgas_runtime` option when there is a program, that does not use UPC or coarrays, and it needs to be executed outside of the aprun/alps job launch context. For example, test a serial program which does not contain any UPC or coarray code on a login or service node, or fork/exec an executable on a compute node. Also, compile non-coarray Fortran using the `-hnocaf` option.

-l *libname*

The `-l libname` option directs the compiler driver to search for the specified object library file when linking an executable file. To request more than one library file, specify multiple `-l` options.

When statically linking, the compiler driver searches for libraries by prepending *ldir*/lib to *libname* and appending .a, for each *ldir* that has been specified by using the -L option. It uses the first file it finds.

When dynamically linking, the library search process is similar to the static case, with a few differences. The compiler driver searches for libraries by prepending *ldir*/lib on the front of *libname* and appending .so on the end of it, for each *ldir* that has been specified by using the -L option. If a matching .so is not found, the compiler driver replaces .so with .a and repeats the process from the beginning. It uses the first file it finds.

There is no search order dependency for libraries.

If personal libraries are specified by using the -l command line option, those libraries are added before the default CCE library list. The -l option is passed to the linker. For example, when the following command line is issued, the linker looks for a library named libmylib.a and adds it to the top of the list of default libraries.

```
cc -l mylib target.c
```

-L *ldir*

Changes the -l option search algorithm to look for library files in directory ldir during link time. To request more than one library directory, specify multiple -L options.

The linker searches for library files in the compiler release-specific directories.

Multiple -L options are treated cumulatively as if all ldir arguments appeared on one -L option preceding all -l options. Therefore, do not attempt to link functions of the same name from different libraries through the use of alternating -L and -l options.

-o *outfile*

Produces an absolute binary file named outfile. A file named a.out is produced by default. When this option is used in conjunction with the -c option and a single source file, a relocatable object file named outfile is produced.

Program Model Specific Options

-h [no]acc

Default: -h acc

Enables or disables compiler recognition of OpenACC pragmas.

-h mpi*n*

Default: mpi0

Enables or disables optimization of MPI operations. mpi1 enables this option.

-h [no]omp

Default: -h omp if -O1 or higher is implied or specified.

Enables or disables compiler recognition of OpenMP pragmas. If -O0 is specified, then -h noomp is implied.

-h upc (cc only)

Default: `off`

Enables compilation of Unified Parallel C (UPC) code. UPC is a C language extension for parallel program development that allows for explicitly specifying parallel programming through language syntax rather than through library functions such as are used in MPI or SHMEM.

Miscellaneous Options

-h cpu=*target_system*

Specifies the Cray system on which the absolute binary file is to be executed, where *target_system* can be either x86-64 or opteron (single or dual-core), barcelona or shanghai (quad-core), istanbul (6-core), mc8 (8-core), mc12 (12-core), interlagos (16-core), interlagos-cu (8-compute unit), abudhabi (16-core) , abudhabi-cu (8-compute unit), ivybridge, sandybridge, or haswell.

The interlagos and abudhabi processors contain up to 8 compute units, each of which contains two integer cores and a shared FPU. These targets assume that the user intends to run with one thread per core (up to 16 per processor), while the `cpu-cu` target assumes that the user intends to run with one thread per compute unit (up to 8 per processor or one thread per FPU).

Rather than setting this option directly, users should load one of the targeting modules (craype-mc12 or craype-interlagos-cu, for example). The targeting modules set `CRAY_CPU_TARGET` and define paths to the corresponding libraries. The compiler driver script translates `CRAY_CPU_TARGET` to the corresponding `cpu=` option when calling the compiler.

If the target_system is set during compilation of any source file, it must also be set to that same target during linking and loading. If a user wishes to override the current target_system value set by the module environment (via the `CRAY_CPU_TARGET` definition), they should do so by specifying `-hcpu=`*target_system* on the compiler command line.

-h craylibs_arch_override

Forces Craymath to honor the processor architecture specified by the `-h cpu` option. Processor architecture is typically specified by loading one of the targeting modules, e.g., `craype-sandybridge`, but can be overriden at link time by using the `-h cpu` option. If the `CRAYLIBS_ARCH_OVERRIDE` environment variable is specified, it takes precedence over this option.

-h [no]fp_trap

Default: `fp_trap` if traps are enabled using the `-K trap` option, or if `-Ofp[0,1]` is in effect. Otherwise, the default is `nofp_trap`.

Controls whether the compiler generates code that is compatible with floating-point traps.

-h ident=*name*

Default: `-h ident=`*File name specified on the command line*

Changes the ident name to *name*. This name is used as the module name in the object file (`.o` suffix) and assembler file (`.s` suffix). Regardless of whether the name is specified or the default name is used, the following transformations are performed on name:

- All `.` characters in ident *name* are changed to `$`.
- If the ident name starts with a number, a `$` is added to the beginning of the ident name.

`-h keepfiles`

Default: off

Prevents the removal of the object (`.o`) and temporary assembly (`.s`) files after an executable is created. Normally, the compiler automatically removes these files after linking them to create an executable. Since the original object files are required to instrument a program for performance analysis, if CrayPat is to be used to conduct performance analysis experiments, use this option to preserve the object files.

`-h keep_frame_pointer`*opts*

Default: `off`

Retain call stack information back to main entry point for CrayPat performance sampling. Prevents call stack frame from being optimized out of a function so CrayPat performance sampling is able to trace call stack back to entry point.

`-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 optimize the runtime characteristics of the application.

`-h network=`*nic*

Specifies the target machine's system interconnection network. Currently, supported values for *nic* are gemini and aries.

`-h pic`, `-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. For the Cray implementation, the `pic` and `PIC` options have the same effect and should be used to compile codes using more than 2GB of static memory, or for creating dynamically linked libraries.

`-h prototype_intrinsics`

Default: off

Simulates the effect of including intrinsics.h at the beginning of a compilation. Use this option if the source code does not include `intrinsics.h` and the code cannot be modified.

See *Intrinsic Functions*.

`-h [no]threadsafe`

Default: `-h threadsafe`

Enables or disables the generation of threadsafe code. Code that is threadsafe can be used with pthreads and OpenMP. This option is not binary-compatible with code generated by Cray C 8.1 or Cray C++ 5.1 and earlier compilers.

Users who need binary compatibility with previously compiled code can use -h nothreadsafe, which causes the compiler to be compatible with Cray C 8.1 or Cray C++ 5.1 and earlier compilers at the expense of not being threadsafe. C or C++ code compiled with -h threadsafe (the default) cannot be linked with C or C++ code compiled with -h nothreadsafe or with code compiled with a Cray C 8.1, Cray C++ 5.1, or earlier compiler.

-K trap=*exceptions*

Default: off

Enable traps for the specified *exceptions*. By default, no exceptions are trapped. Enabling traps by using this option also has the effect of setting `-h [no]fp_trap`.

If the specified options contradict each other, the last option predominates. For example, `-K trap=none,fp` is equivalent to `-Ktrap=fp`.

This option does not affect compile time optimizations; it detects run time exceptions.

This option is processed only at link time and affects the entire program; it is not processed when compiling subprograms. Use this command line option to set traps beginning with execution of the main program. The program may subsequently change these settings by calling intrinsic or library procedures. If this option is used, `-hfp_trap` may need to be specified when other files of the application are compiled.

denorm	Trap on denormalized operands.
divz	Trap on divide-by-zero.
fp	Trap on divz, inv, or ovf exceptions.
inexact	Trap on inexact result (i.e., rounded result). Enabling traps for inexact results is not recommended.
inv	Trap on invalid operation.
none	Disables all traps (default).
ovf	Trap on overflow (i.e., the result of an operation is too large to be represented).
unf	Trap on underflow (i.e., the result of an operation is too small to be represented).

-V

Displays compiler version information. Version information consists of the product name, the version number, and the current date and time.

Unlike all other command-line options, this option can be specified without specifying an input file name. If the command line specifies no source file, no compilation occurs.

-X *npes*

Specifies the number of UPC threads for a UPC source file compiled under the UPC static threads environment. The *npes* value must match the number of processing elements (PEs) that will be specified through aprun at job launch. Ensure that all object files are compiled using the same *npes* value. When using mixed *npes* values or if the number of PEs provided at run time differs from the `-X` *npes* value, a run time error will be received.

#pragma Directives

#pragma directives are used within C/C++ source code to request certain kinds of special processing. Unless otherwise noted, each directive is supported in both the Cray C and C++ compiler. #pragma directives are expressed in the following form:

#pragma[_CRI] identifier [arguments]

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. Macro expansion occurs on the directive line after the directive name. That is, macro expansion is applied only to arguments.

Alternative Form: _Pragma

Compiler directives can also be specified in the following form, which has the advantage in that it can appear inside macro definitions:

```
_Pragma("_CRI identifier")
```

_Pragma is an extension to the C and C++ standards. This form has the same effect as using the #pragma form, except that everything that appeared on the line following the #pragma must now appear inside the double quotation marks and parentheses. The expression inside the parentheses must be a single string literal; it cannot be a macro that expands into a string literal. Macros are expanded in the string literal argument for _Pragma in an identical fashion to the general specification of a #pragma directive.

The following is an example using the #pragma form:

```
#pragma _CRI concurrent
```

The following is the same example using the alternative form:

```
_Pragma("_CRI concurrent")
```

In the following example, the loop automatically vectorizes wherever the macro is used:

```
#define _str( _X ) # _X
#define COPY( _A, _B, _N

{

  int i;
  _Pragma( "_CRI concurrent" )
  _Pragma( _str( _CRI loop_info cache_nt( _B ) ) )
  for ( i = 0; i < _N; i++ ) {
    _A[i] = _B[i];
  }
```

```
}

void
copy_data( int *a, int *b, int n )
{
  COPY( a, b, n );
}
```

Scope

Unless otherwise noted, each directive has both local and global scope. A directive may also have a lexical block scope. A lexical block is the scope within which a directive is on or off and is bounded by the opening curly brace just before the directive was declared, and the corresponding closing curly brace. Only applicable executable statements within the lexical block are affected as indicated by the directive. The lexical block does not include the statements contained within a procedure that is called from the lexical block. This example code fragment shows the lexical block for the UPC strict and relaxed directives:

Lexical scope for UPC directives

```
void Example (void)
{
  #pragma _CRI upc strict // UPC strict state is on
  . . .
  {
     // UPC strict state is on
     #pragma _CRI upc relaxed // UPC strict state is now off
     . . .
  }
// UPC strict state is back on
}
```

Protecting Directives

Use the following coding technique to insure that other compilers used to compile this code will not interpret the directive. Some compilers diagnose any directives that they do not recognize. The Cray C and C++ compilers diagnose directives that are not recognized only if the _CRI specification is used.

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

Directives in C++

C++ prohibits referencing undeclared objects or functions. Objects and functions must be declared prior to using them in a #pragma directive. This is not always the case with C. Some `#pragma` directives take function names as arguments, for example: `#pragma _CRI weak`, `#pragma _CRI suppress`, and `#pragma _CRI inline_always` *name* `[,`*name* `...]`. Member functions and qualified names are allowed for these directives.

Loop Directives

Many directives apply to loops. Unless otherwise noted, these directives must appear before a `for`, `while`, or `do while` loop. These directives may also appear before a label for if...goto loops. If a loop directive appears before a label that is not the top of an if...goto loop, it is ignored.

Automatically vectorize loop whenever COPY macro is used

```
#define _str( _X ) # _X
#define COPY( _A, _B, _N )
{
int i;
_Pragma( "_CRI concurrent" )
_Pragma( _str( _CRI loop_info cache_nt( _B ) ) )
for ( i = 0; i < _N; i++ ) {
_A[i] = _B[i];
}
}
void
copy_data( int *a, int *b, int n )
{
COPY( a, b, n );
}
```

Miscellaneous Directives

[no]autothread

```
#pragma _CRI autothread
#pragma _CRI noautothread
```

Scope: Local

The `[no]autothread` directive turns autothreading on or off for selected blocks of code.

[no]bounds

```
#pragma _CRI bounds
#pragma _CRI nobounds
```

The `bounds` directive specifies that pointer and array references are to be checked. The nobounds directive specifies that this checking is to be disabled. For each dimension, the checks verify that the subscript is greater than or equal to 0 and less than the upper bound. For pointers, the upper bound is computed based on the amount of the memory on the node. This amount is scaled at runtime by the number of UPC threads in the job for UPC pointers-to-shared with definite blocksize. For arrays, the (possibly implicit) declared upper bound of the dimension is used. If the dimension is the THREADS-scaled dimension of a UPC shared array with definite blocksize, the upper bound for the check is computed at runtime based on the number of UPC threads in the job. Both directives may be used only within function bodies. They apply until the end of the function body or until another bounds/nobounds directive appears. They ignore block boundaries.

Use `#pragma _CRI bounds`

```
int a[30];
#pragma _CRI bounds
void f(void)
{
   int x;
   x = a[30];
   .
```

```
    .
    .
}
```

cache

```
#pragma _CRI cache base_name [,base_name ...]
```

base_name The base name of the object that should be placed into the cache. This can be the base name of any object such as an array, scalar structure, and so on, without member references like `C[10]`. If a pointer in the list is specified, only the references, not the pointer itself, are cached.

The `cache` directive asserts that all memory operations with the specified symbols as the base are to be allocated in cache. This is an advisory directive. The cache directive is meaningful for stores in that it allows the user to override a decision made by the automatic cache management. This directive may be locally overridden by the use of a `#pragma loop_info` directive. This directive overrides automatic cache management decisions. To use the directive, place it only in the specification part, before any executable statement.

cache_nt

```
#pragma _CRI cache_nt base_name [,base_name ...]
```

base_name The base name of the object that should use non-temporal reads and writes. This can be the base name of any object such as an array, scalar structure, and so on, without member references like C[10]. If a pointer in the list is specified, only the references, not the pointer itself, have the cache non-temporal property.

Use this directive to identify objects that should not be placed in cache. This is an advisory directive that specifies objects that should use non-temporal reads and writes.

This directive overrides the automatic cache management level that was specified using the `-h cache`*n* option on the compiler command line. This directive may be overridden locally by use of a `loop_info` directive.

duplicate

```
#pragma _CRI duplicate actual as dupname[, dupname] ...
#pragma _CRI duplicate actual as (dupname[, dupname] ...)
```

Scope: Global

The *actual* argument is the name of the actual function to which duplicate names will be assigned. The *dupname* list contains the duplicate names that will be assigned to the actual function. The dupname list may be optionally parenthesized. The word `as` must appear as shown between the actual argument and the comma-separated list of *dupname* arguments. The `duplicate` directive can appear anywhere in the source file, in global scope. The actual name specified on the directive line must be defined somewhere in the source as an externally accessible function; the actual function cannot have a static storage class.

Because duplicate names are simply additional names for functions and are not functions themselves, they cannot be declared or defined anywhere in the compilation unit. To avoid aliasing problems, duplicate names may not be referenced anywhere within the source file, including appearances on other directives. In other words, duplicate names may only be referenced from outside the compilation unit in which they are defined.

```
#pragma _CRI duplicate

#include <complex.h>
extern void maxhits(void);
```

```
#pragma _CRI duplicate maxhits as count, quantity /* OK */
void maxhits(void)
{
#pragma _CRI duplicate maxhits as tempcount
/* Error: #pragma _CRI duplicate can't appear in local scope */
}
double _Complex minhits;
#pragma _CRI duplicate minhits as lower_limit
/* Error: minhits is not declared as a function */
extern void derivspeed(void);
#pragma _CRI duplicate derivspeed as accel
/* Error: derivspeed is not defined */
static void endtime(void)
{
}
#pragma _CRI duplicate endtime as limit
/* Error: endtime is defined as a static function */
```

#pragma _CRI duplicate

The directive argument *dupname* cannot be referenced in same compilation unit.

```
void converter(void)
{
structured(void);
}
#pragma _CRI duplicate converter as factor, multiplier
/* OK */
void remainder(void)
{
}
#pragma _CRI duplicate remainder as factor, structured
/* Error: factor and structured are referenced in this file */
```

#pragma _CRI duplicate

Use duplicate names to provide alternate external names for functions.

```
main.c:
extern void fctn(void), FCTN(void);
main()
{
fctn();
FCTN();
}
fctn.c:
#include <stdio.h>
void fctn(void)
{
printf("Hello world\n");
}
#pragma _CRI duplicate fctn as FCTN
```

Files main.c and fctn.c are compiled and linked using the following command line: **cc main.c fctn.c**. When the executable file a.out is run, the program generates the following output:

```
Hello world
Hello world
```

ident

```
#pragma _CRI ident text
```

The ident pragma directs the compiler to store the string indicated by *text* into the object (.o) file. This can be used to place a source identification string into an object file.

message

```
#pragma _CRI message "text"
```

The message directive directs the compiler to write the message defined by *text* to stderr as a warning message. Unlike the error directive, the compiler continues after processing a message directive.

```
#pragma _CRI message

 #define FLAG 1
 #ifdef FLAG
 #pragma _CRI message "FLAG is Set"
 #else
 #pragma _CRI message "FLAG is NOT Set"
 #endif
```

[no]opt

```
#pragma _CRI opt
#pragma _CRI noopt
```

Scope: Global

The noopt directive disables all automatic optimizations and causes optimization directives to be ignored in the source code that follows the directive. Disabling optimization removes various sources of potential confusion in debugging. The opt directive restores the state specified on the command line for automatic optimization and directive recognition. These directives have global scope and override related command line options.

The following example illustrates the use of the opt and noopt compiler directives:

```
#include <stdio.h>

void sub1(void)
{
        printf("In sub1, default optimization\n");
}

#pragma _CRI noopt

void sub2(void)
{
        printf("In sub2, optimization disabled\n");
}
#pragma _CRI opt

void sub3(void)
{
        printf("In sub3, optimization enabled\n");
```

```
}

main()
{
        printf("Start main\n");
        sub1();
        sub2();
        sub3();
}
```

prefetch

```
#pragma _CRI prefetch [([lines(num)][, level(num)][, write][, nt])] var[, var] ...
```

lines(*num*) Specifies the number of cache lines to be prefetched. num is an expression that evaluates to an integer constant at compilation time. By default, the number of cache lines prefetched is 1.

level(*num*) Specifies the level of cache into which data is loaded. *num* is an expression that evaluates to an integer constant at compilation time. The cache level defaults to 1, the level closest to the processing unit. This level specification has little effect for current x86 targets.

write Specifies that the prefetch is for data to be written. When data is to be written, a prefetch instruction can move a block into the cache so that the expected store will be to the cache. Prefetch for write generally brings the data into the cache in an exclusive or modified state. By default, the prefetch is for data to be read. If the target architecture does not support prefetch for write, the prefetch will automatically become a prefetch for read.

nt Specifies that the prefetch is for non-temporal data. By default, the prefetch is for temporal data. Data with temporal locality (persistence), is expected to be accessed multiple times.

The general prefetch directive instructs the compiler to generate explicit prefetch instructions which load data from memory into cache prior to read or write access. The memory location to be prefetched is defined by *var*, which specifies any valid variable, member, or array element reference.

The compiler issues the prefetch instruction when it encounters the prefetch directive. The directive allows the user to influence almost every aspect of prefetch behavior. The default behavior prefetches one cache line, into L1 cache, for read access, and assumes temporal locality. The prefetch directive can be used inside and outside of loops, in a loop preamble, or before a function call to reduce cache-miss memory latency. The compiler will attempt to avoid multiple prefetches to the same cache line, which can be created as a result of optimization. All variables specified on the same prefetch directive line share the same behavior. If different behavior is needed for different variables, use multiple `prefetch` directive lines. The general `prefetch` directive supersedes the effects of any relevant `loop_info [no]prefetch` directives and the `-h [no]autoprefetch` command line option. The Cray Fortran compiler command line option `-x prefetch` can be used to disable all general prefetch directives in Fortran source code. The compiler command line option `-h nopragma=prefetch` can be used to disable all general prefetch directives in C and C++ source code.

prefetch directive

```
void
add( long * restrict a, long * restrict b, const int n )
{
   int i;
#pragma _CRI prefetch (lines(2)) b[0]
   for ( i = 0; i < n; i++ ) {
#pragma _CRI prefetch b[i+16]
      a[i] += b[i];
   }
```

```
   return;
}
```

probability directives

```
#pragma probability const
#pragma probability_almost_always
#pragma probability_almost_never
```

const Expression that evaluates to a floating point constant at compilation time. (0.0 <= *const* <= 1.0.)

The probablity directives specify information used by interprocedure analysis (IPA) and the optimizer to produce faster code sequences. The specified probability is a hint, rather than a statement of fact. This information is used to guide inlining decisions, branch elimination optimizations, branch hint marking, and the choice of the optimal algorithmic approach to the vectorization of conditional code. These directives can appear anywhere executable code is legal. Each directive applies to the block of code where it appears. It is important to realize that the directive should not be applied to a conditional test directly; rather, it should be used to indicate the relative probability of a then or else branch being executed.

Specify `almost_never` and `almost_always` by using the `probability` *const* values 0.0 and 1.0, respectively.

probability directive

This example states that the probability of entering the block of code with the assignment statement is 0.3 or 30%. This also means that `a[i]` is expected to be greater than `b[i]` 30% of the time. Note that the probability directive appears within the conditional block of code, rather than before it. This removes some of the ambiguity that has plagued other implementations that tie the directive directly to the conditional code.

```
   if ( a[i] > b[i] ) {
#pragma probability 0.3
        a[i] = b[i];
}
```

__builtin_expect intrinsic & probablity directive

These two code fragments are roughly equivalent.

```
if ( __builtin_expect( a[i] > b[i], 0 ) ) {
    a[i] = b[i];
}
```

```
   if ( a[i] > b[i] ) {
#pragma _CRI probability_almost_never
        a[i] = b[i];
}
```

weak

```
#pragma _CRI weak var
```

var The name of an external

`#pragma _CRI weak` *sym1* `=` *sym2*

sym1 Defines an externally visible weak symbol

sym2 Defines an externally visible strong symbol defined in the current compilation.

Scope: Global

When statically linking, the `weak` directive specifies an external identifier that may remain unresolved throughout the compilation. This directive has no effect when dynamically linking. A weak external reference can be a reference to a function or to a data object. A `weak` external does not increase the total memory requirements of the program.The first form allows the declaration of one or more weak references on one line. The second form allows the assignment of a strong reference to a weak reference. The `weak` directive must appear at global scope.

Declaring an object as a `weak` external directs the linker to do one of these tasks:

- Link the object only if it is already linked (a strong reference exists); otherwise, leave it is as an unsatisfied external. The linker does not display an unsatisfied external message if weak references are not resolved.
- If a strong reference is specified in the weak directive, resolve all weak references to it.

The linker treats weak externals as unsatisfied externals, so they remain silently unresolved if no strong reference occurs during compilation. Be responsible to ensure that run time references to weak external names do not occur unless the linker (using some "strong" reference elsewhere) has actually linked the entry point in question.

`weak` directive

Attributes of weak externals depend on the form of the directive:

- First form, weak externals must be declared, but not defined or initialized, in the source file.
- Second form, weak externals may be declared, but not defined or initialized, in the source file.
- Either form, weak externals cannot be declared with a static storage class.

```
extern long x;
#pragma _CRI weak x /* x is a weak external data object */
extern void f(void);
#pragma _CRI weak f /* f is a weak external function */

extern void g(void);
#pragma _CRI weak g=fun;   /* g is a weak external function
                                  with a strong reference to fun */

long y = 4;
#pragma _CRI weak y   /* ERROR - y is actually defined */

static long z;
#pragma _CRI weak z   /* ERROR - z is declared static */

void fctn(void)
{
#pragma _CRI weak a   /* ERROR - directive must be at global scope */
}
```

Vectorization Directives

Because vector operations cannot be expressed directly in the compiler, the compiler must be capable of transforming scalar operations into equivalent vector operations. The candidates for vectorization are operations in loops and assignments of structures. Compiler directives may be used to control vectorization.

concurrent

```
#pragma _CRI concurrent [safe_distance=n]
```

n An integer that represents the number of additional consecutive loop iterations that can be executed in parallel without danger of data conflict. n must be an integer constant > 0. If SAFE_DISTANCE=n is not specified, the distance is assumed to be infinite, and the compiler ignores all cross-iteration dependencies. The concurrent directive is ignored if the safe_distance clause is used and vectorization is requested on the command line.

The `concurrent` directive indicates that no data dependence exists between array references in different iterations of the loop. This directive affects the loop that immediately follows it. This can be useful for vectorization optimizations.

concurrent Directive

The concurrent directive indicates that the relationship `k>3` is true. The compiler will safely load all the array references `x[i-k]`, `x[i-k+1]`, `x[i-k+2]`, `x[i-k+3]` during loop iteration `i`.

```
#pragma _CRI concurrent safe_distance=3

for (i = k + 1; i < n;i++) {
  x[i] = a[i] + x[i-k];
}
```

hand_tuned

```
#pragma _CRI hand_tuned
```

Assert that the code in the next loop nest has been arranged by hand for maximum performance, and the compiler should restrict some of the more aggressive automatic expression rewrites. The compiler should still fully optimize and vectorize the loop within the constraints of the directive. The hand_tuned directive applies to the next loop in the same manner as the `concurrent` and `safe_address` directives.

