IBM XL C/C++ Advanced Edition V8.0 for Blue Gene
IBM XL Fortran Advanced Edition V10.1 for Blue Gene

Using the XL Compilers for Blue Gene

SC10-4310-00

IBM XL C/C++ Advanced Edition V8.0 for Blue Gene
IBM XL Fortran Advanced Edition V10.1 for Blue Gene

Using the XL Compilers for Blue Gene

SC10-4310-00

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 41.

First Edition

This edition applies to:

- IBM XL C/C++ Advanced Edition V8.0 for Blue Gene (Program number 5799-TRG)
- IBM XL Fortran Advanced Edition V10.1 for Blue Gene (Program number 5799-TRH)

and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

IBM welcomes your comments. You can send your comments electronically to the e-mail address listed below. Be sure to include your e-mail address if you wish a reply.

- compinfo@ca.ibm.com

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About this document

The IBM® XL family of optimizing compilers allows you to develop C, C++, and Fortran applications for Blue Gene/L, and comprises the following products:

- IBM XL C/C++ Advanced Edition V8.0 for Blue Gene®
- IBM XL Fortran Advanced Edition V10.1 for Blue Gene

This document discusses specific considerations for developing, compiling, and optimizing C/C++ and Fortran applications for the Blue Gene/L PowerPC® 440d processor architecture and its Double Hummer floating-point unit.

Who should read this document

This document is for anyone who is developing or cross-compiling applications for the Blue Gene supercomputer, is familiar with the Linux® operating system, and who has some previous C, C++, or Fortran programming experience. Users new to the XL compilers can use this document to find information on the capabilities and features unique to the Blue Gene version of the products.

How to use this document

This document is an add-on to the documentation suites for the XL C/C++ Advanced Edition V8.0 for Linux and XL Fortran Advanced Edition V10.1 for Linux products. It covers only material that is specific to the Blue Gene implementation of the compilers, but does not discuss information that is common to other Linux distributions.

How this document is organized

This document includes the following topics:

- Compiling and linking applications on Blue Gene/L describes the commands to cross-compile applications for Blue Gene, the Blue Gene compiler option defaults, Blue Gene specific compiler options, and unsupported compiler options.
- Tuning your code for Blue Gene/L describes optimization strategies that best exploit the SIMD capabilities of the Blue Gene/L 440d processor. Topics include relevant optimization options, structuring data in adjacent pairs, inlining functions, aliasing, structuring computations, and checking for data alignment.
- Using the high performance libraries describes the Mathematical Acceleration Subsystem (MASS) library of tuned scalar and vector functions available for Blue Gene.
- Using XL built-in floating-point functions for Blue Gene/L summarizes the built-in functions that are specifically optimized for the 440d processor's Double Hummer dual FPU.

Conventions used in this document

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Commands, executable names, compiler options and pragma directives	If you specify **-O3**, the compiler assumes **-qhot=level=0**. To prevent all HOT optimizations with **-O3**, you must specify **-qnohot**.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms	Make sure that you update the *size* parameter if you return more than the *size* requested.
`monospace`	Programming keywords and library functions, compiler built-in functions, examples of program code, command strings, or user-defined names	If one or two cases of a `switch` statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the `switch` statement.
UPPERCASE bold	Fortran programming keywords, statements, directives, and intrinsic procedures	The **ASSERT** directive applies only to the **DO** loop immediately following the directive, and not to any nested **DO** loops.
lowercase bold	Fortran lowercase programming keywords and library functions, compiler intrinsic procedures, file and directory names, examples of program code, command strings, or user-defined names	If you specify **-O3**, the compiler assumes **-qhot=level=0**. To prevent all HOT optimizations with **-O3**, you must specify **-qnohot**.

Syntax diagrams

Throughout this document, diagrams illustrate XL C/C++ or XL Fortran syntax. This section will help you to interpret and use those diagrams.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►─── symbol indicates the beginning of a command, directive, or statement.

  The ───► symbol indicates that the command, directive, or statement syntax is continued on the next line.

  The ►─── symbol indicates that a command, directive, or statement is continued from the previous line.

  The ───►◄ symbol indicates the end of a command, directive, or statement.

  Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the │─── symbol and end with the ───│ symbol.

  IBM XL Fortran extensions are marked by a number in the syntax diagram with an explanatory note immediately following the diagram. Program units, procedures, constructs, interface blocks and derived-type definitions consist of several individual statements. For such items, a box encloses the syntax representation, and individual syntax diagrams show the required order for the equivalent Fortran statements.
- Required items are shown on the horizontal line (the main path):

```
►►──keyword──required_argument─────────────────────────────────────────►◄
```

- Optional items are shown below the main path:

```
►►──keyword──┬───────────────────┬─────────────────────────────────────►◄
             └─optional_argument─┘
```

- If you can choose from two or more items, they are shown vertically, in a stack.

  If you *must* choose one of the items, one item of the stack is shown on the main path.

```
►►──keyword──┬─required_argument1─┬────────────────────────────────────►◄
             └─required_argument2─┘
```

  If choosing one of the items is optional, the entire stack is shown below the main path.

```
►►──keyword──┬────────────────────┬────────────────────────────────────►◄
             ├─optional_argument1─┤
             └─optional_argument2─┘
```

- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

```
             ┌─,───────────────────┐
             ▼                     │
►►──keyword────repeatable_argument─┴───────────────────────────────────►◄
```

- The item that is the default is shown above the main path.

```
             ┌─default_argument───┐
►►──keyword──┴─alternate_argument─┴────────────────────────────────────►◄
```

- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sample syntax diagrams

The following syntax diagram example shows the syntax for the **#pragma comment** directive.

```
    (1)     (2)         (3)          (4)     (5)                                                         (9)  (10)
►►──────#─────────pragma─────────comment─────────(─────┬─compiler─────────────────────────────────────────┬─)──────────────────►◄
                                                      ├─date─────────────────────────────────────────────┤
                                                      ├─timestamp────────────────────────────────────────┤
                                                      │                 (6)                              │
                                                      └─┬─copyright─┬──┬─────────────────────────────────┬┘
                                                        └─user──────┘  │  (7)                       (8)  │
                                                                       └─,──────"──token_sequence──"─────┘
```

Notes:

1 This is the start of the syntax diagram.

2 The symbol `#` must appear first.

3 The keyword `pragma` must appear following the `#` symbol.

4 The name of the pragma `comment` must appear following the keyword `pragma`.

5 An opening parenthesis must be present.

6 The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`.

7 A comma must appear between the comment type `copyright` or `user`, and an optional character string.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the **#pragma comment** directive are syntactically correct according to the diagram shown above:

```
#pragma
comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

How to read syntax statements

Syntax statements are read from left to right:
- Individual required arguments are shown with no special notation.
- When you must make a choice between a set of alternatives, they are enclosed by { and } symbols.
- Optional arguments are enclosed by [and] symbols.
- When you can select from a group of choices, they are separated by | characters.
- Arguments that you can repeat are followed by ellipses (...).

Example of a syntax statement

EXAMPLE *char_constant* {*a*|*b*}[*c*|*d*]*e*[,*e*]... *name_list*{*name_list*}...

The following list explains the syntax statement:
- Enter the keyword EXAMPLE.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Optionally, enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each *name*.

Note: The same example is used in both the syntax-statement and syntax-diagram representations.

Examples

The examples in this document, except where otherwise noted, are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible methods to achieve a specific result.

Notes on path names

The path names shown in this document assume the default installation path for the XL C/C++ or XL Fortran compiler. By default, the compiler will be installed in the following directory on the selected disk:

XL C/C++	`/opt/ibmcmp/vacpp/bg/8.0/`
XL Fortran	`/opt/ibmcmp/xlf/bg/10.1/`

You can select a different destination (*relocation-path*) for the compiler. If you choose a different path, the compiler will be installed in the following directory:

XL C/C++	*relocation-path*`/opt/ibmcmp/vacpp/bg/8.0/`
XL Fortran	*relocation-path*`/opt/ibmcmp/xlf/bg/10.1/`

Related information

IBM XL C/C++ and XL Fortran documentation

Product documentation is provided in the following formats:

- README files

  README files contain late-breaking information, including changes and corrections to the product documentation. README files are located in the root directory of the installation CD and by default in the following directory:

XL C/C++	`/opt/ibmcmp/vacpp/bg/8.0/`
XL Fortran	`/opt/ibmcmp/xlf/bg/10.1/`

- Installable man pages

  Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *XL C/C++ Advanced Edition V8.0 for Blue Gene Installation Guide* and *XL Fortran Advanced Edition V10.1 for Blue Gene Installation Guide*.

- Information center

  The information center of searchable HTML files can be launched on a network and accessed remotely or locally. HTML documents are located by default in the following directory:

XL C/C++	`/opt/ibmcmp/vacpp/bg/8.0/doc/en_US/html`
XL Fortran	`/opt/ibmcmp/xlf/bg/10.1/doc/en_US/html`

- PDF documents

  PDF documents are located by default in the following directory:

XL C/C++	`/opt/ibmcmp/vacpp/bg/8.0/doc/en_US/pdf`
XL Fortran	`/opt/ibmcmp/xlf/bg/10.1/doc/en_US/pdf`

  These are also available on the Web at:

  ```
  http://www.ibm.com/software/awdtools/xlcpp/library
  ```

  or

  ```
  http://www.ibm.com/software/awdtools/fortran/xlfortran/library
  ```

  In addition to this document, the following documents comprise the full set of PDF files shipped with this product. TheXL C/C++ for Linux and XL Fortran for Linux PDF files are included as additional references to complement the topics not covered in the Blue Gene specific documents.

Table 2. PDF files shipped with XL C/C++ for Blue Gene

Document title	PDF file name	Description
IBM XL C/C++ Advanced Edition V8.0 for Blue Gene Installation Guide, GC10-4311	install.pdf	Contains information for installing XL C/C++ for Blue Gene and configuring your environment for basic compilation and program execution.
IBM XL C/C++ Advanced Edition V8.0 for Linux Getting Started with XL C/C++, SC09-8015	getstart.pdf	Contains an introduction to theXL C/C++ for Linux product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference, SC09-8013	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions, including those used for parallel processing.
IBM XL C/C++ Advanced Edition V8.0 for Linux Language Reference, SC09-8016	language.pdf	Contains information about the C and C++ programming languages, as supported by IBM, including language extensions for portability and conformance to non-proprietary standards.
IBM XL C/C++ Advanced Edition V8.0 for Linux Programming Guide, SC09-8014	proguide.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization and parallelization, and the XL C/C++ for Linux high-performance libraries.

Table 3. PDF files shipped with XL Fortran for Blue Gene

Document title	PDF file name	Description
IBM XL Fortran Advanced Edition V10.1 for Blue Gene Installation Guide, GC10-4312	install.pdf	Contains information for installing XL Fortran for Blue Gene and configuring your environment for basic compilation and program execution.
IBM XL Fortran Advanced Edition V10.1 for Linux Getting Started with XL Fortran, SC09-8021	getstart.pdf	Contains an introduction to the XL Fortran for Linux product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL Fortran Advanced Edition V10.1 for Linux Compiler Reference, SC09-8019	cr.pdf	Contains information about the various compiler options and environment variables.
IBM XL Fortran Advanced Edition V10.1 for Linux Language Reference, SC09-8018	lr.pdf	Contains information about the Fortran programming language as supported by IBM, including language extensions for portability and conformance to non-proprietary standards, compiler directives and intrinsic procedures.
IBM XL Fortran Advanced Edition V10.1 for Linux Optimization and Programming Guide, SC09-8022	opg.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls, floating-point operations, input/output, application optimization and parallelization, and the XL Fortran high-performance libraries.

More documentation related to the compilers, including red books, white papers, tutorials, and other articles, is available on the Web at:

```
http://www.ibm.com/software/awdtools/xlcpp/library
```

or

```
http://www.ibm.com/software/awdtools/fortran/xlfortran/library
```

Other IBM publications

- *Unfolding the IBM eServer™ Blue Gene Solution*, SG24-6686-00, available at http://www.redbooks.ibm.com
- *Blue Gene/L: Application Development*, SG24-7179-01, available at http://www.redbooks.ibm.com
- Blue Gene literature, such as white papers, available at http://www.ibm.com/servers/deepcomputing/bluegene_literature.html
- *ESSL for Linux on POWER™ V4.2 Guide and Reference*, SA22-7904-02, available at http://publib.boulder.ibm.com/infocenter/clresctr/index.jsp

Technical support

Additional technical support is available from the XL C/C++ or XL Fortran Support page. This page provides a portal with search capabilities to a large selection of technical support FAQs and other support documents. You can find the Support page on the Web at:

```
http://www.ibm.com/software/awdtools/xlcpp/support
```

or

```
http://www.ibm.com/software/awdtools/fortran/xlfortran/support
```

If you cannot find what you need, you can e-mail:

compinfo@ca.ibm.com

For the latest information about XL C/C++ or XL Fortran, visit the product information site at:

```
http://www.ibm.com/software/awdtools/xlcpp
```

or

```
http://www.ibm.com/software/awdtools/fortran/xlfortran
```

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this document or any other XL C/C++ or XL Fortran documentation, send your comments by e-mail to:

compinfo@ca.ibm.com

Be sure to include the name of the document, the part number of the document, the version of the product, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

Chapter 1. Compiling and linking applications for Blue Gene

This section contains information about compiling and linking applications that will run on Blue Gene/L.

For complete information about compiler and linker options, refer to the following, additional documents:
- *XL C/C++ Advanced Edition V8.0 for Linux Getting Started with XL C/C++*
- *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*
- *XL Fortran Advanced Edition V10.1 for Linux Getting Started with XL Fortran*
- *XL Fortran Advanced Edition V10.1 for Linux Compiler Reference*

Compiling programs

To compile a source program, use any of the available XL C/C++ or XL Fortran for Blue Gene compiler invocation commands. The compiler commands prefixed with **blrts_** are for cross-compiling applications for use on Blue Gene. The compiler invocations that are **not** prefixed with **blrts_** create executables targeted for the SLES9 platform, and are provided only for testing and debugging purposes. For the development of applications targeted for the SLES9 platform, IBM provides the XL C/C++ Advanced Edition for Linux and XL Fortran Advanced Edition for Linux products. As well, only the compiler options which are supported by the **blrts_** invocations are supported when using these compiler invocations to create executables for the SLES9 platform.

These commands use the following syntax, where *invocation* can be replaced with any valid compiler invocation command:

```
               ┌──────────────────────────────────────────┐
               ▼                                          │
►►──invocation──┬──────────────────────┬─┬─────────────┬──┴──────────────►◄
                └─command_line_options─┘ └─input_files─┘
```

The parameters of the compiler invocation command can be the names of input files, compiler options, and linkage-editor options. These commands accept essentially the same or XL Fortran language but use different default options. Read the appropriate **vac.cfg** or **xlf.cfg** configuration file to see which option defaults are used:

Compiler	Configuration file
	`/etc/opt/ibmcmp/vac/bg/8.0/vac.cfg`
XL Fortran	`/etc/opt/ibmcmp/xlf/bg/10.1/xlf.cfg`

Different forms of the XL compiler invocation commands support various levels of the C, C++, and Fortran languages. These compiler invocation commands are summarized in Table 4 on page 2 and Table 5 on page 2.

Table 4. XL C/C++ cross-compiler invocations

Invocation	Funtionality
blrts_xlC blrts_xlc++	Source files are compiled as C++ language source code. If any of your source files are C++, you must use this invocation to link with the correct runtime libraries. Source files are compiled with **-qalias=ansi** set. Files with **.c** suffixes, assuming you have not used the **-+** or **-qsourcetype** compiler option.
blrts_xlc	Invokes the compiler for C source files. The following compiler options are implied with this invocation: • -qlanglvl=extc89 • -qalias=ansi • -qcpluscmt • -qkeyword=inline
blrts_cc	Invokes the compiler for C source files. The following compiler options are implied with this invocation: • -qlanglvl=extended • -qnoro • -qnoroconst
blrts_c99	Invokes the compiler for C source files, with support for ISO C99 language features. Full ISO C99 (ISO/IEC 9899:1999) conformance requires the presence of C99-compliant header files and runtime libraries. The following compiler options are implied with this invocation: • -qlanglvl=extc89 • -qalias=ansi • -qcpluscmt • -qkeyword=inline
blrts_c89	Invokes the compiler for C source files, with support for ISO C89 language features. The following options are implied with this invocation: • -qlanglvl=stdc89 • -qalias=ansi • -qstrict_induction • -qnolonglong • -D_ANSI_C_SOURCE • -D__STRICT_ANSI__ Use this invocation for strict conformance to the ANSI standard (ISO/IEC 9899:1990).

Table 5. XL Fortran cross-compiler invocations

Invocation	Funtionality
blrts_xlf blrts_f77 blrts_fort77	Makes programs conform more closely to the FORTRAN 77 standard.
blrts_xlf90 blrts_f90	Makes programs conform more closely to the Fortran 90 standard. For full conformance, compile with any of the following additional compiler options or suboptions: `-qnodirective -qnoescape -qextname -qfloat=nomaf:nofold -qnoswapomp -qlanglvl=90std`

Table 5. XL Fortran cross-compiler invocations (continued)

Invocation	Funtionality
blrts_xlf95 blrts_f95	Makes programs conform more closely to the Fortran 95 standard. For full conformance, compile with any of the following additional compiler options or suboptions: `-qnodirective -qnoescape -qextname -qfloat=nomaf:nofold -qnoswapomp -qlanglvl=95std`

Compiler option defaults

Compilations most commonly occur on the Front End Node. The resulting program can run on the Blue Gene/L system without manually copying the executable to the Service Node. See the “Running Applications” topic in section 5.1 of the *Blue Gene/L Application Development* document to learn how to run programs on Blue Gene/L.

The **blrts_*** compiler invocation commands set certain default compiler options to maximize the use of the Blue Gene architecture.

-qarch=[440d | 440]

Arguments

Specifies which instructions the compiler can generate. Suboptions include:

440
Generates code for the single floating-point unit (FPU) only.

440d
Generates parallel instructions for the 440d Double Hummer dual FPU. This is the default. Note that if you encounter problems with code generation, try resetting this option to **-qarch=440**.

Note: The Double Hummer FPU does not generate exceptions. The **-qflttrap** option, which traps floating-point exceptions, is disabled by default. If you enable the **-qflttrap** option, **-qarch** is automatically reset to **-qarch=440**.

-qbgl

Marks the object file as standalone to run on Blue Gene/L.

-qnoautoconfig

Prevents optimization levels **-O4** and **-O5** from resetting the **-qarch** setting to **auto**, thereby preserving the **-qarch** setting for the target architecture. This allows for cross-compilation to other architectures, such as Blue Gene.

-qtune=440

Optimizes code for the 440 family of processors. This is the default for **-qarch=440** and **-qarch=440d**.

Blue Gene specific options

-qfloat=[rngchk | norngchk]

At optimization level **-O3** and above, and without **-qstrict**, controls whether range checking is performed for input arguments for software divide and inlined square root operations.

Arguments

rngchk | norngchk
Specifying **-qfloat=norngchk** instructs the compiler to skip range checking, allowing for increased performance where division and square root operations are performed repeatedly within a loop. The **-qfloat=norngchk** option also generates Blue Gene-specific sequences for complex 1/x. However, if real(x)∧2 + imag(x)∧2 overflows, a NaN is returned.

Note that with **-qfloat=norngchk** in effect, the divisor of a division operation must not be 0.0, +/- INF, or denormalized values, and the input for sqrt must not be INF; otherwise incorrect results may be produced. For example, if the divisor for a division operation is 0.0 or a denormalized number (absolute value < 2∧(-1022) for double precision, and absolute value < 2∧(-126) for single precision), NaN, instead of INF, may result; when the divisor is +/- INF, NaN, instead of 0.0, may result. If the input is +INF for a sqrt operation, NaN, rather than INF, may result.

-qfloat=norngchk is only allowed when **-qnostrict** is in effect. If **-qstrict** is in effect, **-qfloat=norngchk** is ignored.

Unsupported options

The following compiler options are not supported by the Blue Gene/L hardware and should not be used:

-qsmp This option requires shared memory parallelism, which is not used by the Blue Gene/L.

-qpdf, -qshowpdf
The XL C/C++ and XL Fortran compilers do not fully support tuning optimizations through profile-directed feedback (PDF) for Blue Gene/L.

options specifying 64-bit mode
Blue Gene/L uses a 32-bit architecture. You cannot compile in 64-bit mode. **-q64**, **-qwarn64**, and all other options that apply to 64-bit mode are unsupported.

-qaltivec (C/C++)
The 440 processor does not support VMX instructions or vector data types.

-qenablevmx
The 440 processor does not support VMX instructions or vector data types.

-qpic This option controls the selection of TOC size for Position Independent Code. PIC code is used for shared/dynamic libraries, which are not supported on Blue Gene/L.

-qmkshrobj (C/C++)
This option creates a shared library object. Shared libraries are not supported on Blue Gene/L.

Chapter 2. Tuning your code for Blue Gene/L

The following sections describe strategies that you can use to best exploit the single-instruction-multiple-data (SIMD) capabilities of the Blue Gene/L 440d processor and the XL compilers' advanced instruction scheduling and register allocation algorithms.

Using the compiler optimization options

The **-O3** compiler option provides a high level of optimization and automatically sets other compiler options. For example, the **-qfloat=rsqrt** and **-qmaxmem=1** options are set by default with **-O3**, unless **-qstrict** is specified. Specifying **-O3** implies **-qhot=level=0**, unless you explicitly specify **-qhot** or **-qhot=level=1** option.

The **-qhot=simd** option enables SIMD vectorization of loops, and is enabled by default if you use **-O4**, **-O5**, or **-qhot** .

For more information on optimization options, see *Optimizing your applications* in the *XL C/C++ Advanced Edition V8.0 for Linux Programming Guide* and *Optimizing XL compiler applications* in the *XL Fortran Advanced Edition V10.1 for Linux Optimization and Programming Guide*.

Structuring data in adjacent pairs

The Blue Gene/L 440d processor's dual FPU includes special instructions for parallel computations. The compiler tries to pair adjacent single-precision or double-precision floating point values, to operate on them in parallel. Therefore, you can speed up computations by defining data objects that occupy adjacent memory blocks and are naturally aligned. These include arrays or structures of floating-point values and complex data types.

Whether you use an array, a structure, or a complex scalar, the compiler searches for sequential pairs of data for which it can generate parallel instructions. For example, the C code inFigure 1 on page 5 allows each pair of elements in a structure to be operated on in parallel.