Use of this directive may severely impede performance. Use carefully and evaluate performance before and after employing this directive.

ivdep

```
#pragma _CRI ivdep [ SAFEVL=vlen | INFINITEVL ]
```

vlen

Specifies a vector length in which no dependency will occur. vlen must be an integer between 1 and 1024 inclusive.

INFINITEVL

Specifies an infinite safe vector length. No dependency will occur at any vector length.

When the `ivdep` directive appears before a loop, the compiler ignores vector dependencies, including explicit dependencies, in any attempt to vectorize the loop. ivdep applies only to the first for loop or while loop that follows the directive within the same program unit.

If no vector length is specified, the vector length used is infinity.

If a loop with an `ivdep` directive is enclosed within another loop with an `ivdep` directive, the `ivdep` directive on the outer loop is ignored. When the Cray compiler vectorizes a loop, it may reorder the statements in the source code to remove vector dependencies. When `ivdep` is specified, the statements in the loop or array syntax statement are assumed to contain no dependencies as written, and the Cray compiler does not reorder loop statements.

loop_info

```
#pragma _CRI loop_info prefer_thread
#pragma _CRI loop_info prefer_nothread
#pragma _CRI loop_info [min_trips(c)] [est_trips(c)] [max_trips(c)]
[cache( symbol[,symbol ...] )][cache_nt(symbol[,symbol ...] ) ][prefetch]
[noprefetch]
```

c

An expression that evaluates to an integer constant at compilation time.

min_trips

Specifies guaranteed minimum number of trips.

est_trips

Specifies estimated or average number of trips.

max_trips

Specifies guaranteed maximum number of trips.

cache

Specifies that *symbol* is to be allocated in cache; this is the default if no hint is specified and the `cache_nt` directive is not specified.

cache_nt

Specifies that *symbol* is to use non-temporal reads and writes.

prefetch

Specifies a preference that prefetches be performed for the following loop.

noprefetch

Specifies a preference that no prefetches be performed for the following loop.

symbol

The base name of the object that should not be placed into the cache. This can be the base name of any object (such as an array or scalar structure) without member references like `C[10]`. If specifying a pointer in the list, only the references, not the pointer itself, have the no cache allocate property.

Scope: Local

The `loop_info` directive allows additional information to be specified about the behavior of a loop, including run time trip count, hints on cache allocation strategy, and threading preference. The `loop_info` directive provides information to the optimizer and can produce faster code sequences.

Use `loop_info` immediately before a for loop to indicate minimum, maximum, estimated trip count. The compiler will diagnose misuse at compile time when able, or when option `-h dir_check` is specified at run time.

For cache allocation hints, use the `loop_info` directive to override default settings, `cache` or `cache_nt` directives, or override automatic cache management decisions. The cache hints are local and apply only to the specified loop nest.

Use the `loop_info prefer_thread` directive to indicate the preference that the loop following the directive be threaded. The `loop_info prefer_nothread` indicates the preference that the loop following the directive should not be threaded.

`loop_info` Directive

The minimum trip count is 1 and the maximum trip count is 1000.

```
void
loop_info( double *restrict a, double *restrict b, double s1, int n )
{
   int i;
#pragma _CRI loop_info min_trips(1) max_trips(1000), cache_nt(b)
   for (i = 0; i< n; i++) {
      if(a[i] != 0.0) {
         a[i] = a[i] + b[i]*s1;
      }
   }
}
```

nopattern

`#pragma _CRI nopattern`
Scope: Local

The `nopattern` directive disables pattern matching for the loop immediately following the directive. By default, the compiler detects coding patterns in source code sequences and replaces these sequences with calls to optimized library functions. In most cases, this replacement improves performance. There are cases, however, in which this substitution degrades performance. This can occur, for example, in loops with very low trip counts. In such a case, use the nopattern directive to disable pattern matching and cause the compiler to generate inline code.

The `nopattern` directive disables pattern matching for the loop immediately following the directive.

`nopattern` Directive

Placing the nopattern directive in front of the outer loop of a nested loop turns off pattern matching for the matrix multiply that takes place inside the inner loop.

```
double a[100][100], b[100][100], c[100][100];
void nopat(int n)
{
   int i, j, k;
#pragma _CRI nopattern
   for (i=0; i < n; ++i) {
      for (j = 0; j < n; ++j) {
         for (k = 0; k < n; ++k) {
            c[i][j] += a[i][k] * b[k][j];
         }
```

```
        }
    }
}
```

[no]vector

```
#pragma _CRI novector
#pragma _CRI vector [clause[, clause] ... ]
```

always	Vectorize the loop that immediately follows the directive. This directive states a vectorization preference and does not guarantee that the loop has no memory-dependence hazard. This directive has the same effect as the `prefervector` directive.
aligned	Directs the compiler to generate aligned data movement instructions for array references when vectorizing. For current Intel processors, data alignment is necessary for efficient vectorization. Use with care to improve performance. If some of the access patterns are actually unaligned, using the ALIGNED clause may generate incorrect code. This directive also directs the compiler to ignore explicit and implicit vector dependencies.
unaligned	Directs the compiler to generate unaligned data movement instructions for all array references when vectorizing.

The `novector` directive suppresses compiler attempts to vectorize loops and array syntax statements. It overrides any other vectorization-related directives, as well as the `-h vector` and `-O vector`*n* command line options. These directives are ignored if vectorization or scalar optimization has been disabled.

In C/C++, the novector directive applies only to the following loop. When applied to an outer loop in a nest, the directive also applies to all inner loops. After a vector directive is specified, automatic vectorization is enabled for all loop nests.

`novector` Directive

```
#pragma _CRI novector
for (i = 0; i < h; i++) {     /* Loop not vectorized */
    a[i] = b[i] + c[i];
}
```

permutation

```
#pragma _CRI permutation symbol [, symbol ] ...
```

Specifies that an integer array has no repeated values. This directive is useful when the integer array is used as a subscript for another array (vector-valued subscript). This directive may improve code performance.

In a sequence of array accesses that read array element values from the specified symbols with no intervening accesses that modify the array element values, each of the accessed elements will have a distinct value.

When an array with a vector-valued subscript appears on the left side of the equal sign in a loop, many-to-one assignment is possible. Many-to-one assignment occurs if any repeated elements exist in the subscripting array. If it is known that the integer array is used merely to permute the elements of the subscripted array, it can often be determined that many-to-one assignment does not exist with that array reference.

permutation Directive

Sometimes a vector-valued subscript is used as a means of indirect addressing because the elements of interest in an array are sparsely distributed; in this case, an integer array is used to select only the desired elements, and no repeated elements exist in the integer array. The permutation directive does not apply to the array `a`. Rather, it applies to the pointer used to index into it, `ipnt`. By knowing that ipnt is a permutation, the compiler can safely generate an unordered scatter for the write to `a`.

```
int *ipnt;
#pragma permutation ipnt
...
  for ( i = 0; i < N; i++ ) {
    a[ipnt[i]] = b[i] + c[i];
}
```

[no]pipeline

```
#pragma _CRI pipeline
#pragma _CRI nopipeline
```

Software-based vector pipelining (software vector pipelining) provides additional optimization beyond the normal hardware-based vector pipelining. In software vector pipelining, the compiler analyzes all vector loops and automatically attempts to pipeline a loop if doing so can be expected to produce a significant performance gain. This optimization also performs any necessary loop unrolling.

In some cases the compiler either does not pipeline a loop that could be pipelined or pipelines a loop without producing performance gains. In these situations, use the `pipeline` or `nopipeline` directive to advise the compiler to pipeline or not pipeline the loop immediately following the directive.

Software vector pipelining is valid only for the innermost loop of a loop nest. These directives are advisory only. While the `nopipeline` directive can be used to inhibit automatic pipelining, and the `pipeline` directive can be used to attempt to override the compiler's decision not to pipeline a loop, the compiler cannot be forced to pipeline a loop that cannot be pipelined.

Loops that have been pipelined are so noted in loopmark listing messages.

prefervector

```
#pragma _CRI prefervector
```

Scope: Local

Directs the compiler to vectorize the loop immediately following the directive if the loop contains more than one loop in the nest that can be vectorized. The directive states a vectorization preference and does not guarantee that the loop has no memory-dependence hazard.

prefervector Directive

Both loops can be vectorized, but the directive directs the compiler to vectorize the outer `for` loop. Without the directive and without any knowledge of n and m, the compiler would vectorize the inner loop.

```
float a[1000], b[100][1000];
void
f(int m, int n)
```

```
{
   int i, j;
#pragma _CRI prefervector
   for (i = 0; i < n; i++) {
      for (j = 0; j < m; j++) {
         a[i] += b[j][i];
      }
   }
}
```

pgo loop_info
`#pragma _CRI prefervector`
Scope: Local

Enables profile-guided optimizations by tagging loopmark information as having come from profiling. For information about CrayPat and profile information, see the *Using Cray Performance Measurement and Analysis Tools* guide.

safe_address
`#pragma _CRI safe_address`
Scope: Local

Specifies that it is safe to speculatively execute memory references within all conditional branches of a loop; these memory references can be safely executed in each iteration of the loop. For most code, this directive can improve performance significantly by preloading vector expressions. However, most loops do not require this directive to have preloading performed. `safe_address` is required only when the safety of the operation cannot be determined or index expressions are very complicated.

The `safe_address` directive is an advisory directive. That is, the compiler may override the directive if it determines the directive is not beneficial. If the directive is not used on a loop and the compiler determines that it would benefit from the directive, it issues a message indicating such. The message is similar to this:

```
CC-6375 cc: VECTOR File = ctest.c, Line = 6
A loop would benefit from "#pragma safe_address"
```

If using the directive on a loop and the compiler determines that it does not benefit from the directive, it issues a message that states the directive is superfluous and can be removed.

To see the messages, use the `-h report=v` or `-h msgs` option.

Incorrect use of the directive can result in segmentation faults, bus errors, or excessive page faulting. However, it should not result in incorrect answers. Incorrect usage can result in very severe performance degradations or program aborts.

safe_address directive

In this example, the compiler will not preload vector expressions, because the value of `j` is unknown. However, if it is known that references to `b[i][j]` are safe to evaluate for all iterations of the loop, regardless of the condition, the `safe_address` directive can be used. With the

directive, the compiler can safely load `b[i][j]` as a vector, merge `0.0` where the condition is true, and store the resulting vector safely.

```
void x3( double a[restrict 1000], int j )
{
int i;
#pragma _CRI safe_address
for ( i = 0; i < 1000; i++ ) {
   if ( a[i] != 0.0 ) {
      b[j][i] = 0.0;
      }
   }
}
```

safe_conditional

`#pragma _CRI safe_conditional`

Specifies that it is safe to execute all memory references and arithmetic operations within all conditional branches of the subsequent scalar or vector loop nest. It can improve performance by allowing the hoisting of invariant expressions from conditional code and by allowing prefetching of memory references.

The `safe_conditional` directive is an advisory directive. The compiler may override the directive if it determines the directive is not beneficial.

Incorrect use of the directive can result in segmentation faults, bus errors, or excessive page faulting. However, it should not result in incorrect answers. Incorrect usage can result in very severe performance degradations or program aborts.

safe_conditional directive

In this example, without the `safe_conditional` directive, the compiler cannot precompute the invariant expression `s1*s2` because their values are unknown and may cause an arithmetic trap if executed unconditionally. However, if the condition is known to be true at least once, then `s1*s2` is safe to speculatively execute. The `safe_conditional` compiler directive can be used to imply the safety of the operation. With the directive, the compiler evaluates `s1*s2` outside of the loop, rather than under control of the conditional code. In addition, all control flow is removed from the body of the vector loop, because `s1*s2` no longer poses a safety risk.

```
void
safe_cond( double a[restrict 1000], double s1, double s2 )
{
   int i;
#pragma _CRI safe_conditional
   for (i = 0; i< 1000; i++) {
      if( a[i] != 0.0) {
         a[i] = a[i] + s1*s2;
      }
   }
}
```

Scalar Optimization Directives

Scalar optimization directives control aspects of code generation, register storage, and other scalar operations.

blockable

`#pragma _CRI blockable(`*num_loops*`)`

num_loops	Number of subsequent loops to be blocked

The `blockable` directive specifies that it is legal and desirable to cache block the subsequent loop nest, even when the compiler has not made such a determination. To be legally blockable, the nest must be perfect (without code between constituent loops), rectangular (trip counts of member loops are fixed over the life time of nest), and fully permutable (loop interchange and unrolling is legal at all levels). This directive both permits and requests blocking of the indicated loop nest.

If a `blockingsize` directive is also provided for the indicated loop, the following rules apply:

- If `blockingsize` is at least two, the indicated blockingsize is used.
- If `blockingsize` is zero, the loop itself is not blocked and it is treated as an inner loop (as part of the nest that traverses the cache block tile).
- If `blockingsize` is one, the loop itself is not blocked and it is treated as an outer loop (as a loop in the nest that moves from tile to tile).

When no blockingsize directive is supplied the compiler chooses the blockingsize according to its own heuristics.

`blockable` and `blockingsize` Directives

```
%cat blk.c

#define N 1000

float A[N][N];
float B[N][N];

void
func(int n)
{
#pragma _CRI blockable(2)
#pragma _CRI blockingsize( 32 )
for (int i = 2; i <= N-1; ++i)  {
#pragma _CRI blockingsize( 128 )
        for (int j = 2; j <= N-1; ++j)  {
            A[i][j] = B[i-1][j-1]
                        + B[i-1][j+1]
                        + B[i+1][j-1]
                        + B[i+1][j+1];
        }
}
% cc -c -hlist=md blk.c; cat blk.lst
...
    7.              func(int n)
    8.              {
    9.              #pragma _CRI blockable(2)
   10.              #pragma _CRI blockingsize( 32 )
   11.  + b-------<     for (int i = 2; i <= N-1; ++i)  {
```

```
   12.    b         #pragma _CRI blockingsize( 128 )
   13.    b Vbr4--<     for (int j = 2; j <= N-1; ++j)  {
   14.    b Vbr4            A[i][j] = B[i-1][j-1]
   15.    b Vbr4                    + B[i-1][j+1]
   16.    b Vbr4                    + B[i+1][j-1]
   17.    b Vbr4                    + B[i+1][j+1];
   18.    b Vbr4-->     }
   19.    b------->     }
   20.              }

CC-6294 CC: VECTOR File = blk.c, Line = 11
  A loop was not vectorized because a better candidate was found at
line 13.

CC-6051 CC: SCALAR File = blk.c, Line = 11
  A loop was blocked according to user directive with block size 32.

CC-6051 CC: SCALAR File = blk.c, Line = 13
  A loop was blocked according to user directive with block size 128.
...
```

blockingsize

```
#pragma _CRI blockingsize(n1 [,n2])
#pragma _CRI noblocking
```

n1

Specify a value greater than or equal to 0 for the primary cache.

n2

Specify a value less than or equal to 2**30 for the secondary cache.

If *n1* or *n2* are 0, the loop is not blocked, but the entire loop is inside the block.

The `blockingsize` directive asserts that the loop following the directive is involved in a cache blocking situation for the primary or secondary cache.

The `noblocking` directive prevents the compiler from involving the subsequent loop in a cache blocking situation.

If the loop is involved in a blocking situation, it will have a block size of *n1* for the primary cache and *n2* for the secondary cache. The compiler attempts to include this loop within such a block but cannot guarantee inclusion.

blockingsize Directive

The compiler makes 20 x 20 blocks when blocking, but it could block the loop nest such that loop K is not included in the file.

```
      SUBROUTINE AMAT(X,Y,Z,N,M,MM)
      REAL(KIND=8) X(100,100), Y(100,100), Z(100,100)
      DO K = 1, N
!DIR$ BLOCKABLE(J,I)
!DIR$ BLOCKING SIZE (20)
         DO J = 1, M
!DIR$ BLOCKING SIZE (20)
            DO I = 1, MM
               Z(I,K) = Z(I,K) + X(I,J)*Y(J,K)
            END DO
```

```
      END DO
   END DO
   END
```

If K is excluded, add a BLOCKINGSIZE(0) directive just before loop K to specify that the compiler should generate a loop such as the following example:

```
      SUBROUTINE AMAT(X,Y,Z,N,M,MM)
      REAL(KIND=8) X(100,100), Y(100,100), Z(100,100)
      DO JJ = 1, M, 20
         DO II = 1, MM, 20
            DO K = 1, N
               DO J = JJ, MIN(M, JJ+19)
                  DO I = II, MIN(MM, II+19)
                     Z(I,K) = Z(I,K) + X(I,J)*Y(J,K)
                  END DO
               END DO
            END DO
         END DO
      END DO
      END
```

noblocking

```
#pragma _CRI noblocking
```

Asserts that the loop following the directive should not be cache blocked for the primary or secondary cache. It is an error to place a noblocking directive before a loop that is part of a blockable collection.

[no]collapse

```
#pragma _CRI collapse(loop-number1, loop-number2 [,loop-number3] ... )
```

loop-number

Specify a value greater than or equal to 0.

```
#pragma _CRI nocollapse
```

Scope: Local

When the `collapse` directive is applied to a loop nest, the loop numbers of the participating loops must be listed in order of increasing access stride. Loop numbers range from 1 to the nesting level of the most deeply nested loop. The directive enables the compiler to assume appropriate conformity between trip counts. The compiler diagnoses misuse at compile time (when able); or, if `-h dir_check` is specified, at run time.

The `nocollapse` directive disqualifies the immediately following loop from collapsing with any other loop. Collapse is almost always desirable, so use this directive sparingly. Loop collapse is a special form of loop coalesce. Any perfect loop nest may be coalesced into a single loop, with explicit rediscovery of the intermediate values of original loop control variables. The rediscovery cost, which generally involves integer division, is quite high. Therefore, coalesce is rarely suitable for vectorization. It may be beneficial for multithreading. By definition, loop collapse occurs when loop coalesce may be done without the rediscovery overhead. To meet this requirement, all memory accesses must have uniform stride.

[no]interchange

```
#pragma _CRI interchange(loop_number1, loop_number2[, loop_number3] ...)
```

loop_number

Number from 1 to nesting depth of the most deeply nested loop

`#pragma _CRI nointerchange`
Scope: Local

The `interchange` control directives specify whether or not the order of the following two or more, perfectly nested loops should be interchanged. These directives apply to the subsequent loops.

The `interchange` directive specifies two or more loop numbers, ranging from 1 to the nesting depth of the most deeply nested loop, specified in any order. The compiler reorders perfectly nested loops. If they are not perfectly nested, unexpected results may occur.

The `nointerchange` directive inhibits loop interchange on the loop that immediately follows the directive.

`interchange` Directive

The `interchange` directive reorders the loops; the `k` loop becomes the outermost and the `i` loop the innermost:

```
#define N 100

A[N][N][N];

void
f(int n)
{
  int i, j, k;

#pragma _CRI interchange( 2, 3, 1 )
  for (i=0; i < n; i++) {
    for (k=0; k < n; k++) {
      for (j = 0; j < n; j++) {
        A[k][j][i] = 1.0;
      }
    }
  }
}
```

`suppress`

`#pragma _CRI suppress` *`func`*

Scope: Global

`#pragma _CRI suppress [`*`var`*`]`

Scope: Local

This directive suppresses optimization in two ways, determined by its use with either global or local scope.

- The global scope `suppress` directive specifies that all associated local variables are to be written to memory before a call to the specified function. This ensures that the value of the variables will always be current.
- The local scope `suppress` directive stores current values of the specified variables in memory. If the directive lists no variables, all variables are stored to memory. This directive causes the values of these variables to be reloaded from memory at the first reference following the directive. The net effect of the local `suppress`

directive is similar to declaring the affected variables to be `volatile` except that the volatile qualifier affects the entire program, whereas the local `suppress` directive affects only the block of code in which it resides.

[no]unroll

`#pragma _CRI unroll [n]`

n

Specifies no loop unrolling (n = 0 or 1) or the total number of loop body copies to be generated (2 ≤ n ≤ 63)

`#pragma _CRI nounroll`

Scope: Local

The unroll directive allows the user to control unrolling for individual loops or to specify no unrolling of a loop. Loop unrolling can improve program performance by revealing cross-iteration memory optimization opportunities such as read-after-write and read-after-read. The effects of loop unrolling also include:

- Improved loop scheduling by increasing basic block size
- Reduced loop overhead
- Improved chances for cache hits

Disable loop unrolling for the next loop. The `nounroll` directive is functionally equivalent to the `unroll 0` and `unroll 1` directives. The *n* argument applies only to the `unroll` directive and if a value for *n* is not specified, the compiler will determine the number of copies to generate based on the number of statements in the loop nest. Note: The compiler cannot always safely unroll non-innermost loops due to data dependencies. In these cases, the directive is ignored. The `unroll` directive can be used only on loops with iteration counts that can be calculated before entering the loop. If `unroll` is specified on a loop that is not the innermost loop in a loop nest, the inner loops must be nested perfectly. That is, all loops in the nest can contain only one loop, and the innermost loop can contain work.

`unroll` Directive

`unroll` by 2.

```
#pragma _CRI unroll 2
for (i = 0; i < 10; i++) {
      for (j = 0; j < 100; j++) {
        a[i][j] = b[i][j] + 1;
      }
}
```

With outer loop unrolling, the compiler produces the following nest, in which the two bodies of the inner loop are adjacent:

```
for (i = 0; i < 10; i += 2) {
      for (j = 0; j < 100; j++) {
        a[i][j] = b[i][j] + 1;
      }
      for (j = 0; j < 100; j++) {
        a[i+1][j] = b[i+1][j] + 1;
      }
}
```

The compiler then jams, or fuses, the inner two loop bodies, producing the following nest:

```
for (i = 0; i < 10; i += 2) {
      for (j = 0; j < 100; j++) {
        a[i][j] = b[i][j] + 1;
        a[i+1][j] = b[i+1][j] + 1;
      }
}
```

Illegal unrolling of outer loops

Outer loop unrolling is not always legal because the transformation can change the semantics of the original program. For example, unrolling the following loop nest on the outer loop would change the program semantics because of the dependency between `a[i][...]` and `a[i+1][...]`. The directive will cause incorrect code due to dependencies.

```
#pragma _CRI  unroll 2
for (i = 0; i < 10; i++) {
  for (j = 1; j < 100; j++) {
    a[i][j] = a[i+1][j-1] + 1;
  }
}
```

nofission

`#pragma _CRI nofission` *func*

Scope: Local

Instructs the compiler not to split statements in a given loop into distinct loops. Fission is prevented only for the loop specified; loops nested within the indicated loop remain fission candidates unless likewise annotated.

[no]fusion

```
#pragma _CRI fusion
#pragma _CRI nofusion
```

Scope: Local

The `nofusion` directive instructs the compiler to not attempt loop fusion on the following loop even when the `-h fusion` option was specified on the compiler command line. The fusion directive instructs the compiler to attempt loop fusion on the following loop unless `-h nofusion` was specified on the compiler command line.

Inline and Clone Directives

Inlining and cloning directives can only appear in local scope — inside a function definition.

Inlining directives always take precedence over the command line settings with the exception of `-h ipa0`, which instructs the compiler to ignore inlining directives.

Cloning directives are enabled with `-h ipa5`.

inline

```
#pragma _CRI inline_enable
#pragma _CRI inline_disable
#pragma _CRI inline_reset
#pragma _CRI inline_always [name [,name] ...]
#pragma _CRI inline_never [name [,name] ...]
```

Inlining replaces calls to user-defined functions with the code that represents the function. This can improve performance by saving the expense of the function call overhead. It also increases the possibility of additional code optimization. Inlining may increase object code size.

Enabling inlining instructs the compiler to attempt to inline functions at call sites. Resetting inlining returns the inlining state to the state specified on the command line (`-h ipa`*n* or `-O ipa`*n*).

The enabling/disabling directives remain in effect until the opposite directive is encountered, or until a reset directive is encountered, or until the end of the program unit.

The `always` directive specifies functions that the compiler should always attempt to inline. If the directive is placed in the definition of the function, inlining is attempted at every call site to that function in the entire input file being compiled. If the directive is placed in a function other than the definition, inlining is attempted at every call site to *name* within the specific function containing the directive. The `never` directive specifies functions that should not be inlined. If the directive is placed in the definition of the function, inlining is never attempted at any call site to that function in the entire input file being compiled. If the directive is placed in a function other than the definition, inlining is never attempted at any call site to *name* within the specific function containing the directive. An error message is issued if both never and always are specified for the same procedure in the same program unit.

inline Directive

```
void qux(int x)
{
void bar(void);
int a = 1;

  x = a+a+a+a+a+a+a+a+a+a+a+a+a;
  bar();
}

void foo(void)
{
int j = 1;

#pragma inline_enable /* enable inlining at all call sites here
forward */
 qux(j);
 qux(j);
#pragma inline_disable /* disable inlining at all call sites here
forward */
 qux(j);
#pragma inline_reset /* reset control to the command line -hipa4 */
 qux(j);
}
```

`inline_reset` directive

```
{
void f1()
#pragma _CRI inline_disable  /* No inlining will be done in f1;
  ...
}


void f2()
{
#pragma _CRI inline_disable  /* turn off all inlining to the end of
the routine or another directive is encountered.
  ...

#pragma _CRI inline_reset  /* The inlining state is -h ipa3
for the remainder of f2;
  ...
}
```

Using inlining within function template

When applied to a function template specialization, the inlining directive will apply to all instantiations of the template, not just the specialization. For example, in the following case, both `foo<double>` and `foo<float>` will be affected by the `inline_always` directive:

```
template<typename T>
int foo(T i, int j) {
  return j;
}

template <>
int foo(double f, int i) {
#pragma _CRI inline_always foo
  return i + 5;
}

int bar(float f,int i) {
  return foo(f,i);
}

int main(void) {
  foo(1.0,7);
  bar(1.0,7);
  return 0;
}
```

`clone`

```
#pragma _CRI clone_enable
#pragma _CRI clone_disable
#pragma _CRI clone_reset
#pragma _CRI clone_always [name [,name] ...]
```

`#pragma _CRI clone_never [name [,name] ...]`

Cloning is the attempt to duplicate a procedure under certain conditions and replace dummy arguments with associated constant actual arguments throughout the cloned procedure. The compiler attempts to clone a procedure when a call site contains actual arguments that are scalar integer and/or scalar logical constants. When the constants are exposed to the optimizer, it can generate more efficient code.

If cloning is enabled, cloning is attempted at call sites. If cloning is disabled, cloning is not attempted at call sites. The reset directive returns the cloning to the state specified on the compiler command line. The cloning directives remain in effect until a different cloning directive is encountered or until the end of the program unit. Use compile options `-hipa`*N* or `-Oipa`*N* where *N* is equal to, or greater than 4 to enable all cloning directives. Use `-hnegmsgs` to see messages that highlight where cloning did not occur.

Use the `clonealways` and `clonenever` directives to control cloning of a proecedure for the compilation of the whole input file. If the directive is placed in the definition of the function, cloning is always/never attempted at every call site to name in the entire input file being compiled. If the directive is placed in a function other than the definition, cloning is always/never attempted at every call to *name* within the specific function containing the directive.

PGAS Directive

defer_sync

`#pragma _CRI defer_sync`

PGAS (Partitioned Global Address Space) language data references made by the single statement immediately following the PGAS `defer_sync` directive are not synchronized until the next fence instruction. Normally the compiler synchronizes the references in a statement as late as possible without violating program semantics. The purpose of the `defer_sync` directive is to synchronize the references even later, beyond where the compiler can determine it is safe. Use this directive to force all references in the next statement to be non-blocking. This helps for cases where the compiler cannot prove that it is safe.

For example, if there is a remote-memory access (RMA) put near the end of a subroutine, the compiler must guard against the put value being read back immediately after the subroutine returns, so the put is synchronized just before returning. The programmer, however, may know that the value is not read back and can insert a PGAS `defer_sync` directive.

defer_sync directive in UPC

```
void my_put( shared int* x, int thread, int value ) {
    #pragma pgas defer_sync
        x[thread] = value;
}
```

OpenMP Overview

The OpenMP API provides a parallel programming model that is portable across shared memory architectures from Cray and other vendors. The OpenMP specification is accessible at *http://openmp.org/wp/openmp-specifications/*.

Supported Version

CCE supports the OpenMP API, Version 4.0, with the following exceptions. The most up-to-date exceptions are listed on the man pages:

- Task switching is not implemented. The thread that starts executing a task is the thread that finishes the task.
- Support for OpenMP Random Access Iterators (RAIs) in the C++ Standard Template Library (STL) is deferred.
- The task `depend` clause is supported, but tasks with dependences are serialized.
- Cancellation does not destruct/deallocate implicitly private local variables. It correctly handles explicitly private variables.
- `simd` functions will not vectorize if the function definition is not visible for inlining at the callsite.
- The `linear` clause is not supported on combined or compound constructs.
- The `device` clause is not supported. The other mechanisms for selecting a default device are supported: `OMP_DEFAULT_DEVICE` and `omp_set_default_device`.
- The only API calls allowed in target regions are: `omp_is_initial_device`, `omp_get_thread_num`, `omp_get_num_threads`, `omp_get_team_num`, and `omp_get_num_teams`.
- Parallel constructs are supported in target regions, but they are limited to one thread.
- User-defined reductions are not supported in `target` regions.

Compiling

By default, the CCE compiler recognizes OpenMP directives. These CCE options affect OpenMP applications:

- `-h [no]omp`
- `-h thread`*`n`*

Executing

For OpenMP applications, use both the `OMP_NUM_THREADS` environment variable to specify the number of threads and the `aprun -d`*`depth`* option to specify the number of CPUs hosting the threads. The number of threads specified by `OMP_NUM_THREADS` should not exceed the number of cores in the CPU. If neither the `OMP_NUM_THREADS` environment variable nor the `omp_set_num_threads` call is used to set the number of OpenMP threads, the system defaults to 1 thread. For further information, including example OpenMP programs, see the *Cray Application Developer's Environment User's Guide*.

Debugging

The `-g` option provides debugging support for OpenMP directives. The `-g` option provides debugging support identical to specifying the `-G0` option. This level of debugging implies `-homp` which means that most optimizations disabled but OpenMP directives are recognized, and `-h fp0`. To debug without OpenMP, use `-g -xomp` or `-g -hnoomp`, which will disable OpenMP and turn on debugging.

OpenMP Implementation Defined Behavior

The *OpenMP Application Program Interface Specification*, presents a list of implementation defined behaviors. The Cray implementation is described in the following sections.

Atomicity of memory access by multiple threads

When multiple threads access the same shared memory location and at least one thread is a write, threads should be ordered by explicit synchronization to avoid data race conditions and the potential for non-deterministic results. Always use explicit synchronization for any access smaller than one byte.

Internal Control Variables

Table 3. Initial Values of OpenMP ICVs

ICV	Initial Value	Note
nthreads-var	1	
dyn-var	TRUE	Behaves according to Algorithm 2-1 of the specification.
run-sched-var	static, 0	
stacksize-var	128 MB	
wait-policy-var	ACTIVE	
thread-limit-var	64	Threads may be dynamically created up to an upper limit which is 4 times the number of cores/node. It is up to the programmer to try to limit oversubscription.
max-active-levels-var	1023	
def-sched-var	static, 0	The chunksize is rounded up to improve alignment for vectorized loops.

Dynamic Adjustment of Threads

The ICV *dyn-var* is enabled by default. Threads may be dynamically created up to an upper limit which is 4 times the number of cores/node. It is up to the programmer to try to limit oversubscription.

If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads specified for the parallel region exceeds the number that the runtime system can supply, the program

terminates. The number of physical processors actually hosting the threads at any given time is fixed at program startup and is specified by the `aprun -d` *depth* option. The **`OMP_NESTED`** environment variable and the `omp_set_nested()` call control nested parallelism. To enable nesting, set **`OMP_NESTED`** to true or use the omp_set_nested() call. Nesting is disabled by default.

Tasks

There are no untied tasks in this implementation of OpenMP. There are also no implementation-defined task scheduling points.

Directives and Clauses

- **`atomic`** directive

  When supported by the target architecture, atomic directives are lowered into hardware atomic instructions. Otherwise, atomicity is guaranteed with a lock. OpenMP **`atomic`** directives are compatible with C11 and C++11 atomic operations, as well as GNU atomic builtins.

- **`for`** directive

  For the **`schedule(guided,`** *chunk*) clause, the size of the initial chunk for the master thread and other team members is approximately equal to the trip count divided by the number of threads.

  For the **`schedule(runtime)`** clause, the schedule type and chunk size can be chosen at run time by setting the **`OMP_SCHEDULE`** environment variable. If this environment variable is not set, the schedule type and chunk size default to **`static`** and 0, respectively.

  In the absence of the **`schedule`** clause, the default **`schedule`** is **`static`** and the default chunk size is approximately the number of iterations divided by the number of threads.

- **`parallel`** directive

  If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads specified for the parallel region exceeds the number that the runtime system can supply, the program terminates.

  The number of physical processors actually hosting the threads at any given time is fixed at program startup and is specified by the `aprun -d` *depth* option.

  The `OMP_NESTED` environment variable and the `omp_set_nested()` call control nested parallelism. To enable nesting, set `OMP_NESTED` to `true` or use the `omp_set_nested()` call. Nesting is disabled by default.

- **`loop`** directive

  The integer type or kind used to compute the iteration count of a collapsed loop are signed 64-bit integers, regardless of how the original induction variables and loop bounds are defined. If the schedule specified by the runtime schedule clause is specified and *run-sched-var* is auto, then the Cray implementation generates a static schedule.

- **`private`** clause

  If a variable is declared as private, the variable is referenced in the definition of a statement function, and the statement function is used within the lexical extent of the directive construct, then the statement function references the private version of the variable.

- **`sections`** construct

Multiple structured blocks within a single sections construct are scheduled in lexical order and an individual block is assigned to the first thread that reaches it. It is possible for a different thread to execute each section block, or for a single thread to execute multiple section blocks. There is not a guaranteed order of execution of structured blocks within a section.

- **`single`** directive

  A single block is assigned to the first thread in the team to reach the block; this thread may or may not be the master thread.

Library Routines

- **`omp_set_num_threads`**

  Sets `nthreads-var` to a positive integer. If the argument is < 1, then set `nthreads-var` to 1.

- **`omp_set_schedule`**

  Sets the schedule type as defined by the current specification. There are no implementation defined schedule types.

- **`omp_set_max_active_levels`**

  Sets the *`max-active-levels-var`* ICV. Defaults to 1023. If argument is < 1, then set to 1.

- **`omp_set_dynamic()`**

  The **`omp_set_dynamic()`** routine enables or disables dynamic adjustment of the number of threads available for the execution of subsequent parallel regions by setting the value of the *`dyn-var`* ICV. The default is **`on`**.

- **`omp_set_nested()`**

  The **`omp_set_nested()`** routine enables or disables nested parallelism, by setting the *`nest-var`* internal control variable (ICV). The default is **`false`**.

- **`omp_get_max_active_levels`**

  There is a single max-active-levels-var ICV for the entire runtime system. Thus, a call to **`omp_get_max_active_levels`** will bind to all threads, regardless of which thread calls it.

Environment Variables

`OMP_SCHEDULE`

The default values for this environment variable are static for type and 0 for chunk. For the schedule (guided,chunk) clause, the size of the initial chunk for the master thread and other team members is approximately equal to the trip count divided by the number of threads. For the schedule(runtime) clause, the schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable. If this environment variable is not set, the schedule type and chunk size default to static and 0, respectively. In the absence of the schedule clause, the default schedule is static and the default chunk size is approximately the number of iterations divided by the number of threads.

OMP_NUM_THREADS	If this environment variable is not set and the omp_set_num_threads() routine is not used to set the number of OpenMP threads, the default is 1 thread. The maximum number of threads per compute node is 4 times the number of allocated processors. If the requested value of `OMP_NUM_THREADS` is more than the number of threads an implementation can support, the behavior of the program depends on the value of the `OMP_DYNAMIC` environment variable. If `OMP_DYNAMIC` is false, the program terminates. If `OMP_DYNAMIC` is true, it uses up to 4 times the number of allocated processors. For example, on a 8-core Cray XE system, this means the program can use up to 32 threads per compute node.
OMP_DYNAMIC	The default value is true.
OMP_NESTED	The default value is false.
OMP_STACKSIZE	The default value for this environment variable is 128 MB.
OMP_WAIT_POLICY	Provides a hint to an OpenMP implementation about the desired behavior of waiting threads by setting the wait-policy-var ICV. A compliant OpenMP implementation may or may not abide by the setting of the environment variable. The default value for this environment variable is active.
OMP_MAX_ACTIVE_LEVELS	The default value is 1023.
OMP_THREAD_LIMIT	Sets the number of OpenMP threads to use for the entire OpenMP program by setting the thread-limit-var ICV. The Cray implementation defaults to 4 times the number of available processors.

Cray-specific OpenMP API

This section describes Open MP API specific to Cray.

cray_omp_set_wait_policy

```
void cray_omp_set_wait_policy( const char *policy );
```

This routine allows dynamic modification of the wait-policy-var ICV value, which corresponds to the `OMP_WAIT_POLICY` environment variable. The policy argument provides a hint to the OpenMP runtime library environment about the desired behavior of waiting threads; acceptable values are `ACTIVE` or `PASSIVE` (case insensitive). It is an error to call this routine in an active parallel region. The OpenMP runtime library supports a "wait policy" and a "contention policy," both of which can be set with the following environment variables:

```
OMP_WAIT_POLICY=(ACTIVE|PASSIVE)
CRAY_OMP_CONTENTION_POLICY=(Automatic|Standard|MonitorMwait)
```

These environment variables allow the policies to be set once at program launch for the entire execution. However, in some circumstances it would be useful for the programmer to explicitly change the policy at various points during a program's execution. This Cray-specific routine allows the programmer to dynamically change the wait policy (and potentially the contention policy). This addresses the situation when an application needs OpenMP for the first part of program execution, but there is a clear point after which OpenMP is no longer used. Unfortunately, the idle OpenMP threads still consume resources since they are waiting for more work, resulting in performance degradation for the remainder of the application. A passive-waiting policy might eliminate the performance degradation after OpenMP is no longer needed, but the developer may still want an active-waiting policy for the OpenMP-intensive region of the application. This routine notifies all threads of the policy change at

the same time, regardless of whether they are idle or active (to avoid deadlock from waiting and signaling threads using different policies).

CRAY_OMP_CHECK_AFFINITY

Set the **CRAY_OMP_CHECK_AFFINITY** variable to `TRUE` at execution time to display affinity binding for each OpenMP thread. The messages contain hostname, process identifier, OS thread identifier, OpenMP thread identifier, and affinity binding.

OpenMP Accelerator Support

The OpenMP 4.0 `target` directives are supported for targeting NVIDIA GPUs or the current CPU target. An appropriate accelerator target module must be loaded to use `target` directives.

OpenACC Use

OpenACC is a parallel programming model which facilitates the use of an accelerator device attached to a host CPU. The OpenACC API allows the programmer to supplement information available to the compilers in order to offload code from a host CPU to an attached accelerator device.

This release supports the *OpenACC Application Programming Interface, Version 2.0* standard developed by PGI, Cray Inc., NVIDIA, with support from CAPS entreprise.

Refer to the OpenACC home page at *http://www.openacc-standard.org*. Under the Downloads link, select the *OpenACC 2.0 Specification*.

For the most current information regarding the Cray implementation of OpenACC, see the `intro_openacc(7)` man page. See the `OpenACC.EXAMPLES(7)` man page for example OpenACC codes.

OpenACC Execution Model

The CPU host offloads compute intensive regions to the accelerator device. The accelerator executes parallel regions, which contain work sharing loops executed as *kernels* on the accelerator. The CPU host manages execution on the accelerator by allocating memory on the accelerator, initiating data transfer, sending code, passing arguments to the region, waiting for completion, transferring accelerator results back to the CPU host and releasing memory.

The accelerator on the Cray system supports multiple levels of parallelism. The accelerator executes a *kernel* composed of parallel threads or *vectors*. Vectors (threads) are grouped into sets called *workers*. Threads in a set of workers are scheduled together and execute together. *Workers* are grouped into larger sets called *gangs*. One or more *gangs* may comprise a kernel. To summarize, a *kernel* is executed as a set of *gangs* of *workers* of *vectors*.

The compiler determines the number of gangs/workers/vectors based on the problem and then maps the vectors, workers, and gangs onto the accelerator architecture. Specifying the number of gangs, workers, or vectors is optional but may permit tuning to a particular target architecture. The way that the compiler maps a particular problem onto a constellation of gangs, workers, and vectors which are then mapped onto the accelerator architecture is implementation defined.

OpenACC terminology is situated in the context of the PGAS programming model. In the PGAS model, there may be one or more Processing Elements (PEs) per XK node. Each PE is multi-threaded and each thread can execute vector instructions. The PGAS thread concept is not the same as the OpenACC thread concept.

OpenACC Memory Model

The memory on the accelerator is separate from host memory. Accelerator device memory is not mapped onto the host's virtual memory space. All data movement between host and accelerator memory is initiated by the host through the library functions that move data. Also, it is not assumed that the accelerator can access host memory, though it is supported by some devices. In this model, data movement between memories is managed by the compiler according to OpenACC directives. The programmer needs to be aware of device memory size, as well as memory bandwidth between host and device in order to effectively accelerate a region of code.

Current accelerators implement a weak memory model; they do not support memory coherence between operations executed by different *execution units* - an execution unit is a hardware abstraction which can execute one or more gangs. If an operation updates a memory location and another reads from the same location, or two operations store a value to the same location, the hardware may not guarantee repeatable results. Some potential errors of this type are prevented by the compiler, but it is possible to write an accelerator parallel region that produces inconsistent results. Memory coherence is guaranteed when memory operations referencing the same location are separated by an explicit barrier.

Map the OpenACC Programming Model onto Accelerator Components

The compiler maps the OpenACC execution model (kernels, gangs, workers, vectors) onto the accelerator architecture as described in the following sections.

Stream Multiprocessors (SM) and Scalar Processor (SP) cores

On the Cray XK system, there is one accelerator per node. The accelerator architecture is comprised of two main components - global memory and some number of streaming multiprocessors (SM). Each SM contains multiple scalar processor (SP) cores, schedulers, special-function units, and memory which is shared among all the SP cores. An SP core contains floating point, integer, logic, branching, and move and compare units. Each thread/vector is executed by a core. The SM manages thread execution.

The OpenACC execution model maps to the NVIDIA GPU hardware as follows (GPU terms are in parenthesis): One or more OpenACC kernels may execute on an GPU. The compiler divides a kernel into one or more gangs (blocks) of vectors (threads). Several concurrent gangs (blocks) of threads may execute on one SM depending on several factors, including memory requirements, compiler optimizations, or user directives. A single block (gang) does not span SMs and will remain on one SM until completion. When the SM encounters a block (gang), each gang (block) is further broken up into workers (warps) which are groups of threads to execute in parallel. Scheduling occurs at the granularity of the worker (warp). Individual threads within a warp start together and execute one common instruction at a time. If conditional branching occurs within a worker (warp), the warp serially executes each branch path taken causing some threads to wait until threads converge back to the same instruction. Data dependent conditional code within a warp usually has negative performance impact. Worker (warp) threads also fetch data from memory together and when accessing global memory, the accesses of the threads within a warp are grouped to minimize transactions. Each thread in a worker (warp) is executed on a different SP core.

There may be up to 32 threads in a worker (warp) - a limit defined by the hardware.

See the `intro_openacc(7)` man page for more detail on Partition Mapping.

Memory

There is a hierarchy of memory spaces used by OpenACC threads. Each thread has its own private local memory. Each gang of workers of threads has shared memory visible to all threads of the gang. All OpenACC threads running on a GPU have access to the same global memory. Global memory on the accelerator is accessible to the host CPU.

Mixed Model Support

OpenMP directives may appear inside of OpenACC data or host data regions only. OpenMP directives are not allowed inside of any other OpenACC directives.

OpenACC may not appear inside OpenMP directives. To have OpenACC directives nested inside of OpenMP constructs, place them in calls that are not inlined.

Compile with OpenACC

The CCE compiler recognizes OpenACC directives, by default. Use either the `ftn` or `cc` command to compile.

The CCE compiler does not produce CUDA code. It generates PTX (Parallel Thread Execution) instructions which are then translated into assembly.

Note the following interactions between directives and command line options.

- `-h [no]acc`

  `-h noacc` disables OpenACC directives.

- `-h [no]pragma`

  See *-h [no]pragma[=name[:name ...]]*.

- -h acc_model=*option* [:*option* ...]

  Explicitly controls the execution and memory model utilized by the accelerator support system. The option arguments identify the type of behavior desired. There are three option sets. Only one member of a set may be used at a time; however, all three sets may be used together.

  Default: `auto_async_kernel:fast_addr:no_deep_copy`

  option Set 1:

  auto_async_none Execute kernels and updates synchronously, unless there is an async clause present on the kernels or update directive.

  auto_async_kernel (Default) Execute all kernels asynchronously ensuring program order is maintained.

  auto_async_all Execute all kernels and data transfers asynchronously, ensuring program order is maintained.

- *option* Set 2:

  no_fast_addr Use default types for addressing.

  fast_addr (Default) Attempt to use 32 bit integers in all addressing to improve performance. Base addresses remain as 64 bit. The performance is improved by potentially using fewer registers and faster arithmetic for offset calculations. This optimization may result in incorrect behavior for codes that make use within accelerator regions of any of the following: very large arrays (offsets would require greater than 32 bits), very large array lower bounds (max offset plus lower bound is greater than 32 bits), bitfields/other bit operations.

- *option* Set 3:

  no_deep_copy Do not look inside of an object type to transfer sub-objects. Allocatable members of derived type objects will not be allocated on the device.

Module Support

To compile, ensure that `PrgEnv-cray` module is loaded and that it includes CCE 8.4 or later. Then, either load the `craype-accel-nvidia20` module for Fermi support or the `craype-accel-nvidia35` module for Kepler support.

The `craype-accel-host` module supports compiling and running an OpenACC application on the host X86 processor. This provides source code portability between systems with and without an accelerator. The accelerator directives are automatically converted at compile time to OpenMP equivalent directives.

Use either the `ftn` or `cc` command to compile.

Debug

Use either Alinea DDT or Rogue Wave TotalView.

The following applies to all debuggers:

- To enable debugging, compile use the `-g` option.
- When compiling with the debug option (`-g`), CCE may require additional memory from the accelerator heap, exceeding the 8MB default. In this case, there will be malloc failures during compilation. The environment variable *`CRAY_ACC_MALLOC_HEAPSIZE`* specifies the accelerator heap size in bytes. It may be necessary to increase the accelerator heap size to 32MB (33554432), 64MB (67108864), or greater by setting *`CRAY_ACC_MALLOC_HEAPSIZE`* accordingly. The accelerator heap size defaults to 8MB.
- Debug one rank/image/thread/PE per node.
- CCE does not generate CUDA code, but generates PTX code. Debuggers will not display CUDA intermediate code.
- To enter an OpenACC region using a debugger, breakpoints may be set inside the OpenACC region. It is not possible to do a single step into the region from the code immediately prior to the start of an OpenACC directive.

OpenACC Directives

For information on the OpenACC directives, see the *OpenACC 2.0 Specification* available at *http://www.openacc-standard.org*.

For the most current information regarding the Cray implementation of OpenACC, see the `intro_openacc(7)` man page. See the `OpenACC.EXAMPLES(7)` man page for example OpenACC codes.

Runtime Routines

Runtime routines defined by the standard specification are supported unless otherwise noted in the `intro_openacc(7)` man page.

Cray Specific Runtime Library Routines

The following routines are currently Cray specific. These interfaces are subject to change and their usage may result in non-portable code:

- void cray_acc_memcpy_to_host_async(void* *`host_destination`*, const void* *`device_source`*,size_t size, int *`async_id`*);

  Asynchronously copies *`size`* bytes from the accelerator source address to the host destination address; returns destination address. See async clause for explanation of *async_id*.

- void cray_acc_memcpy_to_device_async(void* *`host_destination`*, const void* *`device_source`*,size_t *`size`*, int *`async_id`*);

  Asynchronously copies *`size`* bytes from the accelerator source address to the host destination address; returns destination address. See async clause for explanation of *async_id*.

- bool cray_acc_get_async_info(int *`async_id`*, void* *`async_info`*);

Returns true if the *async_id* was found to have any architecture specific async information available. The user is responsible for ensuring that the *async_info* pointer points to a async structure from the underlying architecture. For an NVIDIA target this would be a CUDA Stream (CUstream).

CRAY_ACC_DEBUG Output Routines

When the runtime environment variable `CRAY_ACC_DEBUG` is set to 1, 2, or 3, CCE writes runtime commentary of accelerator activity to `STDERR` for debugging purposes; every accelerator action on every PE generates output prefixed with "ACC:". This may produce a large volume of output and it may be difficult to associate messages with certain routines and/or certain PEs.

With this set of API calls, the programmer can enable or disable output at certain points in the code, and modify the string that is used as the debug message prefix.

The `cray_acc_set_debug_*_prefix( void )` routines define a string that is used as the prefix, with the default being "ACC:". The `cray_acc_get_debug_*_prefix( void )` routines are provided so that the previous setting can be restored.

Output from the library is printed with a format string starting with "ACC: %s %s", where the global prefix is printed for the first `%s` (if not NULL), and the thread prefix is printed for the second %s. The global prefix is shared by all host threads in the application, and the thread prefix is set per-thread. By default, strings used in the `%s` fields are empty.

The C interface is provided by `omp.h`:

- `char *cray_acc_get_debug_global_prefix( void )`
- `void cray_acc_set_debug_global_prefix( char * )`
- `char *cray_acc_get_debug_thread_prefix( void )`
- `void cray_acc_set_debug_thread_prefix( char * )`

To enable debug output, set level from 1 to 3, with 3 being the most verbose. Setting a level less than or equal to 0 disables the debug output. The get version is provided so the previous setting can be restored. The thread level is an optional override of the global level.