```
struct quad {
   double a, b, c, d;
};

struct quad x, y, z;

void foo()
{
   z.a = x.a + y.a;
   z.b = x.b + y.b;  /* can load parallel (x.a,x.b), and (y.a, y.b),
                        do parallel add, and store parallel (z.a, z.b) */

   z.c = x.c + y.c;
   z.d = x.d + y.d;  /* can load parallel (x.c,x.d), and (y.c, y.d),
                        do parallel add, and store parallel (z.c, z.d) */
}
```

Figure 1. Adjacent paired data

The advantage of using complex types in arithmetic operations is that the compiler automatically uses parallel add, subtract, and multiply instructions when complex types appear as operands to addition, subtraction, and multiplication operators. Furthermore, the data that you provide does not actually need to represent complex numbers. In fact, both elements are represented internally as two real values. See Complex type manipulation functions for a description of the set of built-in functions available for Blue Gene/L. These functions are designed to efficiently manipulate complex-type data and include a function to convert non-complex data to complex types.

Using vectorizable basic blocks

The compiler schedules instructions most efficiently within *extended basic blocks*. These are code sequences which can contain conditional branches but have no entry points other than the first instruction. Specifically, minimize the use of branching instructions for the following:

- Handling of special cases, such as the generation of not-a-number (NaN) values
- C/C++ error handling that sets a value for **errno**.

  To explicitly inform the compiler that none of your code will set **errno**, compile with the **-qignerrno** compiler option (automatically set with **-O3**).
- C++ exception handlers

  To explicitly inform the compiler that none of your code will throw any exceptions, and therefore, that no exception-handling code needs to be generated, compile with the **-qnoeh** compiler option.

In addition, the optimal basic blocks remove dependencies between computations, so that the compiler sees each statement as entirely independent. You can construct a basic block as a series of independent statements or as a loop that repeatedly computes the same basic block with different arguments.

If you specify the **-qhot=simd** option, along with a minimum optimization level of **-O2**, the compiler can then vectorize these loops by applying various transformations, such as unrolling and software pipelining. See Removing possibilities for aliasing (C/C++), for additional strategies for removing data dependencies.

Using inline functions

An inline function is expanded in any context in which it is called. This avoids the normal performance overhead associated with the branching for a function call, and it allows functions to be included in basic blocks. The XL C/C++ and Fortran compilers provide several options for inlining.

The following options instruct the compiler to automatically inline all functions it deems appropriate:

- XL C/C++:
  - **-O** through **-O5**
  - **-qipa**
- XL Fortran:
  - **-O4** or **-O5**
  - **-qipa**

The following options allow you to select or name functions to be inlined:

- XL C/C++:

  - **-qinline**
  - **-Q**
- XL Fortran:
  - **-Q**

In C/C++, you can also use the standard **inline** function specifier or the **__attribute__(always_inline)** extension in your code to mark a function for inlining.

Important!: Do not overuse inlining because there are limits on how much inlining will be done. Mark only the most important functions.

For more information about the various compiler options for controlling function inlining, see the *XL Fortran Optimization and Programming* and *XL C/C++ Programming Guide*. For information on the different variations of the **inline** keyword supported by XL C/ C++, as well as the inlining function attribute extensions, see the *XL C/C++ Language Reference* .

Turning off range checking

At optimization levels of **-O3** and higher with **-qnostrict** in effect, the compiler may generate software division code instead of hardware floating-point divide instructions, and inlined code for square root operations. This may improve performance where division and square root operations are performed repeatedly within a loop.

To further improve performance, you can specify the **-qfloat=norngchk** option with **-qnostrict** to turn off range checking for input arguments. However, use this option with caution as it may produce undesirable results.

See "-qfloat=[rngchk | norngchk]" on page 4 for more information.

Removing possibilities for aliasing (C/C++)

When you use pointers to access array data in C/C++, the compiler cannot assume that the memory accessed by pointers will not be altered by other pointers that refer to the same address. For example, if two pointer input parameters share memory, the instruction to store the second parameter can overwrite the memory read from the first load instruction. This means that, after a store for a pointer variable, any load from a pointer must be reloaded. Consider the following code example:

```
int i = *p;
*q = 0;
j = *p;
```

If `*q` aliases `*p`, then the value must be reloaded from memory. If `*q` does not alias `*p`, the old value that is already loaded into `i` can be used.

To avoid the overhead of reloading values from memory every time they are referenced in the code, and allow the compiler to simply manipulate values that are already resident in registers, there are several strategies you can use. One approach is to assign input array element values to local variables and perform computations only on the local variables, as shown in Figure 2 on page 8.

```
#include <math.h>
void reciprocal_roots (const double* x, double* f)
{
      double x0 = x[0] ;
      double x1 = x[1] ;
      double r0 = 1.0/sqrt(x0) ;
      double r1 = 1.0/sqrt(x1) ;
      f[0] = r0 ;
      f[1] = r1 ;
}
```

Figure 2. Array parameters assigned to local variables

If you are certain that two references do not share the same memory address, another approach is to use the **#pragma disjoint** directive. This directive asserts that two identifiers do not share the same storage within the scope of their use. Specifically, you can use the pragma to inform the compiler that two pointer variables do not point to the same memory address. The directive in the following exampleFigure 3 indicates to the compiler that the pointers-to-arrays of double x and f do not share memory:

```
__inline void ten_reciprocal_roots (double* x, double* f)
{
#pragma disjoint (*x, *f)
int i;
    for (i=0; i < 10; i++)
  f[i]= 1.0 / sqrt (x[i]);
}
```

Figure 3. The #pragma disjoint directive

Important!: The correct functioning of this directive requires that the two pointers really be disjoint. If they are not, the compiled program will not run correctly.

Structuring computations in batches of five or ten

Floating-point operations are pipelined in the 440 processor so that one floating-point calculation is performed per cycle, with a latency of five cycles. Therefore, to keep the 440d processor's floating-point units busy, organize floating-point computations to perform step-wise operations in batches of five; that is, arrays of five elements and loops of five iterations. For the 440d, which has two FPUs, use batches of ten.

For example, with the 440d, at high optimization with **-qfloat=norngchk**, the function in Figure 4 should perform ten parallel reciprocal roots in about five cycles more than a single reciprocal root. This is because the compiler will perform two reciprocal roots in parallel and then use the empty cycles to run four more parallel reciprocal roots.

```
__inline void ten_reciprocal_roots (double* x, double* f)
{
#pragma disjoint (*x, *f)
    int i;
    for (i=0; i < 10; i++)
 f[i]= 1.0 / sqrt (x[i]);
}
```

Figure 4. A function to calculate reciprocal roots for arrays of ten elements

The definition in Figure 5 shows wrapping of the inlined, optimized `ten_reciprocal_roots` function inside a function that allows you to pass in arrays of any number of elements. This function then passes the values in batches of ten to the `ten_reciprocal_roots` function, and calculates the remaining operations individually.

```
static void unaligned_reciprocal_roots (double* x, double* f, int n)
{
#pragma disjoint (*x, *f)
    while (n >= 10) {
 ten_reciprocal_roots (x, f);
 x += 10;
 f += 10;
    }
    /* remainder */
    while (n > 0) {
 *f = 1.0 / sqrt (*x);
 f++, x++;
    }
}
```

Figure 5. A function to pass values in batches of ten

Checking for data alignment

The Blue Gene/L architecture allows for two double-precision values to be loaded in parallel in a single cycle, provided that the load address is aligned such that the values loaded do not cross a cache-line boundary (which is 32-bytes). If they cross this boundary, the hardware generates an alignment trap. This trap may cause the program to crash or result in a severe performance penalty to be fixed at run time by the kernel.

The compiler does not generate these parallel load and store instructions unless it is sure that it is safe to do so. For non-pointer local and global variables, the compiler knows when this is safe. To allow the compiler to generate these parallel loads and stores for accesses through pointers, include code that tests for correct alignment and gives the compiler hints.

To test for alignment, first create one version of a function which asserts the alignment of an input variable at that point in the program flow. You can use the C/C++ **__alignx** builtin function or the Fortran **ALIGNX** function to inform the compiler that the incoming data is correctly aligned according to a specific byte boundary, so it can efficiently generate loads and stores.

The function takes two arguments. The first argument is an integer constant expressing the number of alignment bytes (must be a positive power of two). The second argument is the variable name, typically a pointer to a memory address.

The C/C++ prototype for the function is:

```
extern
#ifdef __cplusplus
"builtin"
#endif
void __alignx (int n, const void *addr)
```

Here *n* is the number of bytes. For example, `__align(16, y)` specifies that the address `y` is 16-byte aligned.

In Fortran, the built-in subroutine is **ALIGNX(K,M)** , where *K* is of type INTEGER(4), and *M* is a variable of any type. When *M* is an integer pointer, the argument refers to the address of the pointee.

Figure 6 asserts that the variables x and f are aligned along 16-byte boundaries.

```
#include <math.h>
__inline void aligned_ten_reciprocal_roots (double* x, double* f)
{
#pragma disjoint (*x, *f)
int i;
    __alignx (16, x);
    __alignx (16, f);
    for (i=0; i < 10; i++)
 f[i]= 1.0 / sqrt (x[i]);
}
```

Figure 6. Use of the __alignx built-in function

Note: The **__alignx** function does not perform any alignment. It merely informs the compiler that the variables are aligned as specified, at the point where the **__alignx** call is placed. If the variables are not aligned correctly, the program can end abruptly or run very slowly.

After you create a function to handle input variables that are correctly aligned, you can then create a function that tests for alignment and then calls the appropriate function to perform the calculations. The function inFigure 7 checks to see whether the incoming values are correctly aligned. Then it calls the aligned (Example 1-6) or unaligned (Example 1-4) version of the function according to the result.