- `int cray_acc_get_debug_global_level( void )`
- `void cray_acc_set_debug_global_level( int level )`
- `int cray_acc_get_debug_thread_level( void )`
- `void cray_acc_set_debug_thread_level( int level )`

Environment Variables

The following are environment variables are defined by the API specification:

- `ACC_DEVICE_NUM`
- `ACC_DEVICE_TYPE`

The following environment variable is Cray specific:

- `CRAY_ACC_MALLOC_HEAPSIZE`

  Specifies the accelerator heap size in bytes. The accelerator heap size defaults to 8MB. When compiling with the debug option (`-g`), CCE may require additional memory from the accelerator heap, exceeding the 8MB

default. In this case, there will be `malloc` failures during compilation. It may be necessary to increase the accelerator heap size to 32MB (33554432), 64MB (67108864), or greater.

- `CRAY_ACC_DEBUG`

  When set to 1, 2, or 3 (most verbose), writes runtime commentary of accelerator activity to `STDERR` for debugging purposes. There is also an API which allows the programmer to enable/disable debug output and set the output message prefix from within the application.

OpenACC Examples

See the `OpenACC.EXAMPLES(7)` man page for example OpenACC codes.

Unified Parallel C (UPC)

This release supports the UPC Language Specification, Version 1.3. The UPC 1.3 standard is discussed on the UPC specification website, *http://code.google.com/p/upc-specification*.

This chapter describes the Cray specific UPC functionality available in CCE, and features of the specification which are implementation defined. Also see `intro_pgas(7)`, or refer to the appropriate UPC man page.

Be familiar with UPC and understand the differences between the published UPC Introduction and Language Specification paper and the current UPC specification. If there's no familiarity with UPC, refer to the UPC home page at *http://upc.gwu.edu*. Under the Publications link, select the Introduction to UPC and Language Specification paper. This paper is slightly outdated but contains valuable information about understanding and using UPC. The UPC home page also contains, under the Documentation link, the UPC Language Specification paper.

UPC allows for explicitly specifying parallel programming through language syntax rather than library functions such as those used in MPI and SHMEM by allowing for the reading and writing of memory of other processes with simple assignment statements. Program synchronization occurs only when explicitly programmed; there is no implied synchronization.

UPC is a dialect of the C language. It is not available in C++.

UPC allows for the maintainence of a view of the program as a collection of threads operating in a common global address space without the burden of the details of how parallelism is implemented on the machine (for example, as shared memory or as a collection of physically distributed memories).

UPC data objects are private to a single thread or shared among all threads of execution. Each thread has a unique memory space that holds its private data objects, and access to a globally-shared memory space that is distributed across the threads. Thus, every part of a shared data object has an affinity to a single thread.

Cray UPC is compatible with MPI. While it may work in some cases, mixing language-based PGAS with SHMEM is not officially supported.

UPC 1.3 supports a parallel I/O model which provides control over file synchronization. However, if use is continued on the regular C I/O routines, supply the controls as needed to remove race conditions. File I/O under UPC is very similar to standard C because one thread opens a file and shares the file handle, and multiple threads may read or write to the same file.

Cray UPC supports GASP instrumentation. GASP instrumentation enables the use of external performance tools, such as the Parallel Performance Wizard (PPW) from the University of Florida. For more information on GASP and PPW, see *http://gasp.hcs.ufl.edu* and *http://ppw.hcs.ufl.edu*. To instrument for GASP, refer to the command line option *-h gasp[=opt[:opt]]*.

Predefined Macros

The following UPC 1.3 preprocessor macros are supported and defined as follows:

- __UPC__: 1
- __UPC_VERSION__: 201309 (corresponds to the date that 1.3 spec is published)
- UPC_MAX_BLOCK_SIZE: 1073741823

- __UPC_DYNAMIC_THREADS__: 1 (if compiling for dynamic threads, otherwise undefined)
- __UPC_STATIC_THREADS__: 1 (if compiling for static threads, otherwise undefined)
- __UPC_COLLECTIVE__: 1
- __UPC_TICK__: 1
- __UPC_CASTABLE__: 1
- __UPC_IO__: 1
- __UPC_NB__: 1

False Sharing

There is a false sharing hazard when referencing shared `char` and `short` integers.

If two PEs store a `char` or `short` to the same 64-bit word in memory without synchronization, incorrect results can occur. It is possible for one PE's store to be lost. This is because these stores are implemented by reading the entire 64-bit word, inserting the `char` or `short` value and writing the entire word back to memory.

The following output is a result of two PEs writing two different characters into the same word in memory without synchronization:

```
                  Register     Memory
Initial Value                  0x0000
PE 0 Reads          0x0000     0x0000
PE 1 Reads          0x0000     0x0000
PE 0 Inserts 3      0x3000     0x0000
PE 1 Inserts 7      0x0700     0x0000
PE 0 Writes         0x3000     0x3000
PE 1 Writes         0x0700     0x0700
```

Notice that the value stored by PE 0 has been lost. The final value intended was 0x3700. This situation is referred to as false sharing. It is the result of supporting data types that are smaller than the smallest type that can be individually read or written by the hardware. UPC programmers must take care when storing to shared `char` and `short` data that this situation does not occur.

Compile and Link UPC Code

Compiling a PGAS application (UPC, Fortran 2008) requires the PrgEnv-cray module to be loaded.

The `-hupc` option is required to enable recognition of UPC syntax because it is not part of the standard C language.

The `-X npes` option can optionally be used to define the number of threads to use and statically set the value of the `THREADS` constant. See *-X npes* for requirements regarding the use of the `-X npes` option.

The following command creates an executable file:

```
% cc -hupc hello.c -o hello
```

An executable can be created by linking together various object files that were generated from source code written in standard C, UPC, and Fortran. Either `cc` or `ftn` can be used to link the object files:

```
% cc -hupc x.o y.o z.o
```

```
% ftn x.o y.o z.o
```

For dynamic linking, add the `-dynamic` option. For information about linking PGAS applications to use huge pages, see the `intro_hugepages(1)` man page. The Cray implementation of UPC supports adding GASP instrumentation to UPC codes. To instrument for GASP, refer to the command line option *-h gasp[=opt[:opt]]*.

Launch a UPC Application

After compiling the UPC code, run the program using the `aprun` command.

Launch the application using 128 PEs:

```
% aprun -n 128 ./hello
```

If using the `-X npes` compiler option, the same number of threads in the `aprun` command must be specified. The processing elements specified by *npes* are compute node cores/PEs.

By default, each PE reserves 64 MB of symmetric heap space. To increase or decrease this amount, set the `XT_SYMMETRIC_HEAP_SIZE` environment variable to the desired number of bytes. The suffixes K, M, and G are permitted to simplify requests for large values:

```
% export XT_SYMMETRIC_HEAP_SIZE=512M
% aprun -n 128 ./hello
```

The UPC run time system uses GNI and DMAPP (low level libraries) to implement a logically shared, distributed memory programming model. The symmetric heap is mapped onto `hugepages` by DMAPP. It is advisable to also map the static data and/or private heap onto huge pages. See the `intro_hugepages(1)` man page.

Cray Extensions

Cray extensions to UPC that are not part of the UPC Language Specification 1.3 are listed here.

A number of former extensions to UPC 1.2 have been standardized in UPC 1.3, including non-blocking bulk copies (`upc_nb.h`), privatizability (`upc_castable.h`) and timing (`upc_tick.h`) interfaces. These interfaces have been removed from the `upc_cray.h` header and moved into new headers as required by the UPC 1.3 specification. Additionally, some of the semantics and interfaces have been changed slightly, so existing users of these interfaces may need to update their applications.

Team Collectives

The following interfaces, declared in `upc_collective_cray.h`, provide common collective operations on a subset (team) of threads. These are loosely based on the UPC Collectives Library 2.0 proposal, with changes to argument ordering to better match existing practice in UPC and no explicit initialization.

- `CRAY_UPC_TEAM_ALL`
- `CRAY_UPC_TEAM_NODE`
- `cray_upc_op_create(3c)`

- cray_upc_op_free(3c)
- cray_upc_type_size(3c)
- cray_upc_team_rank(3c)
- cray_upc_team_size(3c)
- cray_upc_team_split(3c)
- cray_upc_team_free(3c)
- cray_upc_team_barrier(3c)
- cray_upc_team_allreduce(3c)
- cray_upc_team_reduce(3c)

Node Affinity

Include upc_cray.h to use these extensions.

upc_nodeof() Returns the index of the node of the thread that has affinity to the shared object pointed to by ptr. Similar to upc_threadof().

NODES NODES is an expression with a value of type int; it specifies the number of nodes and has the same value on every thread in the job.

Similar to THREADS, but evaluates to the number of nodes used by the application, equal to the ceiling of the aprun -n value divided by the -N value.

MYNODE MYNODE is an expression with a value of type int; it specifies the unique node index associated with the current thread and has the same value on all threads that are located on the same node.

Similar to MYTHREAD, but evaluates to a node number in the range 0 to NODES - 1, inclusive.

C++ Libraries

Most standard C++ features are supported, except the following:

- String classes using basic string class templates with wide character types or that use the `wstring` standard template class.
- I/O streams using wide character objects.
- File-based streams using file streams with wide character types (`wfilebuf`, `wifstream`, `wofstream`, and `wfstream`)
- Multiple localization libraries; Cray C++ supports only one locale

The C++ standard provides a standard naming convention for library routines. Therefore, classes or routines that use wide characters are named appropriately. For example, the `fscanf` and `sprintf` functions do not use wide characters, but the `fwscanf` and `swprintf` function do.

Coarray C++ Use

Coarray C++ is a template library that implements the coarray concept for Partitioned Global Address Space (PGAS) programming in C++. The template library specifications are contained on a set of `*.html` pages that the CCE installation copies to `/opt/cray/cce/version/doc/html/` on the Cray platform; they may be copied to any location which provides HTML web content for the site, or any location that can be accessed by site local web browsers.

The coarray concept used in Coarray C++ is intentionally very similar to Fortran (ISO/IEC 1539-1:2010) coarrays. Users familiar with Fortran coarrays will notice that terminology and even function names are identical, although the syntax follows C++ conventions.

A coarray adds an additional dimension, called a *codimension*, to a normal scalar or array type. The codimension spans instances of a Single-Program Multiple-Data (SPMD) application, called images, such that each image contains a slice of the coarray equivalent in shape to the original scalar or array type. Each image has immediate access via processor loads and stores to its own slice of the coarray, which resides in that image's local partition of the global address space. By specifying an image number in the cosubscript of the codimension, each image also has access to the slices residing in other images' partitions.

Images are an orthogonal concept to threads, such as those provided by C++11 or OpenMP. Threads are used for shared memory programming where each thread has immediate access to the address space of a single process and possibly some thread-local storage to which only it has access. Images are a broader concept intended to provide communication among cooperating processes that each have their own address space. The mechanism for this cooperation varies by implementation. Typically it involves network communication between processes that have arranged to have identical virtual memory layouts. This communication is one-sided such that a programmer can have an image read or write data that belongs to a different image without writing any code for the second image. Note that images and threads may coexist in the same application; a large networked system with multicore nodes could use coarrays to communicate among nodes but use threads within each node to exploit the multicore parallelism.

In Coarray C++, a coarray is presented as a class template that collectively allocates an object of a specified type within the address space of each image. The coarray object is responsible for managing storage for the object that it allocates. When used in an expression context, the coarray object automatically converts to its managed object so that an image can access its own slice of the coarray without using special syntax. Accessing a slice that belongs to a different image requires specifying the image number as a cosubscript in parenthesis immediately following the coarray object, before any array subscripts. Therefore, the codimension is the slowest-running array dimension, just like Fortran.

The subscript order is backwards from Fortran because in Fortran the slowest-running dimension is rightmost whereas in C++ it is leftmost.

In addition to providing the fundamental ability to allocate and access a coarray, Coarray C++ provides image synchronization, atomic operations, and collectives.

Although this chapter presents Cray's implementation, Coarray C++ is designed to allow portable applications to be written for a variety of computing platforms in the sense that the template library interface is platform independent and can be compiled by any C++03 (ISO/IEC 14882:2003) or C++11 (ISO/IEC 14882:2011) compliant compiler. The implementation of the template library is likely to differ for each platform due to different transport layers (e.g., shared memory or various networks) for communicating data between images.

Compile Coarray C++

The following program is the Coarray C++ equivalent of the classic "Hello World" program. The header file `coarray_cpp.h` provides all Coarray C++ declarations within namespace `coarray_cpp`. Normally a program imports all of the declarations into its namespace with a using directive, but having the namespace gives the programmer flexibility to deal with name conflicts.

```
#include <iostream>
#include <coarray_cpp.h>
using namespace coarray_cpp;
int main( int argc, char* argv[] )
{
    std::cout << "Hello from image " << this_image()
    << " of " << num_images() << std::endl;
    return 0;
}
```

The program is compiled with the Cray compiler and executed using four images as follows:

```
> module load PrgEnv-cray
> CC -o hello hello.cpp
> aprun -n4 ./hello
Hello from image 0 of 4
Hello from image 1 of 4
Hello from image 2 of 4
Hello from image 3 of 4
```

Declare and Access Coarrays

The general form of a coarray declaration is:

```
coarray<T> name;
```

Where *T* is the type of the object that will be allocated in the address space of each image.

A coarray declaration may appear anywhere that a C++ object can be declared. Therefore, a coarray may be declared as a global variable, local variable, static local variable, or as part of a struct or class. It may be allocated statically or dynamically. The only restriction is that a coarray allocation must be executed collectively by all images. The C++ language ensures that this restriction is met for global and static local coarray declarations, but the programmer is responsible for ensuring that local and dynamically-allocated coarrays are declared collectively. For example:

```
coarray<int> x; // global
void
foo( void )
{
   static coarray<int> y; // static local
   coarray<int> z; // local
   coarray<int>* p = new coarray<int>; // dynamically allocated
   ...
   delete p;
}  // z is automatically destroyed here
```

Basic Types

A coarray of a basic C++ type is the simplest kind of coarray. Each image has an instance of the basic type that is managed by its coarray object. A coarray of type int is declared as:

```
coarray<int> x;
```

The declaration may pass an initial value to the constructor. Different images may pass different initial values:

```
coarray<int> x(2);
```

The initializer syntax below is not supported. If it were permitted, then automatic conversion from int to coarray<int> would be allowed, which would loosen type checking and lead to unexpected collective allocations:

```
coarray<int> x = 2;
```

This coarray object will behave as if it were the int that it manages. Assigning to the coarray object will assign a value to the int that is managed by the coarray object:

```
x = 42;
```

Likewise, using the coarray object in any expression where an int is expected shall read the value of the managed int:

```
int y = x + 1;
```

If the coarray object needs to be used in an expression where no particular type is expected, then the managed object can be accessed explicitly via empty parenthesis:

```
// prints the address of the coarray object
std::cout << &x << std::endl;
// prints the address of the int managed by the coarray object
std::cout << &x() << std::endl;
```

Accessing an int that is managed by another image requires specifying the image number within the parenthesis:

```
x(5) = 42; // set x = 42 within the address space of image 5
int y = x(2); // obtain the value of x from the address space of image 2
```

Finally, consider an enhanced version of the Hello World program. In this program, all images write their image number to their local object and then call `sync_all()`, which synchronizes control flow across all images. After the `sync_all()`, each image computes the image number of its left and right neighbors in the image space and prints the values that were written by its neighbors.

```
#include <iostream>
#include <coarray_cpp.h>
using namespace coarray_cpp;
int main( int argc, char* argv[] )
{
    coarray<int> x;
    x = this_image();
    sync_all();
    const int left = ( this_image() - 1 ) % num_images();
    const int right = ( this_image() + 1 ) % num_images();
    std::cout << "Hello from image " << x << "
```

```
          where x(left) = " << x(left) << " and x(right) = "
          << x(right) << std::endl;
    return 0;}

> CC -o hello2 hello2.cpp
> aprun -n4 ./hello2
Hello from image 0 where x(left) = 3 and x(right) = 1
Hello from image 3 where x(left) = 2 and x(right) = 0
Hello from image 2 where x(left) = 1 and x(right) = 3
Hello from image 1 where x(left) = 0 and x(right) = 2
```

Arrays

A coarray of an array type gives every image an array of the same shape. An example of a statically-sized coarray is below. The complete array type, including all extents, is provided as the coarray template's type argument:

```
// Declares a coarray of an array of 10 arrays of 20 ints
coarray<int[10][20]> x;
```

The following declaration is very different:

```
// Declares an array of 10 arrays of 20 coarrays
// of type int. Legal, but very inefficient!
coarray<int> bad[10][20];
```

A coarray of a multidimensional array type is not achieved via nested coarray types. Although such declarations are legal, they are strange and not particularly useful:

```
// Declares a coarray of an array of 10 coarrays of arrays of 20 ints
coarray< coarray<int[20]>[10] > weird;
```

In a dynamically-sized coarray declaration, the extent of the leading dimension is left unbounded. The size of this extent cannot be part of the template type because it is not known at compile time. Instead, the size is passed as a constructor argument:

```
coarray<int[][20]> y(n); // each image must pass the same value
```

Later, the extent of the leading dimension can be extracted from the coarray object via the extent() member function:

```
size_t y_extent = y.extent();
```

An individual element of the local array managed by the coarray object is accessed by applying subscripts directly to the coarray object. When accessing part of the coarray managed by another image, the cosubscript appears in parenthesis before the subscripts:

```
x[4][5] = 1; // set x[4][5] = 1 within this image's address space
y(3)[6][7] = 2; // set y[6][7] = 2 within the address space of image 3
```

Pointers

A coarray of pointers is typically used to implement a "ragged array" where different images need to allocate a different amount of memory as part of the same coarray. An example of a coarray of pointers is:

```
coarray<int*> x;
```

Each image allocates additional memory independently from the collective allocation of the coarray object itself:

```
x = new int[n]; // n usually varies per image
```

Due to the independent allocations, the allocated memory might not be located at the same address within every image's address space. Therefore, accessing the data requires an additional read of the pointer from the target image before a normal read or write can occur. This additional read happens automatically as part of the usual syntax for accessing the data:

```
x(i)[3] = 4; // set x[3] = 4 within the address space of image i
```

The address stored within the pointer may be valid only on the allocating image, unless the program is careful to target the pointer at only symmetric virtual addresses. Great care should be taken with the following code pattern:

```
int* p = x(i); // get an address from image i
p[3] = 4;      // and dereference it on this image
```

Finally, the program must ensure prior to performing any accesses that other images have allocated their memory:

```
coarray<int*> x;x = new int[n];
sync_all();
x(i)[3] = 4;
```

Structs, Unions, and Classes

A coarray of a struct, union, or class behaves like a coarray of a basic type when the entire object is accessed, however special syntax is required for member access due to limitations of C++ operator overloading:

```
struct Point { int x, y; };

coarray<Point> pt;
Point p;

pt = p;    // set pt = p in this image's address space
pt(2) = p; // set pt = p within the address space of image

2pt->x = 0;  // set pt.x = 0 in this image's address space
pt().x = 0; // alternate syntax

// set pt.x = 1 within address space of image i
pt(i).member( &Point::x ) = 1;
```

Calling a member function of an object that resides in the address space of another image (i.e., a remote procedure call) is not supported. By default, when a struct, union, or class is copied between images, it is treated as a Plain Old Data (POD) type such that a bitwise copy occurs. This behavior is not appropriate if the type contains pointers to allocated data. The default behavior can be changed by creating a specialization of

`coarray_traits` where `is_trivially_gettable` is false. C++ requires that the specialization be placed in the same namespace as the general template:

```
struct my_string {
    char* data;
    size_t length;
};

namespace coarray_cpp {
    template < >
    struct coarray_traits<my_string> {
        static const bool is_trivially_gettable = false;
        static const bool is_trivially_puttable = false;
    };
}
```

When `is_trivially_gettable` is false for a type, Coarray C++ expects the type to have a special constructor and a special assignment operator to facilitate reading an object from a remote image:

```
struct my_string {
  char* data;
  size_t length;

  // remote constructor
  my_string( const_coref<my_string> ref );
  // remote assignment operator
  my_string& operator = ( const_coref<my_string> ref );
};
```

The role of the remote constructor or remote assignment operator is to read the POD parts of the object from the other image, use that data to calculate how much memory needs to be allocated, allocate the memory, then read the rest of the object into the newly allocated memory.

Typically, if `is_trivially_gettable` is false for a type, then `is_trivially_puttable` should also be false. When `is_trivially_puttable` is false for a type, a compile time error will occur the program attempts to copy an instance of the type to another image.

Type System

The Coarray C++ type system is modeled closely on the C++ type system. In addition to the coarray type that extends the C++ array concept across images, there are *coreferences* and *copointers* that extend the C++ concepts of references and pointers to refer to objects on other images.

Coreferences

A coreference is returned when a cosubscript is applied to a coarray. Like a C++ reference, a coreference is always associated with an object, called its referent, can never be rebound to a different object, and can never be null. Typically a coreference is either immediately converted to its referent or subscripted, such that it is not necessary to declare a coreference and its fleeting presence can be ignored. Nevertheless, explicit coreferences are useful in some situations. Suppose that a function needs to have access to an object in another image's address space, but does not need to know anything about the coarray containing the object. For example:

```
void foo( coref<int> );
int main( int argc, char* argv[] ){
    coarray<int> x;
    coarray<int[10]> y;
```

```
    ...
    foo( x(2) );
    foo( y(3)[4] );
    ...
return 0;
}
```

In the above code, function foo can access an int that is part of either x or y even though x and y have different shapes. If foo were to require a coarray parameter instead, then it could accept either x or y but not both because the coarrays have different types. Furthermore, foo's coreference parameter makes it clear to someone reading the code that the function's effect is narrow, limited to one object instead of an entire coarray. Two other uses of coreferences are to operate on coarray slices that are larger than a single object and to move data in bulk between images. To make these techniques more useful, coreferences can be created for local objects:

```
int main( int argc, char* argv[] ){
    coarray<int[5][10]> x;
    int local[10];
    coref<int[10]> local_ref( local );
    ...
    // local[0...9] = x(2)[1][0...9]
    local_ref = x(2)[1];
    ...
    // x(3)[4][0...9] = local[0...9]
    x(3)[4] = local_ref;
    ...
    return 0;
}
```

For convenience, the `make_coref` and `make_const_coref` functions create coreferences for local objects without requiring the programmer to write the type of the local object:

```
int main( int argc, char* argv[] )
{
    coarray<int[5][10]> x;
    int local[10];
    ...
    // local[0...9] = x(2)[1][0...9]
    make_coref( local ) = x(2)[1];
    ...
    // x(3)[4][0...9] = local[0...9]
    x(3)[4] = make_const_coref( local );
    ...
    return 0;
}
```

A `const_coref` behaves exactly like a `coref` except that it cannot be used to modify its referent.

Copointers

A coreference can be converted to a copointer by calling its address function; the address-of operator is not overloaded. Local pointers are automatically convertible to copointers. Unlike coreferences, a copointer can be reassociated and can be unassociated or null. Arithmetic on a copointer changes the address to which it points but never changes the image to which it points. Comparisons between two copointers are allowed provided that

both copointers point to the same image. Copointers can be used as iterators with standard C++ function templates. For example, the following code will not assert:

```
int
main( int argc, char* argv[] )
{
    coarray<int[10]> x;
    const size_t left = ( this_image() - 1 ) % num_images();
    const size_t right = ( this_image() + 1 ) % num_images();
    coptr<int> begin = x(right)[0].address();
    // Apply a standard algorithm, using a coptr as an iterator.
    coptr<int> end = x(right)[10].address();
    std::fill( begin, end, image );
    sync_all();
    for ( int i = 0; i < 10; ++i ) {
        assert( x[i] == left );
    }
    return 0;
}
```

They can be used to form linked lists spanning images. The list even can include links that point to local data:

```
#include <iostream>
#include <coarray_cpp.h>
using namespace coarray_cpp;

template < typename T >
struct Link {
    T data;
    coptr< Link<T> > next;
};

coarray< Link<int> > global_links;

int main( int argc, char* argv[] )
{
    Link<int> local_link;
    global_links->data = 2 * this_image();
    global_links->next = &local_link;
    local_link.data = 2 * this_image() + 1;
    if ( this_image() < num_images() - 1 ) {
        local_link.next = global_links(this_image() + 1).address();
    }
    else {
        local_link.next = 0;
    }
    sync_all(); // ensure every image has setup the data

    if ( this_image() == 0 ) {
       for ( coptr< Link<int> > p = global_links(0).address();
             p != NULL; p = p->member( &Link<int>::next ) ) {
           std::cout << p->member( &Link<int>::data ) << std::endl;
       }
    }
    // ensure local_link is not destroyed before it's read by image 0
    sync_all();
    return 0;
}
```

Compiling and executing the above program:

```
> CC -o list list.cpp
> aprun -n4 ./list
```

A `const_coptr` behaves exactly like a `coptr` except that it cannot be used to modify its target.

shape_cast

Various different array types have the same number of elements even though they have a different shape. For example, `int[100]`, `int[2][50]`, and `int[2][2][25]` all have 100 elements. A reference or pointer to a coarray of one of these types can be reinterpreted as a coarray of any of the others via a `shape_cast`, which has the same syntax as the standard C++ `static_cast`, `dynamic_cast`, `reinterpret_cast`, and `const_cast`. A `shape_cast` converts between coarray types of the same ultimate type that have different shapes. For example, a shape_cast cannot be used to reinterpret a `coarray<int[100]>&` as a `coarray<float[100]>&`; that conversion will throw a `std::bad_cast` exception. A `shape_cast` can be used to convert to a smaller shape but not to a larger shape. For example, a `coarray<int[100]>&` may be converted to a `coarray<int[50]>&`, in which case the new coarray can access only the first 50 elements of the original, but it may not be converted to a `coarray<int[200]>&` because that requires more storage and will throw a `std::bad_cast` exception. The example code below shows various legal `shape_casts`:

```
#include <cassert>
#include <iostream>
#include <coarray_cpp.h>

using namespace coarray_cpp;

void foo( const coarray<int[]>& y ) { }
void foo10( const coarray<int[10]>& y ) { }
void foo5( const coarray<int[][5]>& y ) { }
void foo10_5( const coarray<int[10][5]>& y ) { }
void foo50( const coarray<int[50]>& y ) { }
int
main( int argc, char* argv[] )
{
    int extent = 10;
    coarray<int[10]> x_10_s;
    coarray<int[]> x_10_d(extent);
    coarray<int[10][5]> x_10_5_s;
    coarray<int[][5]> x_10_5_d(extent);
    coarray<int> y;

    // Perform all valid combinations of passing the coarrays to the functions,
    // using shape_cast when necessary.
    foo( x_10_s );
    foo( x_10_d );
    foo( shape_cast<int[]>( x_10_5_s ) );
    foo( shape_cast<int[]>( x_10_5_d ) );
    foo10( x_10_s );
    foo10( x_10_d );
    foo5( shape_cast<int[2][5]>( x_10_s ) );
    foo5( shape_cast<int[][5]>( x_10_d ) );

    foo5( x_10_5_s );
    foo5( x_10_5_d );
    foo10_5( x_10_5_s );
    foo10_5( x_10_5_d );
```

```
    foo50( shape_cast<int[50]>( x_10_5_s ) );
    foo50( shape_cast<int[50]>( x_10_5_d ) );

    // Trivial reshape to same shape.
    shape_cast<int>( y );

    // shape_cast from scalar to array.
    shape_cast<int[1]>( y );

    // shape_cast from array to scalar.
    shape_cast<int>( x_10_s );

    // shape_cast to smaller array.
    shape_cast<int[5]>( x_10_s );

    // shape_cast to larger array.
    bool passed = false;
    try {
        shape_cast<int[25]>( x_10_s );
    } catch ( std::bad_cast& e ){
        passed = true;
    }
    assert( passed );

    return 0;
}
```

Control Flow and Synchronization

Write SPMD Code

Coarray C++ follows the Single-Program Multiple-Data model where all images begin executing the same main program but may operate on different data. Conditional code is used to restrict execution to certain images:

```
#include <iostream>
#include <coarray_cpp.h>

using namespace coarray_cpp;

int main( int argc, char* argv[] )
{
    if ( this_image() % 2 == 0 ){
       std::cout << "Hello from even image "
          << this_image() << std::endl;
    }
    else {
        std::cout << "Hello from odd image "
           << this_image() << std::endl;
    }
    return 0;
}
> aprun -n4 ./a.out
Hello from odd image 3
Hello from even image 0
Hello from even image 2
Hello from odd image 1
```

Barriers

A `sync_all()` ensures that all images must execute a `sync_all()` before any image may proceed beyond the `sync_all()` which it executed. It is not required that all images execute exactly the same `sync_all()` in the source code, just that they must execute some `sync_all()`. Failure of all images to participate will cause deadlock.

Function Calls

A coarray may be passed to a function via a reference or a pointer, but may not be passed by value. If a coarray could be passed by value, the call would have to be collective. There would be a collective allocation of a temporary coarray, the data within the original coarray would need to be copied into the temporary coarray, and eventually the temporary coarray would need to be collectively destroyed. Pass by value is expensive and there are better alternatives, like passing a coarray as a const reference, so it is a compile-time error. No matter how a coarray parameter is declared, the type of the actual argument must agree. Automatic conversions are provided between bounded and unbounded arrays; a conversion from unbounded to bounded performs a run-time check to ensure that the extents match and may throw a `mismatched_extent_error` exception.

coatomic

The coatomic template is similar to the C++11 `std::atomic` template, but provides operations that are atomic with respect to images rather than threads. Specializations exist for all basic types and the same operations are supported as for the C++11 `std::atomic` template. Similar convenience typedefs are provided as well so that, for example, `coatomic_long` can be used in place of `coatomic<long>`.

```
coarray< coatomic<long> > x; // or coarray<coatomic_long>

x(i) ^= 3; // atomic update x = x ^ 3 on image i

long old_value = x(i)++; // atomic increment, saving the old value

long new_value = ++x(i); // atomic increment, saving the new value
```

coevent

A *coevent* permits point-to-point synchronization between images. It wraps a coatomic_long that acts as a counter and provides two operations, post and wait. Post atomically increments the counter and wait blocks execution of the calling image until it can atomically decrement the counter to a non-negative value.

```
coarray<coevent> x;

if ( this_image() == 0 ) {
   // do something, then notify image 1
   x(1).post();
}
else if ( this_image() == 1 ) {
    // wait for notification from another image
    x().wait(); // then do something
}
```

comutex

A comutex provides mutual exclusion. The lock function blocks until the mutex can be acquired and the unlock function releases the mutex. The try_lock function attempts to acquire the lock and returns a true upon success.

```
coarray<comutex> m;

m(i).lock();
// critical section, typically guarding access to data on image i
m(i).unlock();
```

Collectives

Coarray C++ provides broadcast and reduction collectives

cobroadcast

`cobroadcast` replicates the value of a coarray on one image across all other images.

```
#include <cassert>
#include <iostream>
#include <coarray_cpp.h>

using namespace coarray_cpp;

int
main( int argc, char* argv[] )
{
    coarray<int> x;
    size_t image = this_image();
    size_t n = num_images();
    if ( image == 0 ) {
        x = 42;
    }
    sync_all();

    // Make x on every image equal the x on image 0.
    cobroadcast( x, 0 );
    sync_all();
    assert( x == 42 );
    return 0;
}
```

coreduce

`coreduce` coreduce applies a function across the coarray values of all images. For convenience, template specializations of `coreduce` are provided for the addition, min, and max operations from the C++ functional header. Implementations are likely to provide optimized versions of at least these reductions.

```
#include <cassert>
#include <iostream>
#include <coarray_cpp.h>
using namespace coarray_cpp;
int
main( int argc, char* argv[] )
{
   coarray<int> sum;
```

```
    coarray<int> min;
    coarray<int> max;
    size_t image = this_image();
    size_t n = num_images();
    sum = image;
    min = image;
    max = image;

    sync_all();

    cosum( sum ); // equivalent to coreduce( sum, std::plus<int> )
    comin( min ); // equivalent to coreduce( min, std::less<int> )
    comax( max ); // equivalent to coreduce( max, std::greater<int> )
    sync_all();

    assert( sum == ( n * ( n - 1 ) / 2 ) );
    assert( min == 0 );
    assert( max == ( n - 1 ) );
    return 0;
}
```

Exceptions

Coarray C++ throws standard C++ exceptions, like `std::bad_cast`, but also throws some special exceptions for coarray-specific errors.

`invalid_image_error`	This exception is thrown whenever a cosubscript is invalid. For example, given a coarray x in a program executed with 4 images, x(4) triggers an exception because the only valid image numbers are 0, 1, 2, and 3.
`invalid_put_error`	This exception is thrown whenever a user-defined type is copied to a different image, but that type has `coarray_traits` that specify that it is not trivially puttable.
`mismatched_extent_error`	This exception is thrown when two arrays in an array assignment have a different shape.
`mismatched_image_error`	This exception is thrown when two copointers are compared or subtracted, but the copointers point to objects on different images.

Memory Consistency Model

`atomic_image_fence()`

The `atomic_image_fence()` function is the Coarray C++ equivalent of the C++11 `std::atomic_thread_fence()` function. It has the same behavior with respect to images as `std::atomic_thread_fence()` has with respect to threads. Typically, it is used to ensure that all memory accesses made by the calling image are visible to all images before performing subsequent memory accesses.

Accesses within a Single Image

The effect of two memory accesses made by an image to its own address space is governed by the C++ memory consistency model. The C++ memory consistency model depends on which version of the C++ standard is implemented by the compiler. In general, a C++03 compiler assumes that an image is single-threaded and offers no memory consistency guarantees if multiple threads perform the accesses, whereas a C++11 compiler provides

a detailed memory consistency model that can be used to reason about the effect of memory accesses within a multithreaded image.

Accesses to Other Images

Multi-byte Accesses

A memory access of an object of size N bytes shall be treated as if it was performed as N arbitrarily ordered single-byte memory accesses. For example, the target image of a write shall not rely on the Nth byte being written last to detect whether the full object has been written.

From Different Images

The execution of a program contains a data race if it contains two conflicting actions in different images, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior. For example, if two images both write to the same object without any synchronization:

```
if ( this_image() == 0 ) {
    x(i) = 0;
}else if ( this_image() == 1 ) {
    x(i) = 1;
}
```

Then the final value of the object is undefined. Various forms of synchronization can impose a specific order, such as in this example:

```
if ( this_image() == 0 ) {
    x(i) = 0;
}

sync_all();

if ( this_image() == 1 )
 { x(i) = 1;
}
```

Where the assignment by image 0 happens before the assignment by image 1 because of the `sync_all()`.

Two atomic operations issued by different images to the same coatomic object have the same ordering relationship as two C++11 threads that perform the same atomic operations on the same object.

From the Same Image

Two memory accesses issued by the same image to non-conflicting memory addresses are unordered.

Two memory accesses issued by the same image to conflicting memory addresses within the address space of a single, different image shall have the same order as if they were made within the issuing image's address space. For example, in the following code:

```
x(i) = 1;int y = x(i);
```

the value of y will be 1 provided that there are no data races. Therefore, a Coarray C++ implementation for a shared memory system could inline x(i) as a direct memory access, allowing the compiler to make the following optimization (forward substitution):

```
x(i) = 1;int y = 1;
```

For distributed memory systems, providing this ordering guarantee is unfortunately somewhat onerous, but it is consistent with ordering guarantees of other PGAS languages, namely UPC and Fortran. Two memory accesses issued by an image to the same distant memory location typically will pass through the issuing processor's memory system, a high-speed communication network, and finally the target processor's memory system. Each hardware component is likely to contain multiple data pathways to increase bandwidth and resiliency, such that two memory accesses traveling on different pathways could bypass each other. Providing the ordering guarantee may require constraining two memory accesses to the same target location to always take the same hardware path to prevent bypass. Alternatively, software can track outstanding memory accesses and defer issuing an access if there is a conflict; however, software ordering adds overhead to each memory access to check for conflicts as well as storage overhead to track the accesses.

Blocking Versus Non-blocking Accesses

When an image makes a blocking read or write access, it does not proceed to execute its next operation until the access fully completes. By contrast, a non-blocking read or write access permits an image to proceed to execute its next operation before the access fully completes and provides some mechanism for ensuring that the operation has completed later.

Writes (Puts)

Neither the target image nor any other image besides the issuing image is required to be able to observe the effects of a write until some form of image synchronization occurs. Therefore, an implementation is permitted to issue non-blocking writes for all writes provided that it can ensure that conflicting accesses issued by the same image occur in program order. Whether this guarantee is provided by software or hardware depends on the implementation. To explicitly issue and manage completion of a non-blocking write, see *Cofutures*.

Reads (Gets)

A Coarray C++ read access is blocking in order to provide a value for use in an arbitrary expression context:

```
coarray<int> x;
...
int y = x(i) + 1; // read of x(i) shall block
```

A non-blocking read is performed via an explicit `get()` member function of coref:

```
  int y;x(i).get( &y );
... // some code that does not access y
atomic_image_fence();
++y;
```

The `get()` member function issues a non-blocking read that is not guaranteed to complete until the next fence. The `atomic_image_fence()` ensures completion of all previously issued memory accesses. The `get()` plus fence solution is appropriate in many cases, but it may be too broad if the fence would force completion of other accesses on which the issuing image does not yet need to wait. To explicitly issue and manage completion of a non-blocking read, see *Cofutures*.

Cofutures

Coarray C++ provides explicit completion management of a non-blocking access via a cofuture, which is modeled on C++11's `std::future`. A coref plays a similar role to C++11's `std::promise`, providing member functions that create a cofuture. Here is an example of a non-blocking read where the storage for the value is contained within the cofuture. The value cannot be accidentally used before the operation has completed, but existing storage cannot be used as the target of the read:

```
coarray<int> x;
...
cofuture<int> f = x(i).get_cofuture(); // or just x(i)
...
int z = f + 1; // using f waits then implicitly returns the value
```

For convenience, a coref can automatically convert to a `cofuture` so that the `get_cofuture()` call can be omitted. Here is an example of a non-blocking read where the storage for the value is external to the `cofuture`. Care must be taken to not access the storage until `wait()` has been called:

```
int y[100];
...
cofuture<void> f = x(i).get_cofuture( y );
...      // code that does not read or write y
f.wait();
...      // code that reads or writes y
```

Note that the `cofuture`'s parameter type is void because it does not store any value.

Here is an example of a non-blocking write. Care must be taken to not overwrite the source of the write until `wait()` has been called.

```
coarray<int> x;
int y;
...
cofuture<void> f = x(i).put_cofuture( y );
...       // code that does not write y
f.wait(); // ensure that the x(i) = y assignment completed
```

Note that the `cofuture`'s parameter type is void because a cofuture for a write never stores a value.

Code Patterns

Coobjects

When a coarray is included as a member of a class, it can be allocated with the class object or it can be allocated later:

```
// An X must be allocated and destroyed
// collectively because it contains a coarray.
class X {
    coarray<int> x;
    ...
};

// But a Y defers its "collectiveness" until
// it needs to allocate the coarray.
class Y {
```

```
    coarray<int>* y;
    ...
};
```

These two options provide flexibility for implementing collective objects, or coobjects, which can encapsulate coarray data movement.

Hoisting a coptr

When a coarray of pointer type is accessed within a loop, there may be unnecessary reads of the pointer from the target image if the same image is accessed repeatedly:

```
coarray<int*> x;
...
for ( int i = 0; i < n; ++i ) {
    int y = x(1)[i]; // reads pointer x(1) each time
    ...
}
```

A `coptr` or `const_coptr` can be used to hoist the read of the pointer:

```
coarray<int*> x;
...
const_coptr<int> p = x(1)[0].address(); // reads pointer x(1) once
for ( int i = 0; i < n; ++i ) {
int y = p[i];
...
}
```

Cray C Extension Use

Complex Data Extensions

Cray C extends the complex data facilities defined by standard C with these extensions:

- Imaginary constants
- Incrementing or decrementing `_Complex` data

The Cray C compiler supports the Cray imaginary constant extension and is defined in the `<complex.h>` header file. This imaginary constant has the following form:

```
Ri
```

R is either a floating constant or an integer constant; no space or other character can appear between R and i. If compiling in strict conformance mode (`-h conform`), the Cray imaginary constants are not available.

The following example illustrates imaginary constants:

```
#include <complex.h>
double complex z1 = 1.2 + 3.4i;
double complex z2 = 5i;
```

The other extension to the complex data facility allows the prefix– and postfixincrement and decrement operators to be applied to the `_Complex` data type. The operations affect only the real portion of a complex number.

`fortran` Keyword

In extended mode, the identifier `fortran` is treated as a keyword. It specifies a storage class that can be used to declare a Fortran-coded external function. The use of the `fortran` keyword when declaring a function causes the compiler to verify that the arguments used in each call to the function are pass by addresses; any arguments that are not addresses are converted to addresses.

As in any function declaration, an optional type-specifier declares the type returned, if any. Type `int` is the default; type `void` can be used if no value is returned (by a Fortran subroutine). The `fortran` storage class causes conversion of lowercase function names to uppercase, and, if the function name ends with an underscore character, the trailing underscore character is stripped from the function name. (Stripping the trailing underscore character is in keeping with UNIX practice.)

Functions specified with a `fortran` storage class must not be declared elsewhere in the file with a `static` storage class.

The `fortran` keyword is not allowed in Cray C++.

An example using the `fortran` keyword is shown in *Interlanguage Communication*.

Hexadecimal Floating-point Constants

The Cray C compiler supports the standard hexadecimal floating constant notations and the Cray hexadecimal floating constant notation. The standard hexadecimal floating constants are portable and have sizes that are dependent upon the hardware. The remainder of this section discusses the Cray hexadecimal floating constant.

The Cray hexadecimal floating constant feature is not portable, because identical hexadecimal floating constants can have different meanings on different systems. It can be used whenever traditional floating-point constants are allowed.

The hexadecimal constant has the usual syntax: `0x` (or `0X`) followed by hexadecimal characters. The optional floating suffix has the same form as for normal floating constants: `f` or `F` (for float), `l` or `L` (for long), optionally followed by an `i` (imaginary).

The constant must represent the same number of bits as its type, which is determined by the suffix (or the default of double). The constant's bit length is four times the number of hexadecimal digits, including leading zeros.

The following example illustrates hexadecimal constant representation:

```
0x7f7fffff.f
```

32-bit float

```
0x0123456789012345.
```

64-bit double

The value of a hexadecimal floating constant is interpreted as a value in the specified floating type. This uses an unsigned integral type of the same size as the floating type, regardless of whether an object can be explicitly declared with such a type. No conversion or range checking is performed. The resulting floating value is defined in the same way as the result of accessing a member of floating type in a union after a value has been stored in a different member of integral type.

The following example illustrates hexadecimal floating-point constant representation that use Cray floating-point format:

```
int main(void)
{
  float f1, f2;
  double g1, g2;

  f1 = 0x3ec00000.f;
  f2 = 0x3fc00000.f;
  g1 = 0x40fa400100000000.;
  g2 = 0x40fa400200000000.;

  printf("f1 = %8.8g\n", f1);
  printf("f2 = %8.8g\n", f2);
  printf("g1 = %16.16g\n", g1);
  printf("g2 = %16.16g\n", g2);
  return 1;
}
```

This is the output for the previous example:

```
f1 = 0.375
f2 = 1.5
g1 = 107520.0625
g2 = 107520.125
```

Predefined Macro Use

The macros listed in this chapter are the Cray-specific predefined macros. To see the entire list of predefined macros, add `-Wp,-list_final_macros` to the `cc` command line. For example, if it is the file `c.c`, specify:

```
% cc -Wp,-list_final_macros c.c > out
```

Predefined macros provide information about the compilation environment. In this chapter, only those macros that begin with the underscore (_) character are defined when running in strict-conformance mode.

Any of the predefined macros except those required by the standard can be undefined by using the `-U` command line option; they can also be redefined by using the `-D` command line option.

A large set of macros is also defined in the standard header files.

Macros Required by the C and C++ Standards

The following macros are required by C and C++ standards:

__TIME__	Time of translation of the source file.
__DATE__	Date of translation of the source file.
__LINE__	Line number of the current line in the source file.
__FILE__	Name of the source file being compiled.
__STDC__	Defined as the decimal constant 1 if compilation is in strict conformance mode; defined as the decimal constant 2 if the compilation is in extended mode. This macro is defined for Cray C and C ++ compilations.
__cplusplus	Defined as `1` when the compiling Cray C++ code and undefined when compiling Cray C code. The `__cplusplus` macro is required by the ISO C++ standard, but not the ISO C standard.

Macros Based on the Host Machine

The following macros provide information about the environment running on the host machine:

__linux	Defined as 1.
__linux__	Defined as 1.
linux	Defined as 1.
__gnu_linux__	Defined as 1.

Macros Based on the Target Machine

The following macros provide information about the characteristics of the target machine:

Macro	x86	AVX
_ADDR64	Defined as 1 if the targeted CPU has 64-bit address registers; if the targeted CPU does not have 64-bit address registers, the macro is not defined.	
__LITTLE_ENDIAN__	Defined as 1.	
_LITTLE_ENDIAN	Defined as 1.	
_MAXVL_8	Defined as 16, the number of 8-bit elements that fit in an XMM register ("vector length").	Defined as 32, the number of 8-bit elements that fit in a YMM register ("vector length").
_MAXVL_16	Defined as 8.	Defined as 16.
_MAXVL_32	Defined as 4.	Defined as 8.
_MAXVL_64	Defined as 2.	Defined as 4.
_MAXVL_128	Defined as 0.	Defined as 2.

Macros Based on the Compiler

The following macros provide information about compiler features:

_RELEASE_MAJOR	Defined as the major release level of the compiler.
_RELEASE_MINOR	Defined as the minor release level of the compiler.
_RELEASE_STRING	Defined as a string that describes the version of the compiler.
_CRAYC	Defined as 1 to identify the Cray C and C++ compilers.

UPC Predefined Macros

The following macros provide information about UPC functions:

__UPC__	The integer constant 1, indicating a conforming implementation.
__UPC_DYNAMIC_THREADS__	The integer constant 1 in the dynamic THREADS translation environment.
__UPC_STATIC_THREADS__	The integer constant 1 in the static THREADS translation environment.

Run C and C++ Applications

To run applications, log in to a login node and set up the user environment. See the *Cray Application Developer's Environment User's Guide* for details on setting up the environment. In the working directory, load the appropriate modules, compile the programs, and launch them using the `aprun` command.

To use the Cray C compiler, load the `PrgEnv-cray` module. Use the `module list` command to get a list of currently loaded modules. If another Programming Environment module is loaded, use the `module swap` command. For example, if `PrgEnv-pgi` is loaded, use this command:

```
% module swap PrgEnv-pgi PrgEnv-cray
```

Then use the `cc -V` command to verify that the Cray C compiler is available.

Compile the application.

```
% cc -o simple simple.c
```

Move the application to a mount point on the Cray system to execute.

```
% aprun -n 4 ./simple | sort
Application 1024906 resources: utime 0, stime 0
hello from pe 0 of 4
hello from pe 1 of 4
hello from pe 2 of 4
hello from pe 3 of 4
```

If the `-X` option is specified on the `cc` command line, then the `aprun -n` option must specify the same number of processing elements (`npes`). Otherwise, a run time error will be received.

For additional information, see the *Cray Programming Environment User's Guide*.

Debug Cray C and C++ Code

The TotalView symbolic debugger is available to help debug C and C++ codes. In addition, the Cray C and C++ compilers provide the following features to help in debugging codes:

- The `-G` and `-g` compiler options provide symbol information about the source code for use by the TotalView debugger. For more information about these compiler options, see *-G debug_lvl, -g*.
- The `-h [no]bounds` option and the `#pragma _CRI [no]bounds` directive allows for checking pointer and array references. The `-h [no]bounds` option is described in *-h [no]bounds (cc)*. The `#pragma _CRI [no]bounds` directive is described in *#pragma _CRI [no]bounds*.
- The `-G3` option optimizes code for use with Cray fast-track debugging and requires use of a debugger that supports fast-track debugging. For more information, see the `lgdb(1)` man page.
- The `#pragma _CRI message` directive allows for adding warning messages to sections of code where problems are suspected. The `#pragma _CRI message` directive is described in *message*.
- The `#pragma _CRI [no]opt` directive allows for selectively isolating portions of the code to optimize, or to toggle optimization on and off in selected portions of the code. The `#pragma _CRI [no]opt` directive is described in *[no]opt*.

TotalView Debugger

Some of the functions available in the TotalView debugger allow for the performance of the following actions:

- Set and clear breakpoints, which can be conditional, at both the source code level and the assembly code level
- Examine core files
- Step through a program, including across function calls
- Reattach to the executable file after editing and recompiling
- Edit values of variables and memory locations
- Evaluate code fragments

Compiler Debug Options

Compiler options control the trade-offs between ease of debugging and compiler optimizations. The compiler produces internal debugger information (DWARF) at all times. The DWARF data provides function and line information to debuggers for tracebacks and breakpoints, as well as type and location information about data variables.

These options are specified as follows:

`-G3` This option permits both full code optimization and the greatest flexibility in setting breakpoints, but requires use of the Cray fast-track debugger. For more information, see the `lgdb(1)` man page.

-G2 With no DWARF, the executable is optimized and as small as possible, but cannot be easily debugged. Only assembly instructions will be visible and only global symbols will be available.

-G1 With partial DWARF and at least some optimization, tracebacks and limited breakpoints are available in the debugger. The source code will be visible and many more symbols will be available. The executable will be somewhat slower and larger in exchange for increased debugger functionality.

-g or -G0 With full DWARF and no optimizations, full debugging will be available, but at the cost of a slower and larger executable.

The `-g` or `-G` options may be specified on a per file basis so that only part of an application incurs the overhead of improved debugging.