```
void reciprocal_roots (double *x, double *f, int n)
{
    /* are both x & f 16 byte aligned? */
    if ( ((((int) x) | ((int) f)) & 0xf) == 0) /*    This could also be done as:
           if (((int) x % 16 == 0) && ((int) f % 16) == 0) */
 aligned_ten_reciprocal_roots (x, f, n);
 else
 ten_reciprocal_roots (x, f, n);
}
```

Figure 7. A function to test for alignment

The alignment test inFigure 7 provides an optimized method of testing for 16-byte alignment by performing a bit-wise OR on the two incoming addresses and testing whether the lowest four bits are 0 (that is, 16-byte aligned).

Chapter 3. Using the high performance libraries

XL C/C++ and XL Fortran Advanced Edition for Blue Gene are shipped with a set of libraries for high-performance mathematical computing:

- The Mathematical Acceleration Subsystem (MASS) is a set of libraries of tuned mathematical intrinsic functions that provide improved performance over the corresponding standard system math library functions. MASS is described in "Using the Mathematical Acceleration Subsystem (MASS)."

Note: In this discussion, rather than repeating the phrase *XL C/C++ function(s) or XL Fortran routine(s)*, the term *function(s)* is used to denote a XL C/C++ function(s) or XL Fortran routine(s) as it applies within its respective contexts.

Using the Mathematical Acceleration Subsystem (MASS)

The MASS libraries consist of a library of scalar XL C/C++ functions or XL Fortran routines, described in "Using the scalar library"; and a set of vector libraries tuned for specific architectures, described in "Using the vector libraries" on page 13. The functions contained in both scalar and vector libraries are automatically called at certain levels of optimization, but you can also call them explicitly in your programs. Note that the accuracy and exception handling might not be identical in MASS functions and system library functions.

"Compiling and linking a program with MASS" on page 20 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in concert with the regular system library scalar functions.

Using the scalar library

The MASS scalar library, `libmass.a`, contains an accelerated set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions. When you compile programs with any of the following options:

- **XL C/C++**: **-qhot -qignerrno -qnostrict**
- **XL Fortran**: **-qhot -qnostrict**
- **-qhot -O3**
- **-O4**
- **-O5**

the compiler automatically uses the faster MASS functions for all math library functions (with the exception of `atan2`, `dnint`, `sqrt`, `rsqrt`). In fact, the compiler first tries to "vectorize" calls to math library functions by replacing them with the equivalent MASS vector functions; if it cannot do so, it uses the MASS scalar functions. When the compiler performs this automatic replacement of math library functions, it uses versions of the MASS functions contained in the system library `libxlopt.a`; you do not need to add any special calls to the MASS functions in your code, or to link to the `libxlopt` library.

If you are not using any of the optimization options listed above, and/or want to explicitly call the MASS scalar functions, you can do so as follows:

For XL C/C++

1. Provide the prototypes for the functions (except `dnint`, `cosisin`, `rsqrt`, and `sincos`), by including `math.h` in your source files.
2. Provide the prototypes for `dnint`, `cosisin`, `rsqrt` and `sincos`, by including `mass.h` in your source files.
3. Link the MASS scalar library `libmass.a` with your application. For instructions, see "Compiling and linking a program with MASS" on page 20.

For XL Fortran

1. Link the MASS scalar library `libmass.a` with your application. For instructions, see "Compiling and linking a program with MASS" on page 20
2. All the MASS scalar routines except `cosisin`, `expm1`, `log1p`, `rsqrt`, and `sincos` are recognized by XL Fortran as intrinsic functions, so no explicit interface block is needed. To provide an interface block for `cosisin`, `expm1`, `log1p`, `rsqrt`, and `sincos`, include `mass.include` in your source file.

The MASS scalar functions accept double-precision parameters and return a double-precision result, except `sincos` which gives 2 double-precision results. They are summarized in Table 6.

Table 6. MASS scalar functions

Function	Description	Prototype
sqrt	Returns the square root of `x`	double sqrt (double x);
rsqrt	Returns the reciprocal of the square root of `x`	double rsqrt (double x);
exp	Returns the exponential function of `x`	double exp (double x);
expm1	Returns (the exponential function of `x`) – 1	double expm1 (double x);
log	Returns the natural logarithm of `x`	double log (double x);
log1p	Returns the natural logarithm of (`x` + 1)	double log1p (double x);
sin	Returns the sine of `x`	double sin (double x);
sincos	Sets `*s` to the sine of `x` and `*c` to the cosine of x.	void sincos (double x, double* s, double* c);
cos	Returns the cosine of `x`	double cos (double x);
cosisin	Returns a complex number with the real part the cosine of `x` and the imaginary part the sine of `x`.	double_Complex cosisin (double);
tan	Returns the tangent of `x`	double tan (double x);
atan	Returns the arc tangent of `x`	double atan (double x);
atan2	Returns the arc tangent of `x/y`	double atan2 (double x, double y);
sinh	Returns the hyperbolic sine of `x`	double sinh (double x);
cosh	Returns the hyperbolic cosine of `x`	double cosh (double x);
tanh	Returns the hyperbolic tangent of `x`	double tanh (double x);
dnint	Returns the nearest integer to `x` (as a double)	double dnint (double x);

Table 6. MASS scalar functions (continued)

Function	Description	Prototype
pow (C/C++)	Returns x raised to the power y	double pow (double x, double y);
x**y (XL Fortran)	Returns x raised to the power y	N/A

The following example shows the XL Fortran interface declaration for the `rsqrt` scalar function:

```
interface

real*8 function rsqrt (%val(x))
  real*8 x      ! Returns the reciprocal of the square root of x.
end function rsqrt

end interface
```

The trigonometric functions (`sin`, `cos`, `tan`) return NaN (Not-a-Number) for large arguments (abs(x)>2**50*pi).

Note: In some cases the MASS functions are not as accurate as the `libm.a` library, and they might handle edge cases differently (`sqrt(Inf)`, for example).

Using the vector libraries

When you compile programs with any of the following options:

- **XL C/C++**: **-qhot -qignerrno -qnostrict**
- **XL Fortran**: **-qhot -qnostrict**
- **-qhot -O3**
- **-O4**
- **-O5**

the compiler automatically attempts to vectorize calls to system math functions by calling the equivalent MASS vector functions (with the exceptions of functions `vatan2` (XLF), `vsatan2`(XLF), `vdnint`, `vdint`, `vsincos`, `vssincos`, `vcosisin`, `vscosisin`, `vqdrt`, `vsqdrt`, `vrqdrt`, `vsrqdrt`, `vpopcnt4`, and `vpopcnt8`). For automatic vectorization, the compiler uses versions of the MASS functions contained in the system library `libxlopt.a`; you do not need to add any special calls to the MASS functions in your code, or to link to the `libxlopt` library.

If you are not using any of the optimization options listed above, and/or want to explicitly call any of the MASS vector functions, you can do so by including the XL C/C++ header `massv.h` or the XL Fortran `massv.include` file in your source files and linking your application with the `libmassv.a` vector library. (Information on linking is provided in "Compiling and linking a program with MASS" on page 20.)

The single-precision and double-precision floating-point functions contained in the vector libraries are summarized in Table 7 on page 14 (XL C/C++) and Table 8 on page 17(XL Fortran). The integer functions contained in the vector libraries are summarized in Table 9 on page 18. Note that in C and C++ applications, only call by reference is supported, even for scalar arguments.

With the exception of a few functions (described below), all of the floating-point functions in the vector libraries accept three parameters:

- a double-precision (for double-precision functions) or single-precision (for single-precision functions) vector output parameter
- a double-precision (for double-precision functions) or single-precision (for single-precision functions) vector input parameter
- an integer vector-length parameter

The functions are of the form

```
function_name (y,x,n)
```

where *y* is the target vector, *x* is the source vector, and *n* is the vector length. The parameters *y* and *x* are assumed to be double-precision for functions with the prefix `v`, and single-precision for functions with the prefix `vs`. As examples, the following XL C/C++ code:

```
#include <massv.h>

double x[500], y[500];
int n;
n = 500;
...
vexp (y, x, &n);
```

outputs a vector *y* of length 500 whose elements are exp(x[i]), where i=0,...,499.

The following XL Fortran code:

```
include 'massv.include'