However, consider the following cases in which optimization is affected by the `-G1` and `-G2` debugging options:

- Vectorization can be inhibited if a label exists within the vectorizable loop.
- Vectorization can be inhibited if the loop contains a nested block and the `-G1` option is specified.
- When the `-G1` option is specified, setting a breakpoint at the first statement in a vectorized loop allows for stop and display at each vector iteration. However, setting a breakpoint at the first statement in an unrolled loop may not allow a stop at each vector iteration.

Interlanguage Communication

The C and C++ compilers provide mechanisms for declaring external functions written in other languages. This enables the writing of portions of an application in C, C++, Fortran, or assembly language, which can be useful in cases where the other languages provide performance advantages or utilities not available in C or C++.

Calls Between C and C++ Functions

The following requirements apply when making calls between functions written in C and C++:

- In Cray C++, the extern "C" linkage is required when declaring an external function that is written in Cray C or when declaring a Cray C++ function that is to be called from Cray C. Normally the compiler mangles function names to encode information about the function's prototype in the external name; this prevents direct access to these function names from a C function. The extern "C" keyword prevents the compiler from performing name mangling.
- The program must be linked using the CC command.
- The program's main routine must be C or C++ code compiled using the CC command.

Objects can be shared between C and C++. There are some Cray C++ objects that are not accessible to Cray C functions, such as classes. The following object types can be shared directly:

- Integral and floating types.
- Structures and unions that are declared identically in C and C++. In order for structures and unions to be shared, they must be declared with identical members in the identical order.
- Arrays and pointers to the above types.

C_add_func is called by the Cray C++ main program:

```
#include <iostream.h>

extern "C" int C_add_func(int, int);
int global_int = 123;

main()
{
  int res, i;

  cout << "Start C++ main" << endl;

  /* Call C function to add two integers and return result. */

 cout << "Call C C_add_func" << endl;
 res = C_add_func(10, 20);
 cout << "Result of C_add_func = " << res << endl;
 cout << "End C++ main << endl;
}
```

C_add_func

```
#include <stdio.h>

extern int global_int;

int C_add_func(int p1, int p2)
{
    printf("\tStart C function C_add_func.\n");
    printf("\t\tp1 = %d\n", p1);
    printf("\t\tp2 = %d\n", p2);
    printf("\t\tglobal_int = %d\n", global_int);
    return p1 + p2;
}
```

Output

```
Start C++ main
Call C C_add_func
        Start C function C_add_func.
                p1 = 10
                p2 = 20
                global_int = 123
Result of C_add_func = 30
End C++ main
```

Call Fortran Functions and Subroutines from C or C++

The following conditions are required to call Fortran Functions and Subroutines from C or C++:

- Fortran uses the call-by-address convention. C and C++ use the call-by-value convention, which means that only pointers should be passed to Fortran subprograms.
- Fortran arrays are in column-major order. C and C++ arrays are in row-major order. This indicates which dimension is indicated by the first value in an array element subscript.
- Single-dimension arrays of signed 32-bit integers and single-dimension arrays of 32-bit floating-point numbers are the only aggregates that can be passed as parameters without changing the arrays.
- Fortran character pointers and character pointers from Cray C and C++ are incompatible.
- Fortran logical values and the Boolean values from C and C++ are not fully compatible.
- External C and C++ variables are stored in common blocks of the same name, making them readily accessible from Fortran programs if the C or C++ variable is in uppercase.
- When declaring Fortran functions or objects in C or C++, the name must be specified in all uppercase letters, digits, or underscore characters and consist of 31 or fewer characters.
- In Cray C, Fortran functions can be declared using the fortran keyword. The fortran keyword is not available in Cray C++. Instead, Fortran functions must be declared by specifying extern "C".

Argument Passing

Because Fortran subroutines expect arguments to be passed by pointers rather than by value, C and C++ functions called from Fortran subroutines must pass pointers rather than values.

All argument passing in Cray C is strictly by value. To prepare for a function call between two Cray C functions, a copy is made of each actual argument. A function can change the values of its formal parameters, but these

changes cannot affect the values of the actual arguments. It is possible, however, to pass a pointer. (All array arguments are passed by this method.) This capability is analogous to the Fortran method of passing arguments.

In addition to passing by value, Cray C++ also provides passing by reference.

Array Storage

C and C++ arrays are stored in memory in row-major order. Fortran arrays are stored in memory in column-major order. For example, the C or C++ array declaration int **A[3][2]** is stored in memory as:

A[0][0]	**A[0][1]**
A[1][0]	**A[1][1]**
A[2][0]	**A[2][1]**

The previously defined array is viewed linearly in memory as:

```
A[0][0] A[0][1] A[1][0] A[1][1] A[2][0] A[2][1]
```

The Fortran array declaration **INTEGER A(3,2)** is stored in memory as:

A(1,1)	**A(2,1)**	**A(3,1)**
A(1,2)	**A(2,2)**	**A(3,2)**

The previously defined array is viewed linearly in memory as:

```
A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2)
```

When an array is shared between Cray C, C++, and Fortran, its dimensions are declared and referenced in C and C++ in the opposite order in which they are declared and referenced in Fortran. Arrays are zero-based in C and C ++ and are one-based in Fortran, so in C and C++, subtract 1 from the array subscripts that would normally be used in Fortran.

For example, using the Fortran declaration of array A in the preceding example, the equivalent declaration in C or C++ is:

```
int a[2][3];
```

The following list shows how to access elements of the array from Fortran and from C or C++:

Fortran	C or C++
A(1,1)	**A[0][0]**
A(2,1)	**A[0][1]**
A(3,1)	**A[0][2]**
A(1,2)	**A[1][0]**
A(2,2)	**A[1][1]**
A(3,2)	**A[1][2]**

Logical and Character Data

Logical and character data need special treatment for calls between C or C++ and Fortran. Fortran has a character descriptor that is incompatible with a character pointer in C and C++. The techniques used to represent logical (Boolean) values also differ between Cray C, C++, and Fortran.

Mechanisms used to convert one type to the other are provided by the `fortran.h` header file and conversion macros shown in the following list:

Macro	Description
`_btol`	Conversion utility that converts a 0 to a Fortran logical **`.FALSE.`** and a nonzero value to a Fortran logical **`.TRUE.`**
`_ltob`	Conversion utility that converts a Fortran logical **`.FALSE.`** to a 0 and a Fortran logical **`.TRUE.`** to a 1.

Access Named Common from C and C++

The following example demonstrates how external C and C++ variables are accessible in Fortran named common blocks. It shows a C or C++ function calling a Fortran subprogram, the associated Fortran subprogram, and the associated input and output.

In this example, the C or C++ structure **`_st`** is accessed in the Fortran subprogram as common block **`ST`**. The Fortran common block ST will be converted to lower case with a trailing underscore added.

The name of the structure and the converted Fortran common block name must match. The C and C++ structure member names and the Fortran common block member names do not have to match, as is shown in this example.

The following Cray C main program calls the Fortran subprogram **`FCTN`**:

```
#include <stdio.h>
struct
{
  int i;
  double a[10];
  long double d;
} _st;

main()
{
   int i;

   /* initialize struct _st */
   _st.I = 12345;

   for (i = 0; i < 10; i++)
      _st.a[i] = i;

   _st.d = 1234567890.1234567890L;

   /* print out the members of struct _st */
   printf("In C: _st.i = %d, _st.d = %20.10Lf\n", _st.i, _st.d);
   printf("In C: _st.a = ");
   for (i = 0; i < 10; i++)
```

```
      printf("%4.1f", _st.a[i]);
   printf("\n\n");

   /* call the fortran function */
   FCTN();
}
```

The following example is the Fortran subprogram **FCTN** called by the previous Cray C main program:

```
C *********** Fortran subprogram (f.f): ***********

    SUBROUTINE FCTN

    COMMON /ST/STI, STA(10), STD
    INTEGER STI
    REAL STA
    DOUBLE PRECISION STD

INTEGER I

    WRITE(6,100) STI, STD
100 FORMAT ('IN FORTRAN: STI = ', I5, ', STD = ', D25.20)
    WRITE(6,200) (STA(I), I = 1,10)
200 FORMAT ('IN FORTRAN: STA =', 10F4.1)
    END
```

The previous Cray C and Fortran examples are executed by the following commands, and they produce the output shown:

```
% cc -c c.c
% ftn -c f.f
% ftn c.o f.o
% ./a.out
 ST.i = 12345, ST.d = 1234567890.1234567890
 In C: ST.a = 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

 IN FORTRAN: STI = 12345, STD = .12345678901234567889D+10
 IN FORTRAN: STA = 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

Access Blank Common from C or C++

Fortran includes the concept of a common block. A common block is an area of memory that can be referenced by any program unit in a program. A named common block has a name specified in names of variables or arrays stored in the block. A blank common block, sometimes referred to as *blank common*, is declared in the same way, but without a name.There is no way to access blank common from C or C++ similar to accessing a named common block. However, a simple Fortran function can be written to return the address of the first word in blank common to the C or C++ program and then use that as a pointer value to access blank common.

Access Fortran blank common from C or C++ Code

```
#include <stdio.h>

struct st
{
  float a;
  float b[10];
} *ST;
```

```
#ifdef __cplusplus
  extern "C" struct st *MYCOMMON(void);
  extern "C" void FCTN(void);
#else
  fortran struct st *MYCOMMON(void);
  fortran void FCTN(void);
#endif

main()
{
    int i;

    ST = MYCOMMON();
    ST->a = 1.0;
    for (i = 0; i < 10; i++)
        ST->b[i] = i+2;
    printf("\n In C and C++\n");
    printf("     a = %5.1f\n", ST->a);
    printf("     b = ");
    for (i = 0; i < 10; i++)
        printf("%5.1f ", ST->b[i]);
    printf("\n\n");

    FCTN();
}
```

The following Fortran subroutine uses blank common and is called from `main()` above.

```
SUBROUTINE FCTN
COMMON // STA,STB(10)
PRINT *, "IN FORTRAN"
PRINT *, "    STA = ",STA
PRINT *, "    STB = ",STB
STOP
END

INTEGER(KIND=8) FUNCTION MYCOMMON()
COMMON // A
MYCOMMON = LOC(A)
RETURN
END
```

The previous Cray C and Fortran code is executed by the following commands, and produces the output shown:

```
% cc -c c1.c
% ftn -c f2.f
% ftn c1.o f1.o
% ./a.out
 In C and C++
     a =   1.0
     b =   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0  11.0

 IN FORTRAN
```

```
     STA =  1.
     STB =  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  10.,  11.
 STOP
```

Here is an example of a Cray C function that calls a Fortran subprogram. The Fortran subprogram example follows the Cray C function example, and the input and output from this sequence follows the Fortran subprogram example.

This example assumes that the Cray Fortran function is compiled with the `-s default32` option enabled. The examples will not work if the `-s default64` option is enabled.

```
/*                C program (main.c):                    */

#include <stdio.h>
#include <string.h>
#include <fortran.h>

/* Declare prototype of the Fortran function. Note the last */
/* argument passes the length of the first argument. */
fortran double FTNFCTN (char *, int *, int);

double FLOAT1 = 1.6;
double FLOAT2; /* Initialized in FTNFCTN */

main()
{
     int clogical, ftnlogical, cstringlen;
     double rtnval;
     char *cstring = "C Character String";

/* Convert clogical to its Fortran equivalent */
     clogical = 1;
     ftnlogical = _btol(clogical);

/* Print values of variables before call to Fortran function */
     printf(" In main: FLOAT1 = %g; FLOAT2 = %g\n",
          FLOAT1, FLOAT2);
     printf(" Calling FTNFCTN with arguments:\n");
     printf(" string = \"%s\"; logical = %d\n\n", cstring, clogical);
     cstringlen = strlen(cstring);
     rtnval = FTNFCTN(cstring, &ftnlogical, cstringlen);

/* Convert ftnlogical to its C equivalent */
     clogical = _ltob(&ftnlogical);

/* Print values of variables after call to Fortran function */
     printf(" Back in main: FTNFCTN returned %g\n", rtnval);
     printf(" and changed the two arguments:\n");
     printf(" string = \"%.*s\"; logical = %d\n",
     cstringlen, cstring, clogical);
}
C                 Fortran subprogram (ftnfctn.f):

     FUNCTION FTNFCTN(STR, LOG)
     REAL FTNFCTN
     CHARACTER*(*) STR
     LOGICAL LOG
```

```
      COMMON /FLOAT1/FLOAT1
      COMMON /FLOAT2/FLOAT2
      REAL FLOAT1, FLOAT2
      DATA FLOAT2/2.4/ ! FLOAT1 INITIALIZED IN MAIN

C       PRINT CURRENT STATE OF VARIABLES
        PRINT*, ' IN FTNFCTN: FLOAT1 = ', FLOAT1,
       1 ';FLOAT2 = ', FLOAT2
        PRINT*, ' ARGUMENTS: STR = "', STR, '"; LOG = ', LOG

C       CHANGE THE VALUES FOR STR(ING) AND LOG(ICAL)
        STR = 'New Fortran String'
        LOG = .FALSE.

        FTNFCTN = 123.4
        PRINT*, ' RETURNING FROM FTNFCTN WITH ', FTNFCTN
        PRINT*
        RETURN
        END
```

The previous Cray C function and Fortran subprogram are executed by the following commands and produce the following output:

```
% cc -c main.c
% ftn -c ftnfctn.f
% ftn main.o ftnfctn.o
% ./a.out
In main: FLOAT1 = 1.6;  FLOAT2 = 2.4
Calling FTNFCTN with arguments:
string = "C Character String"; logical = 1

IN FTNFCTN: FLOAT1 = 1.6; FLOAT2 = 2.4
ARGUMENTS:   STR = "C Character String"; LOG = T
RETURNING FROM FTNFCTN WITH 123.4
Back in main: FTNFCTN returned 123.4
and changed the two arguments:
string = "New Fortran String"; logical = 0
```

Call a Fortran Program from Cray C++

The following example illustrates how a Fortran program can be called from a Cray C++ program:

```
#include <iostream>
using namespace std;
extern "C" int fortran_add_ints_(int *arg1, int &arg2);

main()
{
     int num1, num2, res;
     cout << "Start C++ main" << endl << endl;
     //Call FORTRAN function to add two integers and return result.
     //Note that the second argument is a reference parameter so
     //it is not necessary to take the address of the
     //variable num2.

     num1 = 10;
     num2 = 20;
     cout << "Before Call to FORTRAN_ADD_INTS" << endl;
```

```
      res = fortran_add_ints_(&num1, num2);
      cout << "Result of FORTRAN Add = " << res << endl << endl;
      cout << "End C++ main" << endl;
}
```

The Fortran program that is called from the Cray C++ main function in the preceding example is as follows:

```
INTEGER FUNCTION FORTRAN_ADD_INTS(Arg1, Arg2)
INTEGER Arg1, Arg2

PRINT *," FORTRAN_ADD_INTS, Arg1,Arg2 = ", Arg1, Arg2
FORTRAN_ADD_INTS = Arg1 + Arg2
END
```

The output from the execution of the preceding example is as follows:

```
Start C++ main

Before Call to FORTRAN_ADD_INTS
  FORTRAN_ADD_INTS, Arg1,Arg2 = 10, 20
Result of FORTRAN Add = 30

End C++ main
```

Calling a C or C++ Function from Fortran

Two methods can be used to call C or C++ functions from Fortran:

- Standard Fortran/C Interoperability
- Portable Interoperability Mechanism

Standard Fortran/C Interoperability

For more information about C interoperability, see the current Fortran standard. The `ISO_C_BINDING` module provides interoperability between Fortran intrinsic types and C types. The `ISO_C_BINDING` module provides named constants which can be used as `KIND` type parameters, compatible with C types. In addition to the named constants required by the Fortran standard, Cray compiler provides, as an extension, definitions for 128-bit floating, and complex types. `C_FLOAT128` and `C_FLOAT128_COMPLEX` correspond to C types `__float128` and `__float128 complex`. For more information about C interoperability, see the current Fortran standard.

Portable Interoperability Mechanism

When calling a Cray C++ function from a Fortran program, observe the following rules:

- The Cray C++ function must be declared with extern "C" linkage.
- The program must be linked using the CC() command.
- The program's main routine must be C or C++ code compiled using the CC command.

Call a C function from Fortran

Fortran program `main.f` source code:

```
C  Fortran program (main.f):
```

```
      PROGRAM MAIN

      REAL CFCTN
      COMMON /FLOAT1/FLOAT1
      COMMON /FLOAT2/FLOAT2
      REAL FLOAT1, FLOAT2
      DATA FLOAT1/1.6/ ! FLOAT2 INITIALIZED IN cfctn.c
      LOGICAL LOG
      CHARACTER*24 STR
      REAL RTNVAL

C INITIALIZE VARIABLES STR(ING) AND LOG(ICAL)
      STR = 'Fortran Character String'
      LOG = .TRUE.

C PRINT VALUES OF VARIABLES BEFORE CALL TO C FUNCTION
      PRINT*, 'In main.f: FLOAT1 = ', FLOAT1,
     1               '; FLOAT2 = ', FLOAT2
      PRINT*, 'Calling cfctn.c with these arguments: '
      PRINT*, 'LOG = ', LOG
      PRINT*, 'STR = ', STR

      RTNVAL = CFCTN(STR, LOG)

C PRINT VALUES OF VARIABLES AFTER CALL TO C FUNCTION
      PRINT*, 'Back in main.f:: cfctn.c returned ', RTNVAL
      PRINT*, 'and changed the two arguments to: '
      PRINT*, 'LOG = ', LOG
      PRINT*, 'STR = ', STR

      END PROGRAM
```

Compile `main.f`, creating `main.o`:

```
% ftn -c main.f
```

C function `cfctn.c` source code:

```
/*   C function (cfctn.c)    */
#include <fortran.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

float FLOAT1;    /* Initialized in MAIN */
float FLOAT2 = 2.4;

/* The slen argument passes the length of string in str */
float cfctn_(char * str, int *log, int slen)
{
  int clog;
  float rtnval;
  char *cstring;

/* Convert log passed from Fortran MAIN */
/* into its C equivalent */
  cstring = malloc(slen+1);
  strncpy(cstring, str, slen);
  cstring[slen] = '\0';
```

```
  clog = _ltob(log);

/* Print the current state of the variables */
  printf(" In CFCTN: FLOAT1 = %.1f; FLOAT2 = %.1f\n",
         FLOAT1, FLOAT2);
  printf(" Arguments: str = '%s'; log = %d\n",
  cstring, clog);

/* Change the values for str and log */
  strncpy(str, "C Character String ", 24);
  *log = 0;

  rtnval = 123.4;
  printf(" Returning from CFCTN with %.1f\n\n", rtnval);
  return(rtnval);
}
```

Compile `cfctn.c`, creating `cfctn.o`:

```
% cc -c cfctn.c
```

Link `main.o` and `cfctn.o`, creating executable **`interlang1`**:

```
% % ftn -o interlang1 main.o cfctn.o
```

Run program **`interlang1`**:

```
% ./interlang1
```

Program output:

```
In main.f: FLOAT1 = 1.60000002 ; FLOAT2 = 2.4000001
 Calling cfctn.c with these arguments:
 LOG = T
 STR = Fortran Character String
 In CFCTN: FLOAT1 = 1.6; FLOAT2 = 2.4
 Arguments: str = 'Fortran Character String'; log = 1
 Returning from CFCTN with 123.4

 Back in main.f:: cfctn.c returned 123.400002
 and changed the two arguments to:
 LOG = F
 STR = C Character String
```

Fortran, C, C++ Interoperability

The Cray Compiler supports interoperability mechanisms specfied in the Fortran 2008 standard, ISO/IEC 1539-1:2010, and *TS 29113 Further Interoperability of Fortran and C*.

The Fortran 2008 standard describes interoperabilty features for:

Intrinsic Types The Fortran intrinsic module `ISO_C_BINDING` provides interoperability between Fortran intrinsic types and C types. The `ISO_C_BINDING` module provides named constants which can be used as `KIND` type parameters, compatible with C types.

In addition to the named constants required by the Fortran standard, Cray compiler provides, as an extension, definitions for 128-bit floating, and complex types. `C_FLOAT128` and `C_FLOAT128_COMPLEX` correspond to C types `__float128` and `__float128 complex`.

Derived Types and Structures

Use the `BIND` attribute when creating an interoperable type:

```
USE ISO_C_BINDING
TYPE, BIND(C) :: THIS_TYPE
    . . .
END TYPE THIS_TYPE
```

Global Variables

Use the `BIND` attribute with a common block declaration, or module variable:

```
USE ISO_C_BINDING
     INTEGER(C_INT), BIND(C) :: EXTERN
     INTEGER(C_LONG) :: CVAR
     BIND(C, NAME='var') :: CVAR
     COMMON /A/ I, J
     REAL(C_FLOAT) :: I, J
     BIND(C) :: /A/
```

Pointers

`ISO_C_BINDING` provides a derived type, `c_ptr`, that interoperates with any C pointer type. Also, Fortran named constant `c_null_ptr` is equivalent to the C value NULL.

Subroutines and Function

Declare a Fortran procedure with the BIND attribute. Procedure arguments must be of interoperable type. By default the Fortran compiler converts the procedure name to lower-case (`myfunction`); this is the *binding label*, or corresponding name which is known to the C compiler.

```
FUNCTION MYFUNCTION(X, Y), BIND(C)
```

Specify a different binding label:

```
FUNCTION MYFUNCTION(X, Y), BIND(C, NAME='C_Myfunction')
```

A function result must be scalar and of interoperable type. A subroutine prototype must have a void result.

TS 29113 describes further interoperability features including:

C descriptors `ISO_Fortran_binding.h` defines C structure `CFI_cdesc_t` which facilitates using Fortran data objects from within a C function.

`ISO_Fortran binding.h` Contains additional C structure definitions and macro definitions to interoperate with an allocatable, or data pointer argument.

Interlanguage Communication Examples

Interlanguage Communication using Common Block/Global

```
//  common_c.c : example of function called from common.f90
```

```
#include <stdio.h>
#include <stdlib.h>
#include <ISO_Fortran_binding.h>

//   globals that match up to the common blocks in common.f90
float c_single;

struct common {
  double var1;
  int var2;
} multiple;

int c_int_array[100];

// c function called from Fortran
void global_var_common()
{
  int i;
  //  just prints and sets the globals


  printf(" In global_var_common\n");
  printf("   c_single: %f\n", c_single);
  printf("   multiple: %f, %d\n", multiple.var1, multiple.var2 );
  printf("   c_int_array: %d, %d\n", c_int_array[0],
c_int_array[99]);

  c_single = 2 * c_single;
  multiple.var1 = 77.77;
  multiple.var2 = 17;
  for(i=0; i<100; i++ ) {
     c_int_array[i] = c_int_array[i] * 3;
  }

}  // end of global_var_common
```

```
! common.f90
! Needs common_c.c

program common_block
  use, intrinsic :: iso_c_binding
!  use check_error
  implicit none
!
!  declare the common blocks for c globals
!  one with a single real variable
  real(c_float) r_var
  common /c_single/ r_var
!  one with an integer array
  integer i_array(100)
  common / array / i_array
!  one with two variables
  real(c_double) :: var1
  integer(c_int) :: var2
  common / multiple / var1, var2
!   do the bind c on the common blocks, renaming one
BIND(C,name="c_int_array") :: / array /
BIND(C) :: / multiple /,  /c_single/
```

```
call sub1()

end program common_block

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine sub1( )
  use, intrinsic :: iso_c_binding

!  declare the common blocks for c globals
!  one with a single real variable
  real(c_float) r_var
  common /c_single/ r_var
!  one with an integer array
  integer i_array(100)
  common / array / i_array
  real(c_double) var1
  integer(c_int) var2
  common / multiple / var1, var2
!   do the bind c on the common blocks, renaming array
BIND(C,name="c_int_array") :: / array /
BIND(C) :: / multiple /,  /c_single/

  interface
     subroutine global_var_common( )  bind(c)
       use,intrinsic :: iso_c_binding
       implicit none
     end subroutine global_var_common
  end interface

r_var = -99.3
var1 = 88.88
var2 = -13
i_array = [(i,i=1,100)]


! call the c function
call global_var_common( )

print *, "In sub1"
print *, "  r_var : ", r_var
print *, "  var1  : ", var1
print *, "  var2  : ", var2
print *, "  array : ", i_array(1), i_array(100)

end subroutine
```

Interlanguage Communication using Derived Structure

```
// c program that calls the Fortran subroutine with struct argument,
f2008 C.11.3
//
*********************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <ISO_Fortran_binding.h>

  //  declare the structure type
```

```
struct pass {
int lenc, lenf;
float *c, *f;
};

  //  prototype for the Fortran function
void simulation(long alpha, double *beta, long *gamma, double
delta[], struct pass *arrays);

//  program that calls the Fortran subroutine
int main ( )
{
  int i;
  long alpha, gamma;
  double beta, delta[100];
  struct pass arrays;

  alpha = 1234L;
  gamma = 5678L;
  beta = 12.34;
  for(i=0; i<100; i++ ) {
     delta[i] = i+1;
  }

  //  fill in some of the structure
  arrays.lenc = 100;
  arrays.lenf = 0;
  arrays.c = (float *) malloc( 100*sizeof(float) );
  arrays.f = NULL;
  for(i=0; i<100; i++ ) {
     arrays.c[i] = 2*(i+1);
  }

  //  reference the Fortran subroutine
  simulation(alpha, &beta, &gamma, delta, &arrays);

  printf(" After simulation\n");
  printf("   alpha: %d, beta: %f\n", alpha, beta );
  printf("   gamma: %d\n", gamma );
  printf("   arrays.lenc: %d\n", arrays.lenc);
  printf("   arrays.c[0],[arrays.lenc-1],: %f, %f\n", arrays.c[0],
arrays.c[arrays.lenc-1]);
  printf("   arrays.lenf: %d\n", arrays.lenf);
  printf("   arrays.f[0],[arrays.lenf-1],: %f, %f\n", arrays.f[0],
arrays.f[arrays.lenf-1]);

}  //  end of main
```

```
! Example derived type/structure interoperability, f2008 C.11.3
!*************************************************************
subroutine simulation(alpha, beta, gamma, delta, arrays) bind(c)
  use, intrinsic :: iso_c_binding
  implicit none
  integer (c_long), value :: alpha
  real (c_double), intent(inout) :: beta
  integer (c_long), intent(out) :: gamma
  real (c_double),dimension(*),intent(in) :: delta
  type, bind(c) :: pass
    integer (c_int) :: lenc, lenf
    type (c_ptr) :: c, f
```

```
  end type pass
  type (pass), intent(inout) :: arrays
  real (c_float), allocatable, target, save :: eta(:)
  real (c_float), pointer :: c_array(:)
  integer i

  print *, "In simulation"
  print *, "  alpha: ", alpha, ", beta: ", beta
  print *, "  delta(1),(100): ", delta(1), delta(100)

  ! associate c_array with an array allocated in c
  call c_f_pointer (arrays%c, c_array, [arrays%lenc])
  print *, "  c_array(1),(arrays%lenc): ", c_array(1), c_array(arrays
%lenc)

  ! allocate an array and make it available in c
  arrays%lenf = 100
  allocate (eta(arrays%lenf))
  arrays%f = c_loc(eta)
  eta = [(i*3,i=1,arrays%lenf)]

  ! change argument values
  c_array = c_array * 2.0
  gamma = 77
  beta = -55.66

end subroutine simulation
```

Interlanguage Communication using Module

```
//  c function called from module.f90

#include <stdio.h>
#include <stdlib.h>
#include <ISO_Fortran_binding.h>

//   globals that match up to the module variables in module.f90
float r_var;
double var1;
int var2;
int c_int_array[100];

//  c function called from Fortran
void global_var_module()
{
  int i;
  //  just prints and sets the globals


  printf(" In global_var_module\n");
  printf("   r_var       : %f\n", r_var);
  printf("   var1        : %f\n", var1 );
  printf("   var2        : %d\n", var2 );
  printf("   c_int_array: %d, %d\n", c_int_array[0],
c_int_array[99]);

  r_var = 2 * r_var;
  var1 = 77.77;
```

```
  var2 = 17;
  for(i=0; i<100; i++ ) {
     c_int_array[i] = c_int_array[i] * 3;
  }

}  // end of global_var_module
```

```
! Example of module/global variable interoperability.
! Needs c function from module_c.c
!
**********************************************************************
module module_example_mod
  use, intrinsic :: iso_c_binding
  real(c_float) r_var
  integer i_array(100)
  real(c_double) :: var1
  integer(c_int) :: var2
BIND(C,name="c_int_array") :: i_array
BIND(C) :: r_var, var1, var2
end module module_example_mod

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program module_example
  use module_example_mod
  implicit none

call sub1()

end program module_example

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine sub1( )
  use module_example_mod

  interface  ! for the c function
     subroutine global_var_module( )  bind(c)
       use,intrinsic :: iso_c_binding
       implicit none
     end subroutine global_var_module
  end interface

r_var = -99.3
var1 = 88.88
var2 = -13
i_array = [(i,i=1,100)]


! call the c function
call global_var_module( )

print *, "In sub1"
print *, "  r_var : ", r_var
print *, "  var1  : ", var1
print *, "  var2  : ", var2
print *, "  array : ", i_array(1), i_array(100)

end subroutine
```

Implementation-defined Behavior

The C and/or C++ standards define certain compiler behavior that is implementation defined. The standards require that the behavior of each particular implementation be documented, and define implementation-defined behavior as behavior that is dependent on the characteristics of the implementation for a correct program construct and correct data.

Messages

All diagnostic messages issued by the compilers are reported through the Cray Linux Environment (CLE) message system. For information about messages issued by the compilers and for information about the Cray Linux Environment (CLE) message system, see *Compiler Message System Use*.

Environment

When `argc` and `argv` are used as parameters to the main function, the array members `argv[0]` through `argv[argc-1]` contain pointers to strings that are set by the command shell. The shell sets these arguments to the list of words on the command line used to invoke the compiler (the argument list). For further information about how the words in the argument list are formed, refer to the documentation on the shell being run.

A third parameter, `char **envp`, provides access to environment variables. The value of the parameter is a pointer to the first element of an array of null-terminated strings that matches the output of the env command. The array of pointers is terminated by a null pointer.

The compiler does not distinguish between interactive devices and other, noninteractive devices. The library, however, may determine that `stdin`, `stdout`, and `stderr` (`cin`, `cout`, and `cerr` in Cray C++) refer to interactive devices and buffer them accordingly.

Identifiers

The *identifier* (as defined by the standards) is merely a sequence of letters and digits. Specific uses of identifiers are called *names*.

The Cray C compiler treats the first 255 characters of a name as significant, regardless of whether it is an internal or external name. The case of names, including external names, is significant. In Cray C++, all characters of a name are significant.

Types

The table below summarizes Cray C and C++ types and the characteristics of each type. Representation is the number of bits used to represent an object of that type. Memory is the number of storage bits that an object of that type occupies.

In the Cray C and C++ compilers, size, in the context of the sizeof operator, refers to the size allocated to store the operand in memory; it does not refer to representation, as specified in the table. Thus, the sizeof operator will

return a size that is equal to the value in the *Memory* column of the table divided by 8 (the number of bits in a byte).

Table 4. Data Type Mapping

Type	Representation Size and Memory Storage Size (bits)
`bool` (C++)	8
`_Bool` (C)	8
`char`	8
`wchar_t`	32
`short`	16
`int`	32
`long`	64
`long long`	64
`float`	32
`double`	64
`long double`	64
`float complex`	64 (each part is 32 bits)
`double complex`	128 (each part is 64 bits)
`long double complex`	128 (each part is 64 bits)
`_float128`	128
`_float128 complex`	256 (each part is 128 bits)
`Pointers`	64

Vectorization of 8- and 16-bit data types is deferred.

Characters

The full 8-bit ASCII code set can be used in source files. Characters not in the character set defined in the standard are permitted only within character constants, string literals, and comments. The `-h [no]calchars` option allows the use of the `$` sign in identifier names. For more information about the `-h [no]calchars` option, see *-h [no]calchars*.

A character consists of 8 bits. Up to 8 characters can be packed into a 64-bit word. A plain `char` type (that is, one that is declared without a `signed` or unsigned keyword) is treated as a signed type.

Character constants and string literals can contain any characters defined in the 8-bit ASCII code set. The characters are represented in their full 8-bit form. A character constant can contain up to 8 characters. The integer value of a character constant is the value of the characters packed into a word from left to right, with the result right-justified, as shown in the following table:

Table 5. Packed Characters

Character Count	Integer Value
`'a'`	`0x61`
`'ab'`	`0x6162`

In a character constant or string literal, if an escape sequence is not recognized, the \ character that initiates the escape sequence is ignored, as shown in the following table:

Table 6. Unrecognizable Escape Sequences

Character Constant	Integer Value	Explanation
`'\a'`	`0x7`	Recognized as the ASCII BEL character
`'\8'`	`0x38`	Not recognized; ASCII value for 8
`'\['`	`0x5b`	Not recognized; ASCII value for `[`
`'\c'`	`0x63`	Not recognized; ASCII value for `c`

Wide Characters

Wide characters are treated as signed 64-bit integer types. Wide character constants cannot contain more than one multibyte character. Multibyte characters in wide character constants and wide string literals are converted to wide characters in the compiler by calling the `mbtowc()` function. The current locale in effect at the time of compilation determines the method by which `mbtowc()` converts multibyte characters to wide characters, and the shift states required for the encoding of multibyte characters in the source code. If a wide character, as converted from a multibyte character or as specified by an escape sequence, cannot be represented in the extended execution character set, it is truncated.

Integers

All integral values are represented in a two's complement format. When an integer is converted to a shorter signed integer, and the value cannot be represented, the result is the truncated representation treated as a signed quantity. When an unsigned integer is converted to a signed integer of equal length, and the value cannot be represented, the result is the original representation treated as a signed quantity.

The bitwise operators (unary operator ~ and binary operators <<, >>, &, ^, and |) operate on signed integers in the same manner in which they operate on unsigned integers. The result of *e*1 >> *e*2, where *e*1 is a negative-valued signed integral value, is that *e*1 is right-shifted *e*2 bit positions; vacated bits are filled with 1s. This behavior can be modified by using the `-h nosignedshifts` option (see *General Language Options*). Bits higher than the sixth bit are not ignored.

The result of the / operator is the largest integer less than or equal to the algebraic quotient when either operand is negative and the result is a nonnegative value. If the result is a negative value, it is the smallest integer greater than or equal to the algebraic quotient. The / operator behaves the same way in C and C++ as in Fortran.

The sign of the result of the percent (%) operator is the sign of the first operand.

Integer overflow is ignored. Because some integer arithmetic uses the floating-point instructions, floating-point overflow can occur during integer operations. Division by 0 and all floating-point exceptions, if not detected as an error by the compiler, can cause a run time abort.

128-Bit Floating Point and 256-Bit Complex Predefined Types

The Cray C and C++ Compilers now support 128-bit floating point and 256-bit complex predefined types using the X86-64 ABI definitions for type names and data layout. These types are sometimes referred to as *quad-precision*. In C and C++, use `__float128`, and `__float128 _complex`. The header file `quadmath.h` defines the 128-bit functions and the header file `complex.h` defines the complex functions.

The base type itself uses 128 bits of storage with a guaranteed minimum alignment on a 128-bit boundary, little endian, has a 15-bit exponent, a 113-bit mantissa, and an exponent bias of 16383, and is compatible with the `gcc` implementation.

In C and C++, `long double` remains identical to `double` – 64-bit IEEE, and not 80-bit extended precision.

C forms of intrinsic math functions offer full support for quad-precision types. See the `intro_quad_precision(3i)` man page for a complete list of intrinsic functions that support quad-precision.

There is no printf descriptor for `__float128`. `quadmath.h` defines gnu functions `quadmath_snprintf()` and `strtoflt128()` that `convert __float128` to strings:

```
extern __float128 strtoflt128 (const char *, char **);
extern int quadmath_snprintf (char *str, size_t size, const char *format, ...)
```

Alternatively, use a Fortran subroutine to print a 128-bit floating point number:

```
% cat printf128main.c printf128.f
#include <quadmath.h>
#include <math.h>
printf128_(__float128 *x);
void main(){
      __float128 x=1.2345678901234567890123456789012345678900Q;
      printf128_(&x);
}

      subroutine printf128(qx)
      real*16 qx
      print "('qx:', f5.1, e45.35)",qx,qx
      end

%cc printf128main.c printf128.o && ./a.out
qx: 1.2 0.12345678901234567890123456789012346E+01
```

Arrays and Pointers

An `unsigned long` value can hold the maximum size of an array. The type `size_t` is defined to be a `typedef` name for unsigned long in the headers: `malloc.h`, `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, and `time.h`. If more than one of these headers is included, only the first defines `size_t`.

A type `long` can hold the difference between two pointers to elements of the same array. The type `ptrdiff_t` is defined to be a `typedef` name for long in the header `stddef.h`.

If a pointer type's value is cast to a signed or unsigned `long int`, and then cast back to the original type's value, the two pointer values will compare equal. Type-casting from pointer to `long int` enables pointer arithmetic. For example:

```
static void **Table;
size_t offset = -BlockSize[nr];
Table = (void **) malloc(MAXBLOCKS * sizeof(void *));
Table[i] = (void **) (((long) Table[i]) + offset);
```

Pointers on Cray Linux Environment (CLE) systems are byte pointers. Byte pointers use the same internal representation as integers; a byte pointer counts the numbers of bytes from the first address.

A pointer can be explicitly converted to any integral type large enough to hold it. The result will have the same bit pattern as the original pointer. Similarly, any value of integral type can be explicitly converted to a pointer. The resulting pointer will have the same bit pattern as the original integral type.

Registers

Use of the register storage class in the declaration of an object has no effect on whether the object is placed in a register. The compiler performs register assignment aggressively; that is, it automatically attempts to place as many variables as possible into registers.

Classes, Structures, Unions, Enumerations, and Bit Fields

Accessing a member of a union by using a member of a different type results in an attempt to interpret, without conversion, the representation of the value of the member as the representation of a value in the different type.

Members of a class or structure are packed into words from left to right. Padding is appended to a member to correctly align the following member, if necessary. Member alignment is based on the size of the member:

- For a member bit field of any size, alignment is any bit position that allows the member to fit entirely within a 64–bit word.
- For a member with a size less than 64 bits, alignment is the same as the size. For example, a `char` has a size and alignment of 8 bits; a `float` has a size and alignment of 32 bits.
- For a member with a size equal to or greater than 64 bits, alignment is 64 bits.
- For a member with array type, alignment is equal to the alignment of the element type.

A plain `int` type bit field is treated as a `signed int` bit field.

The values of an enumeration type are represented in the type `signed int` in C; they are a separate type in C++.

Qualifiers

When an object that has `volatile`-qualified type is accessed, it is simply a reference to the value of the object. If the value is not used, the reference need not result in a load of the value from memory.

Declarators

A maximum of 12 pointer, array, and/or function declarators are allowed to modify an arithmetic, structure, or union type.

Statements

The compiler has no fixed limit on the maximum number of case values allowed in a `switch` statement. The Cray C++ compiler parses asm statements for correct syntax, but otherwise ignores them.

Exceptions

In Cray C++, when an exception is thrown, the memory for the temporary copy of the exception being thrown is allocated on the stack and a pointer to the allocated space is returned.

System Function Calls

For a description of the form of the unsuccessful termination status that is returned from a call to `exit(3)`, see the `exit(3)` man page.

Preprocessing

The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the same character in the execution character set. No such character constant has a negative value. For each, 'a' has the same value in the two contexts:

```
#if 'a' == 97
if ('a' == 97)
```

The `-I` option and the method for locating included source files is described in *-I incldir*.

The source file character sequence in an `#include` directive must be a valid or Cray Linux Environment (CLE) file name or path name. An `#include` directive may specify a file name by means of a macro, provided the macro expands into a source file character sequence delimited by double quotes or < and > delimiters, as follows:

```
#define myheader "./myheader.h"
#include myheader
#define STDIO <stdio.h>
#include STDIO
```

The macros `__DATE__` and `__TIME__` contain the date and time of the beginning of translation.

Library and Linker Use

This topic describes the libraries that are available with the Cray C and C++ compilers and the linker.

Cray C and C++ Libraries

Libraries that support Cray C and C++ are automatically available when using the `CC` or `cc` command to compile the programs. These commands automatically issue the appropriate directives to link the program with the appropriate functions. If the program strictly conforms to the C or C++ standards, there is no need to know library names and locations. If the program requires other libraries or if direct control over the linking process is wanted, more knowledge of the linker and libraries is necessary.

The Standard Template Library (STL) is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template. Be sure there is a complete understanding of templates and how they work before using them.

Linker

When using the `cc` or `CC` command to invoke the compiler, and the program compiles without errors, the linker is called. Specifying the `-c` option on the command line produces relocatable object files (*.o) without calling the linker. These relocatable object files can then be used as input to the linker command by specifying the file names on the appropriate linker command line.

For example, the following command line compiles a file called `target.c` and produces the relocatable object file called `target.o` in the current working directory:

```
cc -c target.c
```

Then use file `target.o` as input to the linker or save the file to use with other relocatable object files to compile and create a linked executable file (`a.out` by default).

Because of the special code needed to handle templates, constructors, destructors, and other C++ language features, object files generated by using the `CC` command should be linked using the `CC` command.

Cray C and C++ Dialect Use

This topic details the features of the C and C++ languages that are accepted by the Cray C and C++ compilers, including certain language dialects and anachronisms. Users should be aware of these details, especially users who are porting codes from other environments.

Cray C++ Language Conformance

The Cray C compiler accepts the C++ language as defined by the ISO/IEC 14882:2003 standard, except for exported templates. C++ supports the ISO 2003 Standard Template Library (STL) headers but abrogates support for pre-standard template headers that have the `.h` extension.

C++ codes that use the pre-standard template headers must be updated to the ISO C++ standard.

The Cray C++ compiler also has a `cfront` compatibility mode, which duplicates a number of features and bugs of `cfront`. Complete compatibility is not guaranteed or intended. The mode allows programmers who have used `cfront` features to continue to compile their existing code. Command line options are also available to enable and disable anachronisms and strict standard-conformance checking.

Cray C++ Features

The following features, which are in the ISO/IEC 14882:2003 standard but not in traditional C++[1], are supported:

- The dependent statement of an if, while, do-while, or for is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.
- The expression tested in an if, while, do-while, or for, as the first operand of a ? operator, or as an operand of the &&, ||, or ! operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.
- A global-scope qualifier is allowed in member references of the form `x.::A::B` and `p->::A::B`.
- The precedence of the third operand of the `?` operator is changed.
- If control reaches the end of the `main()` routine, and the `main()` routine has an integral return type, it is treated as if a `return 0;` statement was executed.
- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.
- A functional-notation cast of the form `A()` can be used even if A is a class without a (nontrivial) constructor.
- The temporary that is created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have nontype template parameters.

- A reference to `const volatile` cannot be bound to an *rvalue*.
- Qualification conversions such as conversion from `T**` to `T const * const` are allowed.
- Digraphs are recognized.
- Operator keywords (for example, `and` or `bitand`) are recognized.
- Static data member declarations can be used to declare member constants.
- `bool` is recognized.
- RTTI (run time type identification), including `dynamic_cast` and the `typeid` operator, is implemented.
- Declarations in tested conditions (within `if`, `switch`, `for`, and `while` statements) are supported.
- Array `new` and `delete` are implemented.
- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast)` are implemented.
- Definition of a nested class outside its enclosing class is allowed.
- `mutable` is accepted on nonstatic data member declarations.
- Namespaces are implemented, including using declarations and directives.
- Access declarations are broadened to match the corresponding using declarations.
- The `typename` keyword is recognized.
- `explicit` is accepted to declare nonconverting constructors.
- The scope of a variable declared in the *for-init-statement* of a `for` loop is the scope of the loop (not the surrounding scope).
- Member templates are implemented.
- The new specialization syntax (using `template <>`) is implemented.
- Type qualifiers, `const` and `volatile` referred to as *cv-qualifiers* are retained on *rvalues* (in particular, on function return values).
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between process overlay directives (PODs) and non-PODs with trivial constructors.
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions).
- A `typedef` name can be used in an explicit destructor call.
- Placement delete is supported.
- An array allocated via a placement new can be deallocated via `delete`.
- `enum` types are considered to be nonintegral types.
- Partial specification of class templates is implemented.
- Partial ordering of function templates is implemented.
- Function declarations that match a function template are regarded as independent functions, not as “guiding declarations” that are instances of the template.
- It is possible to overload operators using functions that take `enum` types and no class types.
- Explicit specification of function template arguments is supported.

- Unnamed template parameters are supported.
- The new lookup rules for member references of the form `x.A::B` and `p->A::B` are supported.
- The notation `:: template` (and `->template`, etc.) is supported.
- In a reference of the form `f()->g()`, with `g` a static member function, `f()` is evaluated. Likewise for a similar reference to a static data member. The ARM specifies that the left operand is not evaluated in such cases.
- `enum` types can contain values larger than can be contained in an `int`.
- Default arguments of function templates and member functions of class templates are instantiated only when the default argument is used in a call.
- String literals and wide string literals have `const` type.
- Argument-dependent (Koenig) lookup of function names is implemented.
- Class and function names declared only in unqualified friend declarations are not visible except for functions found by argument-dependent lookup.
- A `void` expression can be specified on a return statement in a void function.
- `reinterpret_cast` allows casting a pointer to a member of one class to a pointer to a member of another class even when the classes are unrelated.
- Two-phase name binding in templates as described in the Working Paper is implemented.
- Putting a try/catch around the initializers and body of a constructor is implemented.
- `template` parameters are implemented.
- Universal character set escapes (e.g., `\uabcd`) are implemented.
- `extern inline` functions are supported.
- Covariant return types on overriding virtual functions are supported.

C++ Libraries

Most standard C++ features are supported, except the following:

- String classes using basic string class templates with wide character types or that use the `wstring` standard template class.
- I/O streams using wide character objects.
- File-based streams using file streams with wide character types (`wfilebuf`, `wifstream`, `wofstream`, and `wfstream`)
- Multiple localization libraries; Cray C++ supports only one locale

The C++ standard provides a standard naming convention for library routines. Therefore, classes or routines that use wide characters are named appropriately. For example, the `fscanf` and `sprintf` functions do not use wide characters, but the `fwscanf` and `swprintf` function do.

C++ Anachronisms Accepted

C++ anachronisms are enabled by using the `-h anachronisms` command line option. When anachronisms are enabled, the following anachronisms are accepted:

- `overload` is allowed in function declarations. It is accepted and ignored.

- Definitions are not required for static data members that can be initialized by using the default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array can be specified in an array delete operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name can be omitted in a base class initializer if there is only one immediate base class.
- Assignment to the `this` pointer in constructors and destructors is allowed. This is only allowed if anachronisms are enabled and the assignment to this configuration parameter is enabled.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a non-nested class name if no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-`const` type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-`const` class type may be initialized from an rvalue of the class type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and can participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when checking for compatibility, therefore, the following statements declare the overloading of two functions named `f`:

```
int f(int);
int f(x) char x; { return x; }
```

Extensions Accepted in Normal C++ Mode

Certain C++ extensions are accepted except when strict standard conformance mode is enabled, in which case a warning or caution message may be issued.

A `friend` declaration for a class can omit the `class` keyword

```
class B;
class A  {
   friend B;   // Should be "friend class B"
};
```

Constants of scalar type can be defined within classes

```
class A {
    const int size=10;
    int a[size];
};
```

Default assignment operator

An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a “default” assignment operator; that is, such a declaration blocks the implicit generation of a copy assignment operator. This is `cfront` behavior that is known to be relied upon in at least one widely used library.