real*8 x(500), y(500)
integer n
n = 500
...
call vexp (y, x, n)
```

outputs a vector *y* of length 500 whose elements are exp(x(i)), where i=1,...,500.

The functions `vdiv`, `vsincos`, `vpow`, and `vatan2` (and their single-precision versions, `vsdiv`, `vssincos`, `vspow`, and `vsatan2`) take four parameters. The functions `vdiv`, `vpow`, and `vatan2` take the parameters (*z*,*x*,*y*,*n*). The function `vdiv` outputs a vector *z* whose elements are x[i]/y[i], where i=0,..,*n–1. The function `vpow` outputs a vector *z* whose elements are $x[i]^{y[i]}$, where i=0,..,*n–1. The function `vatan2` outputs a vector *z* whose elements are atan(x[i]/y[i]), where i=0,..,*n–1. The function `vsincos` takes the parameters (*y*,*z*,*x*,*n*), and outputs two vectors, *y* and *z*, whose elements are sin(x[i]) and cos(x[i]), respectively.

In `vcosisin(y,x,n)` and `vscosisin(y,x,n)`, *x* is a vector of *n* elements and the function outputs a vector *y* of *n* `complex` elements of the form (cos(x[i]),sin(x[i])).

Table 7. MASS floating-point vector functions (XL C/C++)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vacos	vsacos	Sets `y[i]` to the arc cosine of `x[i]`, for i=0,..,*n–1	void vacos (double y[], double x[], int *n);	void vsacos (float y[], float x[], int *n);
vacosh	vsacosh	Sets `y[i]` to the hyperbolic arc cosine of `x[i]`, for i=0,..,*n–1	void vacosh (double y[], double x[], int *n);	void vsacosh (float y[], float x[], int *n);

Table 7. MASS floating-point vector functions (XL C/C++) (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vasin	vsasin	Sets `y[i]` to the arc sine of `x[i]`, for i=0,..,*n–1	void vasin (double y[], double x[], int *n);	void vsasin (float y[], float x[], int *n);
vasinh	vsasinh	Sets `y[i]` to the hyperbolic arc sine of `x[i]`, for i=0,..,*n–1	void vasinh (double y[], double x[], int *n);	void vsasinh (float y[], float x[], int *n);
vatan2	vsatan2	Sets `z[i]` to the arc tangent of `x[i]/y[i]`, for i=0,..,*n–1	void vatan2 (double z[], double x[], double y[], int *n);	void vsatan2 (float z[], float x[], float y[], int *n);
vatanh	vsatanh	Sets `y[i]` to the hyperbolic arc tangent of `x[i]`, for i=0,..,*n–1	void vatanh (double y[], double x[], int *n);	void vsatanh (float y[], float x[], int *n);
vcbrt	vscbrt	Sets `y[i]` to the cube root of `x[i]`, for i=0,..,*n-1	void vcbrt (double y[], double x[], int *n);	void vscbrt (float y[], float x[], int *n);
vcos	vscos	Sets `y[i]` to the cosine of `x[i]`, for i=0,..,*n–1	void vcos (double y[], double x[], int *n);	void vscos (float y[], float x[], int *n);
vcosh	vscosh	Sets `y[i]` to the hyperbolic cosine of `x[i]`, for i=0,..,*n–1	void vcosh (double y[], double x[], int *n);	void vscosh (float y[], float x[], int *n);
vcosisin	vscosisin	Sets the real part of `y[i]` to the cosine of `x[i]` and the imaginary part of `y[i]` to the sine of `x[i]`, for i=0,..,*n–1	void vcosisin (double _Complex y[], double x[], int *n);	void vscosisin (float _Complex y[], float x[], int *n);
vdint		Sets `y[i]` to the integer truncation of `x[i]`, for i=0,..,*n–1	void vdint (double y[], double x[], int *n);	
vdiv	vsdiv	Sets `z[i]` to `x[i]/y[i]`, for i=0,..,*n–1	void vdiv (double z[], double x[], double y[], int *n);	void vsdiv (float z[], float x[], float y[], int *n);
vdnint		Sets `y[i]` to the nearest integer to `x[i]`, for i=0,..,n–1	void vdnint (double y[], double x[], int *n);	
vexp	vsexp	Sets `y[i]` to the exponential function of `x[i]`, for i=0,..,*n–1	void vexp (double y[], double x[], int *n);	void vsexp (float y[], float x[], int *n);
vexpm1	vsexpm1	Sets `y[i]` to (the exponential function of `x[i]`)-1, for i=0,..,*n–1	void vexpm1 (double y[], double x[], int *n);	void vsexpm1 (float y[], float x[], int *n);

Table 7. MASS floating-point vector functions (XL C/C++) (continued)

Double-precision function	**Single-precision function**	**Description**	**Double-precision function prototype**	**Single-precision function prototype**
vlog	vslog	Sets `y[i]` to the natural logarithm of `x[i]`, for i=0,..,*n–1	void vlog (double y[], double x[], int *n);	void vslog (float y[], float x[], int *n);
vlog10	vslog10	Sets `y[i]` to the base-10 logarithm of `x[i]`, for i=0,..,*n–1	void vlog10 (double y[], double x[], int *n);	void vslog10 (float y[], float x[], int *n);
vlog1p	vslog1p	Sets `y[i]` to the natural logarithm of (`x[i]`+1), for i=0,..,*n–1	void vlog1p (double y[], double x[], int *n);	void vslog1p (float y[], float x[], int *n);
vpow	vspow	Sets `z[i]` to `x[i]` raised to the power `y[i]`, for i=0,..,*n-1	void vpow (double z[], double x[], double y[], int *n);	void vspow (float z[], float x[], float y[], int *n);
vqdrt	vsqdrt	Sets `y[i]` to the fourth root of `x[i]`, for i=0,..,*n-1	void vqdrt (double y[], double x[], int *n);	void vsqdrt (float y[], float x[], int *n);
vrcbrt	vsrcbrt	Sets `y[i]` to the reciprocal of the cube root of `x[i]`, for i=0,..,*n-1	void vrcbrt (double y[], double x[], int *n);	void vsrcbrt (float y[], float x[], int *n);
vrec	vsrec	Sets `y[i]` to the reciprocal of `x[i]`, for i=0,..,*n–1	void vrec (double y[], double x[], int *n);	void vsrec (float y[], float x[], int *n);
vrqdrt	vsrqdrt	Sets `y[i]` to the reciprocal of the fourth root of `x[i]`, for i=0,..,*n-1	void vrqdrt (double y[], double x[], int *n);	void vsrqdrt (float y[], float x[], int *n);
vrsqrt	vsrsqrt	Sets `y[i]` to the reciprocal of the square root of `x[i]`, for i=0,..,*n–1	void vrsqrt (double y[], double x[], int *n);	void vsrsqrt (float y[], float x[], int *n);
vsin	vssin	Sets `y[i]` to the sine of `x[i]`, for i=0,..,*n–1	void vsin (double y[], double x[], int *n);	void vssin (float y[], float x[], int *n);
vsincos	vssincos	Sets `y[i]` to the sine of `x[i]` and `z[i]` to the cosine of `x[i]`, for i=0,..,*n–1	void vsincos (double y[], double z[], double x[], int *n);	void vssincos (float y[], float z[], float x[], int *n);
vsinh	vssinh	Sets `y[i]` to the hyperbolic sine of `x[i]`, for i=0,..,*n–1	void vsinh (double y[], double x[], int *n);	void vssinh (float y[], float x[], int *n);
vsqrt	vssqrt	Sets `y[i]` to the square root of `x[i]`, for i=0,..,*n–1	void vsqrt (double y[], double x[], int *n);	void vssqrt (float y[], float x[], int *n);
vtan	vstan	Sets `y[i]` to the tangent of `x[i]`, for i=0,..,*n-1	void vtan (double y[], double x[], int *n);	void vstan (float y[], float x[], int *n);

Table 7. MASS floating-point vector functions (XL C/C++) (continued)

Double-precision function	**Single-precision function**	**Description**	**Double-precision function prototype**	**Single-precision function prototype**
vtanh	vstanh	Sets `y[i]` to the hyperbolic tangent of `x[i]`, for i=0,..,*n–1	void vtanh (double y[], double x[], int *n);	void vstanh (float y[], float x[], int *n);

Table 8

Table 8. MASS floating-point vector library functions (XL Fortran)

Double-precision function	Single-precision function	Arguments	Description
vacos	vsacos	`(y,x,n)`	Sets y(i) to the arc cosine of x(i), for i=1,..,n
vacosh	vsacosh	`(y,x,n)`	Sets y(i) to the hyperbolic arc cosine of x(i), for i=1,..,n
vasin	vsasin	`(y,x,n)`	Sets y(i) to the arc sine of x(i), for i=1,..,n
vasinh	vsasinh	`(y,x,n)`	Sets y(i) to the arc hyperbolic sine of x(i), for i=1,..,n
vatan2	vsatan2	`(z,x,y,n)`	Sets z(i) to the arc tangent of x(i)/y(i), for i=1,..,n
vatanh	vsatanh	`(y,x,n)`	Sets y(i) to the arc hyperbolic tangent of x(i), for i=1,..,n
vcbrt	vscbrt	`(y,x,n)`	Sets y(i) to the cube root of x(i), for i=1,..,n
vcos	vscos	`(y,x,n)`	Sets y(i) to the cosine of x(i), for i=1,..,n
vcosh	vscosh	`(y,x,n)`	Sets y(i) to the hyperbolic cosine of x(i), for i=1,..,n
vcosisin	vscosisin	`(y,x,n)`	Sets the real part of y(i) to the cosine of x(i) and the imaginary part of y(i) to the sine of x(i), for i=1,..,n
vdint		`(y,x,n)`	Sets y(i) to the integer truncation of x(i), for i=1,..,n
vdiv	vsdiv	`(z,x,y,n)`	Sets z(i) to x(i)/y(i), for i=1,..,n
vdnint		`(y,x,n)`	Sets y(i) to the nearest integer to x(i), for i=1,..,n
vexp	vsexp	`(y,x,n)`	Sets y(i) to the exponential function of x(i), for i=1,..,n
vexpm1	vsexpm1	`(y,x,n)`	Sets y(i) to (the exponential function of x(i))-1, for i=1,..,n
vlog	vslog	`(y,x,n)`	Sets y(i) to the natural logarithm of x(i), for i=1,..,n
vlog10	vslog10	`(y,x,n)`	Sets y(i) to the base-10 logarithm of x(i), for i=1,..,n
vlog1p	vslog1p	`(y,x,n)`	Sets y(i) to the natural logarithm of (x(i)+1), for i=1,..,n

Table 8. MASS floating-point vector library functions (XL Fortran) (continued)

vpow	vspow	`(z,x,y,n)`	Sets z(i) to x(i) raised to the power y(i), for i=1,..,n
vqdrt	vsqdrt	`(y,x,n)`	Sets y(i) to the 4th root of x(i), for i=1,..,n
vrcbrt	vsrcbrt	`(y,x,n)`	Sets y(i) to the reciprocal of the cube root of x(i), for i=1,..,n
vrec	vsrec	`(y,x,n)`	Sets y(i) to the reciprocal of x(i), for i=1,..,n
vrqdrt	vsrqdrt	`(y,x,n)`	Sets y(i) to the reciprocal of the 4th root of x(i), for i=1,..,n
vrsqrt	vsrsqrt	`(y,x,n)`	Sets y(i) to the reciprocal of the square root of x(i), for i=1,..,n
vsin	vssin	`(y,x,n)`	Sets y(i) to the sine of x(i), for i=1,..,n
vsincos	vssincos	`(y,z,x,n)`	Sets y(i) to the sine of x(i) and z(i) to the cosine of x(i), for i=1,..,n
vsinh	vssinh	`(y,x,n)`	Sets y(i) to the hyperbolic sine of x(i), for i=1,..,n
vsqrt	vssqrt	`(y,x,n)`	Sets y(i) to the square root of x(i), for i=1,..,n
vtan	vstan	`(y,x,n)`	Sets y(i) to the tangent of x(i), for i=1,..,n
vtanh	vstanh	`(y,x,n)`	Sets y(i) to the hyperbolic tangent of x(i), for i=1,..,n

For XL C/C++, the integer functions are of the form *function_name* (*x*[], **n*), where `x[]` is a vector of 4-byte (for `vpopcnt4`) or 8-byte (for `vpopcnt8`) numeric objects (integral or floating-point), and `*n` is the vector length.

Table 9. MASS integer vector library functions (XL C/C++)

Function	Description	Prototype
vpopcnt4	Returns the total number of 1 bits in the concatenation of the binary representation of `x[i]`, for i=0,..,*n–1 , where `x` is a vector of 32-bit objects.	unsigned int vpopcnt4 (void *x, int *n)
vpopcnt8	Returns the total number of 1 bits in the concatenation of the binary representation of `x[i]`, for i=0,..,*n–1 , where `x` is a vector of 64-bit objects.	unsigned int vpopcnt8 (void *x, int *n)

For **XL Fortran**, the integer routines are of the form *function_name* (*x*, *n*), where *x* is a vector of 4-byte (for `vpopcnt4`) or 8-byte (for `vpopcnt8`) numeric objects (integer or floating-point), and *n* is the vector length. The vector integer routines are summarized in Table 9.

Table 10. MASS integer vector library functions (XL Fortran)

Function	Description	Interface
vpopcnt4	Returns the total number of 1 bits in the concatenation of the binary representation of x(i), for i=1,...,n, where x is vector of 32-bit objects	`integer*4 function vpopcnt4 (x, n) integer*4 x(*), n`
vpopcnt8	Returns the total number of 1 bits in the concatenation of the binary representation of x(i), for i=1,...,n, where x is vector of 64-bit objects	`integer*4 function vpopcnt8 (x, n) integer*8 x(*)`

The following example shows XL Fortran interface declarations for some of the MASS double-precision vector routines:

```
interface

subroutine vsqrt (y, x, n)
  real*8 y(*), x(*)
  integer n       ! Sets y(i) to the square root of x(i), for i=1,..,n
end subroutine vsqrt

subroutine vrsqrt (y, x, n)
  real*8 y(*), x(*)
  integer n       ! Sets y(i) to the reciprocal of the square root of x(i),
                  ! for i=1,..,n
end subroutine vrsqrt

end interface
```

The following example shows XL Fortran interface declarations for some of the MASS single-precision vector routines:

```
interface

subroutine vssqrt (y, x, n)
  real*4 y(*), x(*)
  integer n      ! Sets y(i) to the square root of x(i), for i=1,..,n
end subroutine vssqrt

subroutine vsrsqrt (y, x, n)
  real*4 y(*), x(*)
  integer n      ! Sets y(i) to the reciprocal of the square root of x(i),
                  ! for i=1,..,n
end subroutine vsrsqrt

end interface
```

Overlap of input and output vectors

In most applications, the MASS vector functions are called with disjoint input and output vectors; that is, the two vectors do not overlap in memory. Another common usage scenario is to call them with the same vector for both input and output parameters (for example, `vsin (y, y, &n)`). Other kinds of overlap (where input and output vectors are neither disjoint nor identical) should be avoided, since they may produce unexpected results:

- For calls to vector functions that take one input and one output vector (for example, `vsin (y, x, &n)`):

  The vectors `x[0:n-1]` and `y[0:n-1]` must be either disjoint or identical, or unexpected results may be obtained.
- For calls to vector functions that take two input vectors (for example, `vatan2 (y, x1, x2, &n)`):

The previous restriction applies to both pairs of vectors `y`, `x1` and `y`, `x2`. That is, `y[0:n-1]` and `x1[0:n-1]` must be either disjoint or identical; and `y[0:n-1]` and `x2[0:n-1]` must be either disjoint or identical.

- For calls to vector functions that take two output vectors (for example, `vsincos (y1, y2, x, &n)`):

  The above restriction applies to both pairs of vectors `y1`, `x` and `y2`, `x`. That is, `y1[0:n-1]` and `x[0:n-1]` must be either disjoint or identical; and `y2[0:n-1]` and `x[0:n-1]` must be either disjoint or identical. Also, the vectors `y1[0:n-1]` and `y2[0:n-1]` must be disjoint.

Consistency of MASS vector functions

All the functions in the MASS vector libraries are consistent, in the sense that a given input value will always produce the same result, regardless of its position in the vector, and regardless of the vector length.

Compiling and linking a program with MASS

To compile an application that calls the functions in the MASS libraries, specify **mass** and **massv** on the **-l** linker option.

For example, if the MASS libraries are installed in the default directory, you could specify:

```
blrts_xlc progc.c -o progc -lmass -lmassv

blrts_xlf progf.f -o progf -lmass -lmassv
```

The MASS functions must run in the round-to-nearest rounding mode and with floating-point exception trapping disabled. (These are the default compilation settings.)

Using libmass.a with the math system library

If you wish to use the `libmass.a` scalar library for some functions and the normal math library `libm.a` for other functions, follow this procedure to compile and link your program:

1. Use the **ar** command to extract the object files of the desired functions from `libmass.a`. For most functions, the object file name is the function name followed by `.s32.o`.[1] For example, to extract the object file for the `tan` function, the command would be:

   ```
   ar -x tan.s32.o libmass.a
   ```

2. Archive the extracted object files into another library:

   ```
    ar -qv libfasttan.a tan.s32.o
    ranlib libfasttan.a
   ```

3. Create the final executable using **blrts_xlc**, specifying **-lfasttan** instead of **-lmass**:

   ```
   blrts_xlc sample.c -o sample dir_containing_libfasttan.a -lfasttan
   ```

   This links only the `tan` function from MASS (now in `libfasttan.a`) and the remainder of the math functions from the standard system library.

Exceptions:

1. The `sin` and `cos` functions are both contained in the object file `sincos.s32.o`. The `cosisin` and `sincos` functions are both contained in the object file `cosisin.s32.o`.

Chapter 4. Using XL builtin floating-point functions for Blue Gene/L

The XL C/C++ and XL Fortran compilers include a set of built-in functions that are optimized for the PowerPC architecture. For a full description of them, refer to the following documents (available from the Web pages listed at the beginning of this chapter):

- *Built-in functions for POWER and PowerPC architectures* in *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*
- *Intrinsic procedures* in *XL Fortran Advanced Edition V10.1 for Linux Language Reference*

In addition, on Blue Gene/L, the XL compilers provide a set of built-in functions that are specifically optimized for the PowerPC 440d's Double Hummer dual FPU. These built-in functions provide an almost one-to-one correspondence with the Double Hummer instruction set.

All of the C/C++ and Fortran built-in functions operate on complex data types, which have an underlying representation of a two-element array, in which the real part represents the *primary* element and the imaginary part represents the *second* element. The input data you provide does not actually need to represent complex numbers: in fact, both elements are represented internally as two real values, and none of the built-in functions actually performs complex arithmetic. A set of built-in functions especially designed to efficiently manipulate complex-type variables is also available.

The Blue Gene/L built-in functions perform the several types of operations as explained in the following paragraphs.

Parallel operations perform SIMD computations on the primary and secondary elements of one or more input operands. They store the results in the corresponding elements of the output. As an example, Figure 8 on page 22 illustrates how a parallel multiply operation is performed.

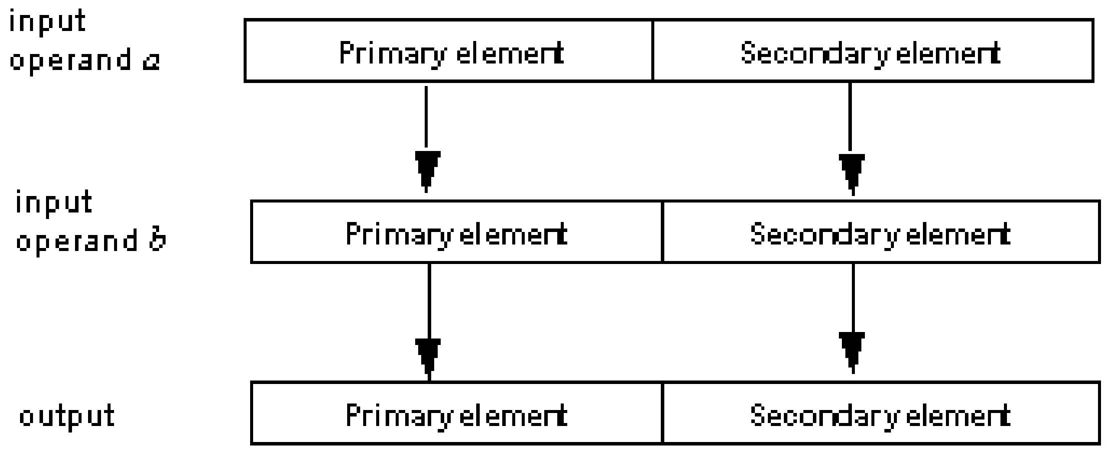

Figure 8. Parallel operations

Cross operations perform SIMD computations on the opposite primary and secondary elements of one or more input operands. They store the results in the corresponding elements in the output. As an example, Figure 9 illustrates how a cross-multiply operation is performed.

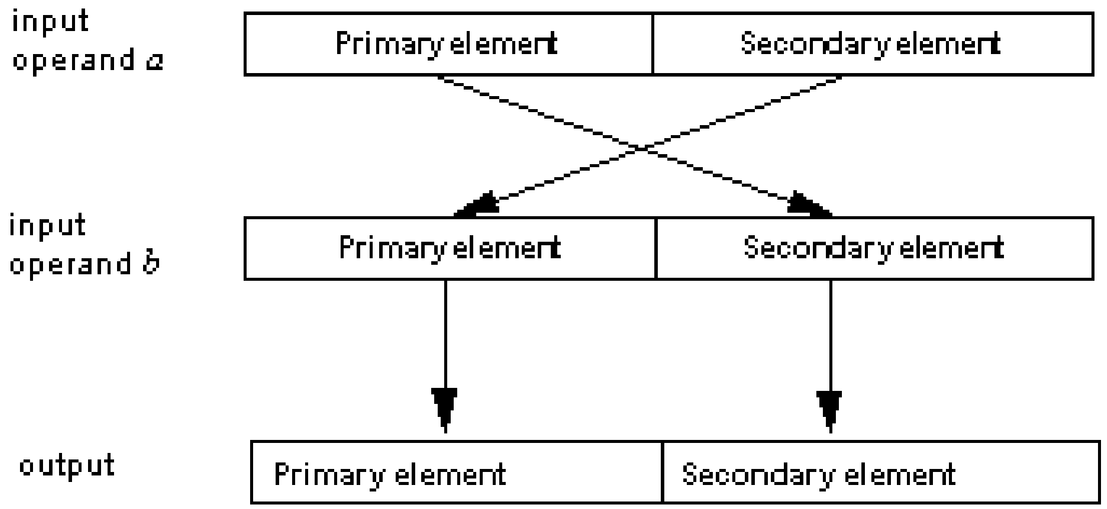

Figure 9. Cross operations

Copy-primary operations perform SIMD computation between the corresponding primary and secondary elements of two input operands, where the primary element of the first operand is replicated to the secondary element. As an example, Figure 10 on page 23 illustrates how a cross-primary multiply operation is performed.

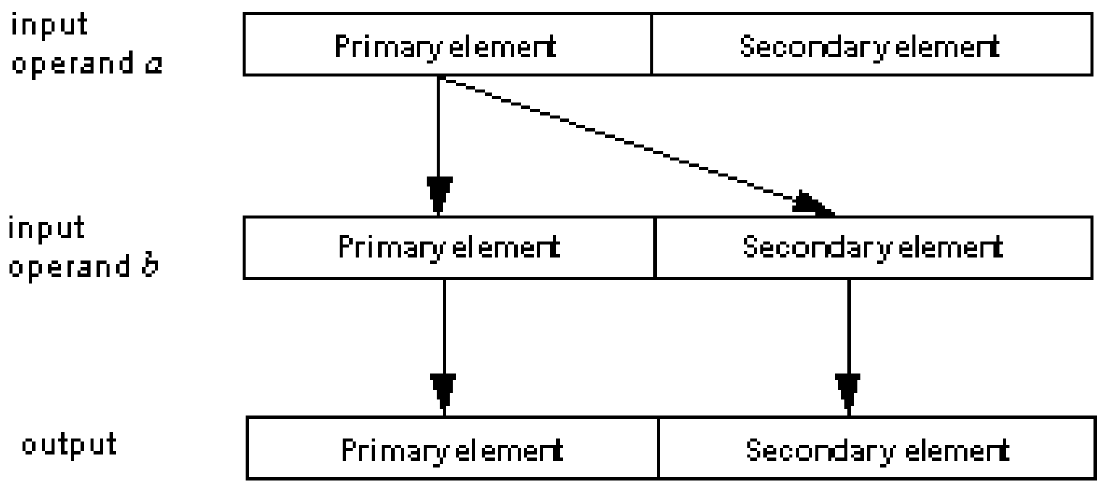

Figure 10. Copy-primary operations

Copy-secondary operations perform SIMD computation between the corresponding primary and secondary elements of two input operands, where the secondary element of the first operand is replicated to the primary element. As an example, Figure 11 illustrates how a cross-secondary multiply operation is performed.

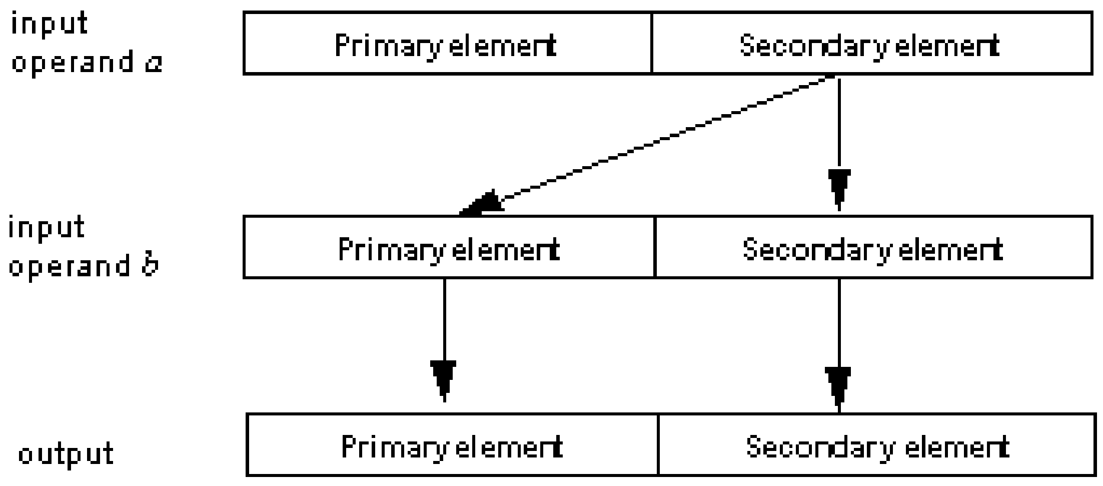

Figure 11. Copy-secondary operations

In *cross-copy* operations, the compiler crosses either the primary or secondary element of the first operand, so that copy-primary and copy-secondary operations can be used interchangeably to achieve the same result. The operation is performed on the total value of the first operand. As an example, Figure 12 on page 24 illustrates the result of a cross-copy multiply operation.

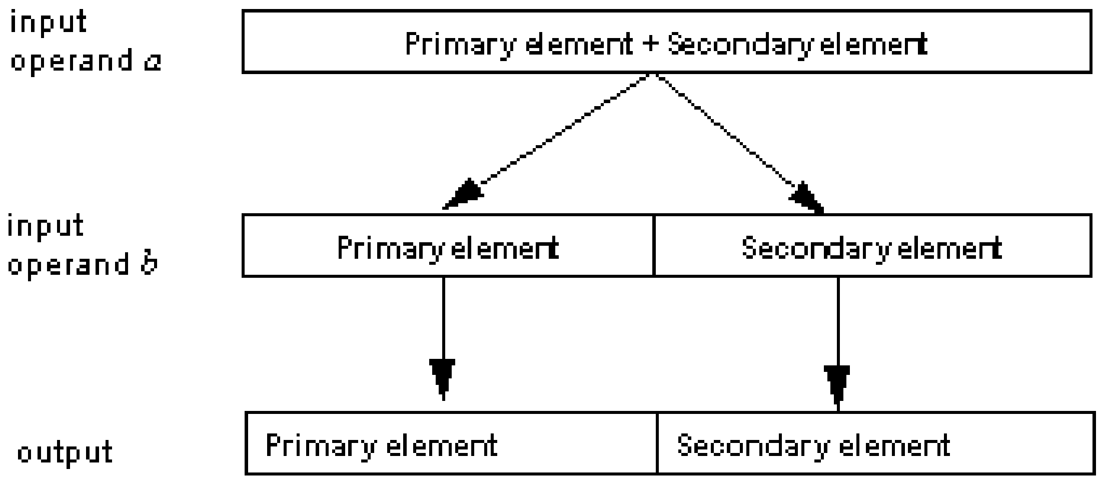

Figure 12. Cross-copy operations

The following sections describe the available built-in functions by category:

- Complex type manipulation functions
- Load and store functions
- Move functions
- Arithmetic functions
- Select functions

For each function, the C/C++ prototype is provided. In C, you do not need to include a header file to obtain the prototypes. The compiler includes them automatically. In C++, you need to include the header file `builtins.h`.

Fortran does not use prototypes for built-in functions. Therefore, the interfaces for the Fortran functions are provided in textual form. The function names omit the double underscore (__) in Fortran.

All of the built-in functions, with the exception of the complex type manipulation functions, require compilation under **-qarch=440d** . This is the default setting on Blue Gene/L.

To help clarify the English description of each function, the following notation is used:

element (*variable*)

where *element* represents one of *primary* or *secondary* , and *variable* represents input variable *a* , *b* , or *c* , and the output variable *result* . For example, consider the following formula:

```
primary(result) = primary(a) + primary(b)
```

The formula indicates that the primary element of input variable *a* is added to the primary element of input variable *b* and stored in the primary element of the *result*.

To optimize your calls to the Blue Gene/L built-in functions, follow the guidelines provided in Tuning your code for Blue Gene/L. Using the **alignx** built-in function (described in Checking for data alignment), and specifying the **disjoint** pragma (described in Removing possibilities for aliasing (C/C++)), are recommended for code that calls any of the built-in functions.

Complex type manipulation functions

The functions described in this section are useful for efficiently manipulating complex data types, by allowing you to automatically convert real floating-point data to complex types, and to extract the real (primary) and imaginary (secondary) parts of complex values.

Table 11. Complex type manipulation functions

Function	**Convert dual reals to complex (single-precision): __cmplxf**
Purpose	Converts two single-precision real values to a single complex value. The real *a* is converted to the primary element of the return value, and the real *b* is converted to the secondary element of the return value.
Formula	primary(result) = a secondary(result) = b
C/C++ prototype	float _Complex __cmplxf (float a, float b);
Fortran description	CMPLX(A,B) where A is of type REAL(4) where B is of type REAL(4) result is of type COMPLEX(4)
Function	**Convert dual reals to complex (double-precision): __cmplx**
Purpose	Converts two double-precision real values to a single complex value. The real *a* is converted to the primary element of the return value, and the real *b* is converted to the secondary element of the return value.
Formula	primary(result) = a secondary(result) = b
C/C++ prototype	double _Complex __cmplx (double a, double b); long double _Complex __cmplxl (long double a, long double b);[1]
Fortran description	CMPLX(A,B) where A is of type REAL(8) where B is of type REAL(8) result is of type COMPLEX(8)
Function	**Extract real part of complex (single-precision): __crealf**
Purpose	Extracts the primary part of a single-precision complex value *a* , and returns the result as a single real value.
Formula	result = primary(a)
C/C++ prototype	float __crealf (float _Complex a);
Fortran description	N/A
Function	**Extract real part of complex (double-precision): __creal, __creall**
Purpose	Extracts the primary part of a double-precision complex value *a* , and returns the result as a single real value.

Table 11. Complex type manipulation functions (continued)

Formula	result = primary(a)
C/C++ prototype	double __creal (double _Complex a); long double __creall (long double _Complex a);[1]
Fortran description	N/A
Function	**Extract imaginary part of complex (single-precision): __cimagf**
Purpose	Extracts the secondary part of a single-precision complex value *a* , and returns the result as a single real value.
Formula	result = secondary(a)
C/C++ prototype	float __cimagf (float _Complex a);
Fortran description	N/A
Function	Extract imaginary part of complex (double-precision): __cimag, __cimagl
Purpose	Extracts the imaginary part of a double-precision complex value *a* , and returns the result as a single real value.
Formula	result =secondary(a)
C/C++ prototype	double __cimag (double _Complex a); long double __cimagl (long double _Complex a);[1]
Fortran description	N/A
Notes: 1. 128-bit C/C++ long double types are not supported on Blue Gene/L. Long doubles are treated as regular double-precision doubles.	

Load and store functions

Table 12 lists and explains the various parallel load-and-store functions that are available.

Table 12. Load and store functions

Function	**Parallel load (single-precision): __lfps**
Purpose	Loads parallel single-precision values from the address of *a* , and converts the results to double-precision. The first word in *address(a)* is loaded into the primary element of the return value. The next word, at location *address(a)* +4, is loaded into the secondary element of the return value.
Formula	primary(result) = a[0] secondary(result) = a[1]
C/C++ prototype	double _Complex __lfps (float * a);
Fortran description	LOADFP(A) where A is of type REAL(4) result is of type COMPLEX(8)
Function	**Cross load (single-precision): __lfxs**

Table 12. Load and store functions (continued)

Purpose	Loads single-precision values that have been converted to double-precision, from the address of *a*. The first word in *address(a)* is loaded into the secondary element of the return value. The next word, at location *address(a)* +4, is loaded into the primary element of the return value.
Formula	primary(result) = a[1] secondary(result) = a[0]
C/C++ prototype	double _Complex __lfxs (float * a);
Fortran description	LOADFX(A) where A is of type REAL(4) result is of type COMPLEX(8)
Function	**Parallel load: __lfpd**
Purpose	Loads in parallel values from the address of *a*. The first word in *address(a)* is loaded into the primary element of the return value. The next word, at location *address(a)* +8, is loaded into the secondary element of the return value.
Formula	primary(result) = a[0] secondary(result) = a[1]
C/C++ prototype	double _Complex __lfpd(double* a);
Fortran description	LOADFP(A) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross load: __lfxd**
Purpose	Loads values from the address of *a*. The first word in *address(a)* is loaded into the secondary element of the return value. The next word, at location *address(a)* +8, is loaded into the primary element of the return value.
Formula	primary(result) = a[1] secondary(result) = a[0]
C/C++ prototype	double _Complex __lfxd (double * a);
Fortran description	LOADFX(A) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Parallel store (single-precision): __stfps**
Purpose	Stores in parallel double-precision values that have been converted to single-precision, into *address(b)*. The primary element of *a* is converted to single-precision and stored as the first word in *address(b)*. The secondary element of *a* is converted to single-precision and stored as the next word at location *address(b)* +4.

Formula	b[0] = primary(a)
	b[1] = secondary(a)

C/C++ prototype	void __stfps (float * b, double _Complex a);

Table 12. Load and store functions (continued)

Fortran description	STOREFP(B,A) where B is of type REAL(4) where A is of type COMPLEX(8) result is none
Function	**Cross store (single-precision): __stfxs**
Purpose	Stores double-precision values that have been converted to single-precision, into *address(b)*. The secondary element of *a* is converted to single-precision and stored as the first word in *address(b)*. The primary element of *a* is converted to single-precision and stored as the next word at location *address(b)* +4.
Formula	b[0] = secondary(a) b[1] = primary(a)
C/C++ prototype	void __stfxs (float * b, double _Complex a);
Fortran description	STOREFX(B,A) where B is of type REAL(4) where A is of type COMPLEX(8) result is none
Function	**Parallel store: __stfpd**
Purpose	Stores in parallel values into *address(b)*. The primary element of *a* is stored as the first double word in *address(b)*. The secondary element of *a* is stored as the next double word at location *address(b)* +8.

Formula	b[0] = primary(a)
	b[1] = secondary(a)

C/C++ prototype	void __stfpd (double * b, double _Complex a);
Fortran description	STOREFP(B,A) where B is of type REAL(8) where A is of type COMPLEX(8) result is none
Function	**Cross store: __stfxd**
Purpose	Stores values into *address(b)*. The secondary element of *a* is stored as the first double word in *address(b)*. The primary element of *a* is stored as the next double word at location *address(b)* +8.

Formula	b[0] = secondary(a)
	b[1] = primary(a)

C/C++ prototype	void __stfxd (double * b, double _Complex a);
Fortran description	STOREFP(B,A) where B is of type REAL(8) where A is of type COMPLEX(8) result is none
Function	**Parallel store as integer: __stfpiw**

Table 12. Load and store functions (continued)

Purpose	Stores in parallel floating-point double-precision values into *b* as integer words. The lower-order 32 bits of the primary element of *a* are stored as the first integer word in *address(b)*. The lower-order 32 bits of the secondary element of *a* are stored as the next integer word at location *address(b)* +4. This function is typically preceded by a call to the **__fpctiw** or **__fpctiwz** built-in functions, described in Unary functions, which perform parallel conversion of dual floating-point values to integers.

Formula	b[0] = primary(a)
	b[1] = secondary(a)

C/C++ prototype	void __stfpiw (int * b, double _Complex a);
Fortran description	STOREFP(B,A) where B is of type INTEGER(4) where A is of type COMPLEX(8) result is none

Move functions

Table 13.

Function	**Cross move: __fxmr**
Purpose	Swaps the values of the primary and secondary elements of operand *a*.

Formula	primary(result) = secondary(a)
	secondary(result) = primary(a)

C/C++ prototype	double _Complex __fxmr (double _Complex a);
Fortran description	FXMR(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Arithmetic functions

The following sections describe all the arithmetic built-in functions, categorized by their number of operands:

- Unary functions
- Binary functions
- Multiply-add functions

Unary functions

Unary functions operate on a single input operand. These functions are listed in Table 14.

Table 14. Unary functions

Function	**Parallel convert to integer: __fpctiw**

Table 14. Unary functions (continued)

Purpose	Converts in parallel the primary and secondary elements of operand *a* to 32-bit integers. After a call to this function, use the **__stfpiw** function to store the converted integers in parallel, as described in Load and store functions.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpctiw (double _Complex a);
Fortran description	FPCTIW(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel convert to integer and round to zero: __fpctiwz**
Purpose	Converts in parallel the primary and secondary elements of operand *a* to 32 bit integers and rounds the results to zero. After a call to this function, you will want to use the **__stfpiw** function to store the converted integers in parallel, as described in Load and store functions.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpctiwz(double _Complex a);
Fortran description	FPCTIWZ(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel round double-precision to single-precision: __fprsp**
Purpose	Rounds in parallel the primary and secondary elements of double-precision operand *a* to single precision.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fprsp (double _Complex a);
Fortran description	FPRSP(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel reciprocal estimate: __fpre**
Purpose	Calculates in parallel double-precision estimates of the reciprocal of the primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpre(double _Complex a);
Fortran description	FPRE(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel reciprocal square root: __fprsqrte**
Purpose	Calculates in parallel double-precision estimates of the reciprocals of the square roots of the primary and secondary elements of operand *a*.

Table 14. Unary functions (continued)

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fprsqrte (double _Complex a);
Fortran description	FPRSQRTE(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negate: __fpneg**
Purpose	Calculates in parallel the negative absolute values of the primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpneg (double _Complex a);
Fortran description	FPNEG(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel absolute: __fpabs**
Purpose	Calculates in parallel the absolute values of the primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpabs (double _Complex a);
Fortran description	FPABS(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negate absolute: __fpnabs**
Purpose	Calculates in parallel the negative absolute values of the primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpnabs (double _Complex a);
Fortran description	FPNABS(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Binary functions

Binary functions operate on two input operands. The functions are listed in Table 15.

Table 15.

Function	**Parallel add: __fpadd**

Table 15. (continued)

Purpose	Adds in parallel the primary and secondary elements of operands *a* and *b*.

Formula	primary(result) = primary(a) + primary(b)
	secondary(result) = secondary(a) + secondary(b)

C/C++ prototype	double _Complex __fpadd (double _Complex a, double _Complex b);
Fortran description	FPADD(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel subtract: __fpsub**
Purpose	Subtracts in parallel the primary and secondary elements of operand *b* from the corresponding primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a) - primary(b)
	secondary(result) = secondary(a) - secondary(b)

C/C++ prototype	double _Complex __fpsub (double _Complex a, double _Complex b);