```
struct A { };
struct B : public A {
   B& operator=(A&);
};
```

By default, as well as in `cfront` compatibility mode, there will be no implicit declaration of `B::operator=(const B&)`, whereas in strict-ANSI mode, `B::operator=(A&)` is not a copy assignment operator and `B::operator=(const B&)` is implicitly declared.

Implicit type conversion between a pointer to an `extern "C"` function and a pointer to an `extern "C++"` function

```
extern "C" void f(); // f's type has extern "C" linkage
void (*pf)()         // pf points to an extern "C++" function
             = &f;   // error unless implicit conversion allowed
```

The ? operator

The `?` operator, for which the second and third operands are string literals or wide string literals, can be implicitly converted to one of the following:

```
char *
wchar_t *
```

In C++ string literals are const. There is a deprecated implicit conversion that allows conversion of a string literal to `char *`, dropping the `const`. That conversion, however, applies only to simple string literals. Allowing it for the result of a `?` operation is an extension:

```
char *p = x ? "abc" : "def";
```

Extensions Accepted in C or C++ Mode

The following extensions are accepted in C or C++ mode except when strict standard conformance modes is enabled, in which case a warning or caution message may be issued.

- The special `lint` comments `/*ARGSUSED*/`, `/*VARARGS*/` (with or without a count of nonvarying arguments), and `/*NOTREACHED*/` are recognized.
- A translation unit (input file) can contain no declarations.
- Comment text can appear at the ends of preprocessing directives.

- Bit fields can have base types that are `enum` or integral types in addition to `int` and `unsigned int`. This corresponds to the ANSI Common Extensions topic.
- `enum` tags can be incomplete as long as the tag name is defined and resolved by specifying the brace-enclosed list later.
- An extra comma is allowed at the end of an `enum` list.
- The final semicolon preceding the closing of a `struct` or `union` type specifier can be omitted.
- A label definition can be immediately followed by a right brace (}). (Normally, a statement must follow a label definition.)
- An empty declaration (a semicolon preceded by nothing) is allowed.
- An initializer expression that is a single value and is used to initialize an entire static array, struct, or union does not need to be enclosed in braces. ANSI C requires braces.
- In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it.
- The address of a variable with register storage class may be taken.
- In an integral constant expression, an integer constant can be cast to a pointer type and then back to an integral type.
- In duplicate size and sign specifiers (for example, `short short` or `unsigned unsigned`) the redundancy is ignored.
- Benign redeclarations of `typedef` names are allowed. That is, a `typedef` name can be redeclared in the same scope with the same type.
- Dollar sign ($) characters can be accepted in identifiers by using the `-h calchars` command line option. This is not allowed by default.
- Numbers are scanned according to the syntax for numbers rather than the pp-number syntax. Thus, `0x123e+1` is scanned as three tokens instead of one token that is not valid. If the `-h conform` option is specified, the pp-number syntax is used.
- Assignment and pointer differences are allowed between pointers to types that are interchangeable but not identical, for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size (for example, `int *` and `long *`). Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `const int **`). Comparisons and pointer difference of such pairs of pointer types are also allowed.
- In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ANSI C, these are allowed by some operators, and not by others (generally, where it does not make sense).
- Pointers to different function types may be assigned or compared for equality (==) or inequality (!=) without an explicit type cast. This extension is not allowed in C++ mode.
- A pointer to `void` can be implicitly converted to or from a pointer to a function type.
- External entities declared in other scopes are visible:

```
void f1(void) { extern void f(); }
void f2() { f(); /* Using out of scope declaration */ }
```

- In C mode, end-of-line comments (//) are supported.
- A non-lvalue array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.
- The `fortran` keyword. For more information, see *fortran Keyword*.
- Cray hexadecimal floating point constants. For more information, see *Hexadecimal Floating-point Constants*.

C++ Extensions Accepted in `cfront` Compatibility Mode

The `cfront` compatibility mode is enabled by the `-h cfront` command-line option. The following extensions are accepted in `cfront` compatibility mode:

- Type qualifiers on the `this` parameter are dropped in contexts such as in the following example:

```
struct A {
   void f() const;
};
void (A::*fp)() = &A::f;
```

  This is a safe operation. A pointer to a `const` function can be put into a pointer to non-`const`, because a call using the pointer is permitted to modify the object and the function pointed to will not modify the object. The opposite assignment would not be safe.
- Conversion operators that specify a conversion to `void` are allowed.
- A nonstandard `friend` declaration can introduce a new type. A `friend` declaration that omits the elaborated type specifier is allowed in default mode, however, in `cfront` mode the declaration can also introduce a new type name. An example follows:

```
struct A {
   friend B;
};
```

- The third operator of the `?` operator is a conditional expression instead of an assignment expression.
- A reference to a pointer type may be initialized from a pointer value without use of a temporary even when the reference pointer type has additional type qualifiers above those present in the pointer value. For example:

```
int *p;
const int *&r = p;   // No temporary used
```

- A reference can be initialized to `NULL`.
- Because `cfront` does not check the accessibility of types, access errors for types are issued as warnings instead of errors.
- When matching arguments of an overloaded function, a `const` variable with a value of 0 is not considered to be a null pointer constant. In general, in overload resolution, a null pointer constant must be entered as "0" to be considered a null pointer constant (e.g., '\0' is not considered a null pointer constant).
- An alternate form of declaring pointer-to-member-function variables is supported, as shown in the following example:

```
struct A {
 void f(int);
 static void sf(int);
 typedef void A::T3(int);  // nonstd typedef decl
```

```
 typedef void T2(int);     // std typedef
};
typedef void A::T(int);    // nonstd typedef decl
T* pmf = &A::f;            // nonstd ptr-to-member decl
A::T2* pf = A::sf;         // std ptr to static mem decl
A::T3* pmf2 = &A::f;       // nonstd ptr-to-member decl
```

In this example, `T` is construed to name a function type for a nonstatic member function of class `A` that takes an `int` argument and returns void; the use of such types is restricted to nonstandard pointer-to-member declarations. The declarations of `T` and `pmf` in combination are equivalent to the following single standard pointer-to-member declaration:

```
void (A::* pmf)(int) = &A::f;
```

A nonstandard pointer-to-member declaration that appears outside of a class declaration, such as the declaration of `T`, is normally not valid and would cause an error to be issued. However, for declarations that appear within a class declaration, such as `A::T3`, this feature changes the meaning of a valid declaration. `cfront` version 2.1 accepts declarations, such as T, even when A is an incomplete type; so this case is also accepted.

- Protected member access checking is not done when the address of a protected member is taken. For example:
-

```
class B { protected: int i; };
class D : public B { void mf()};

void D::mf() {
 int B::* pmi1 = &B::i; // error, OK in cfront mode
 int D::* pmi2 = &D::i; // OK
}
```

Protected member access checking for other operations (such as everything except taking a pointer-to-member address) is done normally

- The destructor of a derived class can implicitly call the private destructor of a base class. In default mode, this is an error but in `cfront` mode it is reduced to a warning. For example:

```
class A {
   ~A();
};
class B : public A {
   ~B();
};
B::~B(){}           // Error except in cfront mode
```

- When disambiguation requires deciding whether something is a parameter declaration or an argument expression, the pattern *`type-name-or-keyword (identifier ...)`* is treated as an argument. For example:

```
class A { A(); };
double d;
A x(int(d));
A(x2);
```

By default, `int(d)` is interpreted as a parameter declaration (with redundant parentheses), and so `x` is a function; but in `cfront` compatibility mode `int(d)` is an argument and `x` is a variable.

The declaration `A(x2)` is also misinterpreted by `cfront`. It should be interpreted as the declaration of an object named `x2`, but in `cfront` mode it is interpreted as a function style cast of `x2` to the type A.

Similarly, the following declaration declares a function named `xzy`, that takes a parameter of type function taking no arguments and returning an `int`. In `cfront` mode, this is interpreted as a declaration of an object that is initialized with the value `int()`, which evaluates to 0.

```
int xyz(int());
```

- A named bit field can have a size of 0. The declaration is treated as though no name had been declared.
- Plain bit fields (such as bit fields declared with a type of `int`) are always signed.
- The name given in an elaborated type specifier can be a `typedef` name that is the synonym for a class name. For example:

```
typedef class A T;
class T *pa;                    // No error in cfront mode
```

- No warning is issued on duplicate size and sign specifiers, as shown in the following example:

```
short short int i;  // No warning in cfront mode
```

- Virtual function table pointer-update code is not generated in destructors for base classes of classes without virtual functions, even if the base class virtual functions might be overridden in a further derived class. For example:

```
struct A {
  virtual void f() {}
  A() {}
  ~A() {}
};
struct B : public A {
  B() {}
  ~B() {f();} // Should call A::f according to ARM 12.7
};
struct C : public B {
  void f() {}
} c;
```

In `cfront` compatibility mode, `B::~B calls C::f`.

- An extra comma is allowed after the last argument in an argument list. For example:

```
f(1, 2, );
```

- A constant pointer-to-member function can be cast to a pointer-to-function, as in the following example. A warning is issued.

```
struct A {int f();};
main () {
int (*p)();
p = (int (*)())A::f;                    // Okay, with warning
}
```

- Arguments of class types that allow bitwise copy construction but also have destructors are passed by value like C structures, and the destructor is not called on the copy. In normal mode, the class object is copied into

a temporary, the address of the temporary is passed as the argument, and the destructor is called on the temporary after the call returns. Because the argument is passed by value instead of by address, code like this compiled in `cfront` mode is not calling-sequence compatible with the same code compiled in normal mode. In practice, this is not much of a problem, since classes that allow bitwise copying usually do not have destructors.

- A union member may be declared to have the type of a class for which the user has defined an assignment operator (as long as the class has no constructor or destructor). A warning is issued.
- When an unnamed class appears in a `typedef` declaration, the `typedef` name may appear as the class name in an elaborated type specifier. For example:

```
typedef struct { int i, j; } S;
struct S x;                          // No error in cfront mode
```

- Two member functions may be declared with the same parameter types when one is static and the other is nonstatic with a function qualifier. For example:

```
class A {
  void f(int) const;
  static void f(int); // No error in cfront mode
};
```

- The scope of a variable declared in the `for-init-statement` is the scope to which the `for` statement belongs. For example:

```
int f(int i) {
  for (int j = 0; j < i; ++j) { /* ... */ }
  return j; // No error in cfront mode
}
```

- Function types differing only in that one is declared `extern "C"` and the other `extern "C++"` can be treated as identical:

```
typedef void (*PF)();
extern "C" typedef void (*PCF)();
void f(PF);
void f(PCF);
```

By contrast, in standard C++, PF and PCF are different and incompatible types; PF is a pointer to an `extern "C++"` function whereas PCF is a pointer to an `extern "C"` function; and the two declarations of f create an overload set.

- Functions declared `inline` have internal linkage.
- `enum` types are regarded as integral types.
- An uninitialized const object of non-POD class type is allowed even if its default constructor is implicitly declared as in the following example:

```
struct A { virtual void f(); int i; };
const A a;
```

- A function parameter type is allowed to involve a pointer or reference to array of unknown bounds.

- If the user declares an `operator=` function in a class, but not one that can serve as the default `operator=`, and bitwise assignment could be done on the class, a default `operator=` is not generated. Only the user-written `operator=` functions are considered for assignments, so bitwise assignment is not done.

Compiler Message System Use

This topic describes how to use the message system to control and use messages issued by the compiler. Explanatory texts for messages can be displayed online through the use of the `explain` command.

Expand Messages with the `explain` Command

Use the `explain` command to display an explanation of any message issued by the compiler. The command takes as an argument, the message number, including the number's prefix. The prefix for Cray C and C++ is `CC`.

In the following sample dialog, the `cc` command invokes the compiler on source file `bug.c`. Message `CC-24` is displayed. The `explain` command displays the expanded explanation for this message.

```
% 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.
```

Control the Use of Messages

This section summarizes the command line options that affect the issuing of messages from the compiler.

Command Line Options

`-h errorlimit[=`*`n`*`]`	Specifies the maximum number of error messages the compiler prints before it exits.
`-h [no]message=`*`n`*`[:...]`	Enables or disables the specified compiler messages, overriding `-h msglevel`.
**`-h msglevel_`*`n`*	Specifies the lowest severity level of messages to be issued.
`-h [no]msgs`	Enables or disables the writing of optimization messages to `stderr`.
`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.
**`-h report=`*`args`*	Generates optimization report messages.

Environment Options for Messages

The following are used by the message system:

NLSPATH	Specifies the default value of the message system search path environment variable.
LANG	Identifies the requirements for native language, local customs, and coded character set with regard to the message system.
MSG_FORMAT	Controls the format in which error messages are received.

ORIG_CMD_NAME Environment Variables

Override the command name printed in the message. If the environment variable ORIG_CMD_NAME is set, the value of ORIG_CMD_NAME is used as the command name in the message. This functionality is provided for use with shell scripts that invoke the compiler. By setting ORIG_CMD_NAME to the name of the script, any message printed by the compiler appears as though it was generated by the script. For example, the following C shell script is named newcc:

```
#
setenv ORIG_CMD_NAME 'basename $0'
cc $*
```

A message generated by invoking newcc resembles the following:

```
CC-8 newcc: ERROR File = x.c, Line = 1
  A new-line character appears inside a string literal.
```

Because the environment variable ORIG_CMD_NAME is set to newcc, this appears as the command name instead of cc in this message.

The ORIG_CMD_NAME environment variable is not part of the message system. It is supported by the Cray C and C++ compilers as an aid to programmers. Other products, such as the Fortran compiler and the linker, may support this variable. However, do not rely on support for this variable in any other product.

Be careful when setting the environment variable ORIG_CMD_NAME. If setting ORIG_CMD_NAME inadvertently, the compiler may generate messages with an incorrect command name. This may be particularly confusing if, for example, ORIG_CMD_NAME is set to newcc when the Fortran compiler prints a message. The Fortran message will look as though it came from newcc.

Message Severity

Each message issued by the compiler falls into one of the following categories of messages, depending on the severity of the error condition encountered or the type of information being reported.

COMMENT	Inefficient programming practices.
NOTE	Unusual programming style or the use of outmoded statements.
CAUTION	Possible user error. Cautions are issued when the compiler detects a condition that may cause the program to abort or behave unpredictably.
WARNING	Probable user error. Indicates that the program will probably abort or behave unpredictably.
ERROR	Fatal error; that is, a serious error in the source code. No binary output is produced.

INTERNAL	Problems in the compilation process. Please report internal errors immediately to the system support staff, so that a bug report can be filed.
LIMIT	Compiler limits have been exceeded. Normally, the source code or environment can be modified to avoid these errors. If limit errors cannot be resolved by such modifications, please report these errors to the system support staff, so that a bug report can be filed.
INFO	Useful additional information about the compiled program.
INLINE	Information about inline code expansion performed on the compiled code.
SCALAR	Information about scalar optimizations performed on the compiled code.
VECTOR	Information about vectorization optimizations performed on the compiled code.
OPTIMIZATION	Information about general optimizations.
IPA_INFO	Information about interprocedural optimizations.

Common System Messages

The errors in the following list can occur during the execution of a user program. The operating system detects them and issues the appropriate message. These errors are not detected by the compiler and are not unique to C and C++ programs; they may occur in any application program written in any language.

- `Operand Range Error`

  An operand range error occurs when a program attempts to load or store in an area of memory that is not part of the user's area. This usually occurs when an invalid pointer is dereferenced.

- `Program Range Error`

  A program range error occurs when a program attempts to jump into an area of memory that is not part of the user's area. This may occur, for example, when a function in the program mistakenly overwrites the internal program stack. When this happens, the address of the function from which the function was called is lost. When the function attempts to return to the calling function, it jumps elsewhere instead.

- `Error Exit`

  An error exit occurs when a program attempts to execute an invalid instruction. This error usually occurs when the program's code area has been mistakenly overwritten with words of data (for example, when the program stores in a location pointed to by an invalid pointer).

Intrinsic Functions

The C and C++ intrinsic functions either allow for direct access to some hardware instructions or result in generation of inline code to perform some specialized functions. These intrinsic functions are processed completely by the compiler. In many cases, the generated code is one or two instructions. These are called functions because they are invoked with the syntax of function calls.

To get access to most of the intrinsic functions, the Cray C++ compiler requires that either the `intrinsics.h` file be included or that the intrinsic functions being called are explicitly declared. If the source code does not have an intrinsics.h statement and the code cannot be modified, use the `-h prototype_intrinsics` option instead. If an intrinsic function is intrinsically declared, the declaration must agree with the documentation or the compiler treats the call as a call to a normal function, not the intrinsic function. The `-h nointrinsics` command line option causes the compiler to treat these calls as regular function calls and not as intrinsic function calls.

There are built-in atomic memory intrinsic functions of the form `__sync_*` that do not require an include file nor any explicit declaration.

The types of the arguments to intrinsic functions are checked by the compiler, and if any of the arguments do not have the correct type, a warning message is issued and the call is treated as a normal call to an external function. If the intention was to call an external function with the same name as an intrinsic function, change the external function name. The names used for the Cray C intrinsic functions are in the name space reserved for the implementation.

Several of these intrinsic functions have both a vector and a scalar version. If a vector version of an intrinsic function exists and the intrinsic is called within a vectorized loop, the compiler uses the vector version of the intrinsic. For details on whether it has a vector version, refer to the appropriate intrinsic function man page.

Atomic Memory Operations

Atomic memory operations (AMOs), unlike other functions, cannot be interrupted by the system and can allow multiple threads to safely modify the same variable under certain conditions. The AMO intrinsics allow for the adding, subtracting, AND, NAND, OR, and XOR values together, or comparing and swapping values.

Local AMOs operate on variables in the processor's local memory (cache domain); they do not use the network interface to access memory. Multiple threads using local atomic memory operations to access the same variable need to be running within the same processor cache domain, which implies that they must be running on the same node. Local AMOs are atomic with respect to each other. The compiler issues an error message if a user tries to apply a local AMO intrinsic to a Unified Parallel C or shared variable or Fortran coarray that is not local to the current thread.

Global AMOs use the network interface to access variables in memory. The variables may or may not be in the processor's local cache domain. Global AMOs are atomic with respect to each other. Global AMOs are used to modify a Unified Parallel C (UPC) shared variable or Fortran coarray and are available only when compiling UPC (`-hupc`) or coarray Fortran (`-hcaf`).

A global AMO uses a different mechanism for achieving atomicity than a local AMO, so local and global AMOs are not atomic with respect to each other. Global and local AMOs should not be used concurrently on the same memory location, without synchronization.

It is possible to safely modify a variable using both atomic and non-atomic operations within a single UPC thread or Fortran image; however, if a thread or image modifies a variable with an atomic operation and a different thread or image concurrently modifies the same variable with a non-atomic operation, the result is indeterminate.

Global atomic memory operations (global AMO) are typically used to atomically modify a Unified Parallel C (UPC) shared variable or Fortran coarray. The target of a global AMO can be located in a different cache domain, so a global AMO is not atomic with respect to memory operations performed locally within the target's cache domain. Therefore, the application must use synchronization to ensure that global AMOs and local memory operations are not used concurrently on the same memory location.

For synopses of AMO functions, see the `amo(3i)` man page.

Local Atomic Memory Operations

The following functions, defined in `intrinsics.h`, perform various local atomic memory operations:

`__builtin_ia32_lfence`	(Load fence) Insures that all memory loads issued before this intrinsic are visible in memory before any future loads are executed.
`__builtin_ia32_sfence`	(Store fence) Insures that all memory stores issued before this intrinsic are visible in memory before any future stores are executed.
`__builtin_ia32_mfence`	(Memory fence) Insures that all memory stores and loads issued before this intrinsic are visible in memory before any future stores or loads are executed.

Functions built into the compiler do not require an include file, nor a specific compilation option for use. The following local atomic, built-in functions return the value of the object **before** the named operation occurs.

In this discussion, an object is an entity that is referred to by a pointer. A value is an actual number, bit mask, etc. that is not referred to by a pointer. The allowed object and value types are signed and unsigned integer types of 1, 2, 4, or 8 bytes.

- The `__sync_fetch_and_add` function fetches the object pointed to by `ptr`, adds `value`, places the result into the object pointed to by `ptr`, and returns the old value of the object pointed to by `ptr`.
- The `__sync_fetch_and_sub` function fetches the object pointed to by `ptr`, subtracts `value`, places the result into the object pointed to by `ptr`, and returns the old value of the object pointed to by `ptr`.
- The `__sync_fetch_and_or` function fetches the object pointed to by `ptr`, `ORs value`, places the result into the object pointed to by `ptr`, and returns the old value of the object pointed to by `ptr`.
- The `__sync_fetch_and_and` function fetches the object pointed to by `ptr`, `ANDs value`, places the result into the object pointed to by `ptr`, and returns the old value of the object pointed to by `ptr`.
- The `__sync_fetch_and_xor` function fetches the object pointed to by `*ptr`, `XORs value`, places the result into the object pointed to by `ptr`, and returns the old value of the object pointed to by `ptr`.
- The `__sync_fetch_and_nand` function fetches the object pointed to by `ptr`, `NANDs value`, places the result into the object pointed to by `ptr`, and returns the old value of the object pointed to by `ptr`.

The following local atomic, built-in functions return the value of the object after the named operation occurs:

- The `__sync_add_and_fetch` function adds `value` to the object pointed to by `ptr` and returns the new value of the object pointed to by `ptr`.
- The `__sync_sub_and_fetch` function subtracts `value` from the object pointed to by `ptr` and returns the new value of the object pointed to by `ptr`.

- The `__sync_or_and_fetch` function `ORs value` with the object pointed to by `ptr` and returns the new value of the object pointed to by `ptr`.
- The `__sync_and_and_fetch` function `ANDs value` with the object pointed to by `ptr` and returns the new value of the object pointed to by `ptr`.
- The `__sync_xor_and_fetch` function `XORs value` with the object pointed to by `ptr` and returns the new value of the object pointed to by `ptr`.
- The `__sync_nand_and_fetch` function `NANDs value` with the current value of `ptr` and returns the new contents of `ptr`.
- The `__sync_val_compare_and_swap` function performs an atomic compare and swap. If the current value of `*ptr` is `compareValue`, then write `replacementValue` into `*ptr` and return the contents of `*ptr` before the operation.
- The `__sync_lock_test_and_set` function writes *`value`* into *`*ptr`*, and returns the previous contents of *`*ptr`*.

Global Atomic Memory Operations

Global atomic memory operations (global AMO) are typically used to atomically modify a Unified Parallel C (UPC) shared variable or Fortran coarray.

The target of a global AMO can be located in a different cache domain, so a global AMO is not atomic with respect to memory operations performed locally within the target's cache domain. Therefore, the application must use synchronization to ensure that global AMOs and local memory operations are not used concurrently on the same memory location.

The following intrinsics are defined in `intrinsics.h`. Functions without the `_upc` suffix accept both shared and non-shared pointers as the first argument. Functions with the `_upc` suffix accept only shared pointers as the first argument.

In this discussion, an object is an entity that is referred to by a pointer. A value is an actual number, bit mask, etc. that is not referred to by a pointer.

- The `_amo_aadd` and `_amo_aadd_upc` functions (atomic add) add `value` to the object pointed to by `ptr`.
- The `_amo_aaddf` and `_amo_aaddf` functions (atomic add and fetch) add `value` to the object pointed to by `ptr` and return the new value.
- The `_amo_afadd` and `_amo_afadd_upc` functions (atomic fetch and add) add `value` to the object pointed to by `ptr` and return the old value of the object.
- The `_amo_aax` and `_amo_aax_upc` functions (atomic AND and XOR) `AND` the object pointed to by `ptr` with `andMask`, `XOR` the result with `xorMask`, and place the result into the object.
- The `_amo_afax` and `_amo_afax_upc` functions (atomic fetch and AND and XOR) `AND` the object pointed to by `ptr` with `andMask`, `XOR` the result with `xorMask`, place the result into the object, and return the old value of the object.
- The `_amo_aandf` and `_amo_aandf_upc` functions (atomic AND and fetch) `AND` the object pointed to by `ptr` with `value`, place the result into the object, and return the new value of the object.
- The `_amo_afand` and `_amo_afand_upc` functions (atomic fetch and AND) `AND` the object pointed to by `ptr` with `value`, place the result into the object, and return the old value of the object.
- The `_amo_anandf` and `_amo_anandf_upc` functions (atomic NAND and fetch) `NAND` the object pointed to by `ptr` with `value`, place the result into the object, and return the new value of the object.

- The `_amo_afnand` and `_amo_afnand_upc` functions (atomic fetch and NAND) `NAND` the object pointed to by `ptr` with `value`, place the result into the object, and return the old value of the object.
- The `_amo_aorf` and `_amo_aorf_upc` functions (atomic OR and fetch) `OR` the object pointed to by `ptr` with `value`, place the result into the object, and return the new value of the object.
- The `_amo_afor` and `_amo_afor_upc` functions (atomic fetch and OR) `OR` the object pointed to by `ptr` with `value`, place the result into the object, and return the old value of the object.
- The `_amo_axorf` and `_amo_axorf_upc` functions (atomic XOR and fetch) `XOR` the object pointed to by `ptr` with `value`, place the result into the object, and return the new value of the object.
- The `_amo_afxor` and `_amo_afxor_upc` functions (atomic fetch and XOR) `XOR` the object pointed to by `ptr` with `value`, place the result into the object, and returns the old value of the object.
- The `_amo_acswap` and `_amo_acswap_upc` functions (atomic compare and swap) compare and swap a value by replacing the contents of the object pointed to by `ptr` with `replacementValue` if `compareValue` is equal to the object pointed to by `ptr` and return the old value of the object.
- The `_amo_aswap` and `_amo_aswap_upc` functions (atomic swap) swap a value by replacing the contents of the object pointed to by `ptr` with `replacementValue`. This function always returns the old value.
- The `_amo_aflush` and `_amo_aflush_upc` functions (atomic flush) force `*ptr` to be written to memory.

For more information, see the `amo(3i)` man page.

Intrinsic Bit Operations

These instrinsic functions copy, count, or shift bits, or compute the parity bit.

`_dshiftl`

Move the left most n bits of an integer into the right side of another integer, and return that integer.

`_dshiftr`

Move the right most n bits of an integer into the left side of another integer and return that integer.

`_pbit`

Copies the rightmost bit of a word to the nth bit, from the right, of another word.

`_pbits`

Copies the rightmost m bits of a word to another word beginning at bit n.

`_poppar`

Computes the parity bit for a variable.

`_popcnt`
`_popcnt32`
`_popcnt64`

Counts the number of set bits in 32-bit and 64-bit integer words.

`_leadz`
`_leadz32`
`_leadz64`

Counts the number of leading 0 bits in 32-bit and 64-bit integer words.

`_gbit`

_gbit returns the value of the nth bit from the right.

_gbits

Returns a value consisting of m bits extracted from a variable, beginning at nth bit from the right.

Intrinsic Mask Operations

These instrinsic functions create bit masks.

_mask

Creates a left-justified or right-justified bit mask with all bits set to 1.

_mask1

Returns a left-justified bit mask with i bits set to 1.

_maskr

Returns a right-justified bit mask with i bits set to 1.

Miscellaneous Intrinsic Operations

These instrinsic functions perform various operations.

_int_mult_upper

Multiplies integers and returns the uppermost bits.

_ranf

Computes a pseudo-random floating-point number ranging from 0.0 through 1.0.

_rtc

Return a real-time clock value expressed in clock ticks.