Fortran description	FPSUB(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel multiply: __fpmul**
Purpose	Multiples in parallel the values of primary and secondary elements of operands *a* and *b*.

Formula	primary(result) = primary(a) × primary(b)
	secondary(result) = secondary(a) × secondary(b)

C/C++ prototype	double _Complex __fpmul (double _Complex a, double _Complex b);
Fortran description	FPMUL(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Cross multiply: __fxmul
Purpose	The product of the secondary element of *a* and the primary element of *b* is stored as the primary element of the return value. The product of the primary element of *a* and the secondary element of *b* is stored as the secondary element of the return value.

Formula	primary(result) = secondary(a) x primary(b)
	secondary(result) = primary(a) × secondary(b)

C/C++ prototype	double _Complex __fxmul (double _Complex a, double _Complex b);
Fortran description	FXMUL(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross copy multiply: _fxpmul, __fxsmul**

Table 15. (continued)

Purpose	Both of these functions can be used to achieve the same result. The product of *a* and the primary element of *b* is stored as the primary element of the return value. The product of *a* and the secondary element of *b* is stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b)
	secondary(result) = a x secondary(b)

C/C++ prototype	double _Complex __fxpmul (double _Complex b, double a); double _Complex __fxsmul (double _Complex b, double a);
Fortran description	FXPMUL(B,A) or FXSMUL(B,A) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Multiply-add functions

Multiply-add functions take three input operands, multiply the first two, and add or subtract the third.

Table 16.

Function	**Parallel multiply-add: __fpmadd**
Purpose	The sum of the product of the primary elements of *a* and *b*, added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of the secondary elements of *a* and *b*, added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = primary(a) × primary(b) + primary(c)
	secondary(result) = secondary(a) × secondary(b) + secondary(c)

C/C++ prototype	double _Complex __fpmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negative multiply-add: __fpnmadd**
Purpose	The sum of the product of the primary elements of *a* and *b*, added to the primary element of *c*, is negated and stored as the primary element of the return value. The sum of the product of the secondary elements of *a* and *b*, added to the secondary element of *c*, is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(primary(a) × primary(b) + primary(c))
	secondary(result) = -(secondary(a) × secondary(b) + secondary(c))

C/C++ prototype	double _Complex __fpnmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Table 16. (continued)

Function	Parallel multiply-subtract: __fpmsub
Purpose	The difference of the primary element of *c*, subtracted from the product of the primary elements of *a* and *b*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of the secondary elements of *a* and *b*, is stored as the secondary element of the return value.

Formula	primary(result) = primary(a) × primary(b) - primary(c)
	secondary(result) = secondary(a) × secondary(b) - secondary(c)

C/C++ prototype	double _Complex __fpmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negative multiply-subtract: __fpnmsub**
Purpose	The difference of the primary element of *c*, subtracted from the product of the primary elements of *a* and *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of the secondary elements of *a* and *b*, is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(primary(a) × primary(b) - primary(c))
	secondary(result) = -(secondary(a) × secondary(b) - secondary(c))

C/C++ prototype	double _Complex __fpnmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross multiply-add: __fxmadd**
Purpose	The sum of the product of the secondary element of *a* and the primary element of *b*, added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of the primary element of *a* and the secondary *b*, added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = secondary(a) × primary(b) + primary(c)
	secondary(result) = primary(a) × secondary(b) + secondary(c)

C/C++ prototype	double _Complex __fxmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross negative multiply-add: __fxnmadd**

Table 16. (continued)

Purpose	The sum of the product of the secondary element of *a* and the primary element of *b* , added to the primary element of *c*, is negated and stored as the primary element of the return value. The sum of the product of the primary element of *a* and the secondary element of *b*, added to the secondary element of *c*, is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(secondary(a) × primary(b) + primary(c))
	secondary(result) = -(primary(a) × secondary(b) + secondary(c))

C/C++ prototype	double _Complex __fxnmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross multiply-subtract: __fxmsub**
Purpose	The difference of the primary element of *c*, subtracted from the product of the secondary element of *a* and the primary element of *b* , is stored as the primary element of the return primary element of *a* and the secondary element of *b* is stored as the secondary element of the return value.

Formula	primary(result) = secondary(a) × primary(b) - primary(c)
	secondary(result) = primary(a) × secondary(b) - secondary(c)

C/C++ prototype	double _Complex __fxmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross negative multiply-subtract: __fxnmsub**
Purpose	The difference of the primary element of *c*, subtracted from the product of the secondary element of *a* and the primary element of *b* , is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of the primary element of *a* and the secondary element of *b* , is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(secondary(a) × primary(b) - primary(c))
	secondary(result) = -(primary(a) × secondary(b) - secondary(c))

C/C++ prototype	double _Complex __fxnmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross copy multiply-add: __fxcpmadd, __fxcsmadd**

Table 16. (continued)

Purpose	Both of these functions can be used to achieve the same result. The sum of the product of *a* and the primary element of *b*, added to the primary element of *c* , is stored as the primary element of the return value. The sum of the product of *a* and the secondary element of *b*, added to the secondary element of *c* , is stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b) + primary(c)
	secondary(result) = a x secondary(b) + secondary(c)

C/C++ prototype	double _Complex __fxcpmadd (double _Complex c, double _Complex b, double a); double _Complex __fxcsmadd (double _Complex c, double _Complex b, double a);
Fortran description	FXCPMADD(C,B,A) or FXCSMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy negative multiply-add: __fxcpnmadd, __fxcsnmadd**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c* , subtracted from the product of *a* and the secondary element of *b*, is negated stored as the secondary element of the return value.

Formula	primary(result) = -(a x primary(b) + primary(c))
	secondary(result) = -(a x secondary(b) + secondary(c))

C/C++ prototype	double _Complex __fxcpnmadd (double _Complex c, double _Complex b, double a); double _Complex __fxcsnmadd (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNMADD(C,B,A) or FXCSNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy multiply-subtract: __fxcpmsub, __fxcsmsub**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b*, is stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b) - primary(c)
	secondary(result) = a x secondary(b) - secondary(c)

C/C++ prototype	double _Complex __fxcpmsub (double _Complex c, double _Complex b, double a); double _Complex __fxcsmsub (double _Complex c, double _Complex b, double a);

Table 16. (continued)

Fortran description	FXCPMSUB(C,B,A) or FXCSMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy negative multiply-subtract: __fxcpnmsub, __fxcsnmsub**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b*, is negated stored as the secondary element of the return value.

Formula	primary(result) = -(a x primary(b) - primary(c))
	secondary(result) = -(a x secondary(b) - secondary(c))

C/C++ prototype	double _Complex __fxcpnmsub (double _Complex c, double _Complex b, double a); double _Complex __fxcsnmsub (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNMSUB(C,B,A) or FXCSNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy sub-primary multiply-add: __fxcpnpma, __fxcsnpma**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b*, is negated and stored as the primary element of the return value. The sum of the product of *a* and the secondary element of *b*, added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = -(a x primary(b) - primary(c))
	secondary(result) = a x secondary(b) + secondary(c)

C/C++ prototype	double _Complex __fxcpnpma (double _Complex c, double _Complex b, double a); double _Complex __fxcsnpma (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNPMA(C,B,A) or FXCSNPMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy sub-secondary multiply-add: __fxcpnsma, __fxcsnsma**
Purpose	Both of these functions can be used to achieve the same result. The sum of the product of *a* and the primary element of *b*, added to the primary element of *c*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b*, is negated and stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b) + primary(c))
	secondary(result) = -(a x secondary(b) - secondary(c))

Table 16. (continued)

C/C++ prototype	double _Complex ____fxcpnsma (double _Complex c, double _Complex b, double a); double _Complex __fxcsnsma (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNSMA(C,B,A) or FXCSNSMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed multiply-add: __fxcxma**
Purpose	The sum of the product of *a* and the secondary element of *b*, added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of *a* and the primary element of *b*, added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = a x secondary(b) + primary(c)
	secondary(result) = a x primary(b) +secondary(c)

C/C++ prototype	double _Complex __fxcxma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed negative multiply-subtract: __fxcxnms**
Purpose	The difference of the primary element of *c*, subtracted from the product of *a* and the secondary element of *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the primary element of *b*, is negated and stored as the primary secondary of the return value.

Formula	primary(result) = -(a × secondary(b) - primary(c))
	secondary(result) = -(a × primary(b) - secondary(c))

C/C++ prototype	double _Complex __fxcxnms (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNMS(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed sub-primary multiply-add: __fxcxnpma**
Purpose	The difference of the primary element of *c*, subtracted from the product of *a* and the secondary element of *b*, is stored as the primary element of the return value. The sum of the product of *a* and the primary element of *b*, added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = (a × secondary(b) - primary(c))
	secondary(result) = a × primary(b) + secondary(c)

C/C++ prototype	double _Complex __fxcxnpma (double _Complex c, double _Complex b, double a);

Table 16. (continued)

Fortran description	FXCXNPMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed sub-secondary multiply-add: __fxcxnsma**
Purpose	The sum of the product of *a* and the secondary element of *b*, added to the primary element of *c*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the primary element of *b*, is stored as the secondary element of the return value.

Formula	primary(result) = a x secondary(b) + primary(c))
	secondary(result) = -(a x primary(b) - secondary(c))

C/C++ prototype	double _Complex __fxcxnsma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNSMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)

Select functions

Table 17 lists and explains the select functions that are available.

Table 17. Select functions

Function	**Parallel select: __fpsel**
Purpose	The value of the primary element of *a* is compared to zero. If its value is equal to or greater than zero, the primary element of *c* is stored in the primary element of the return value. Otherwise, the primary element of *b* is stored in the primary element of the return value. The value of the secondary element of *a* is compared to zero. If its value is equal to or greater than zero, the secondary element of *c* is stored in the secondary element of the return value. Otherwise, the secondary element of *b* is stored in the secondary element of the return value.

Formula	primary(result) = if primary(a) ≥ 0 then primary(c); else primary(b)
	secondary(result) = if secondary(a) ≥ 0 then primary(c); else secondary(b)

C/C++ prototype	double _Complex __fpsel (double _Complex a, double _Complex b, double _Complex c);
Fortran description	FPSEL(A,B,C) where A is of type COMPLEX(8) where B is of type COMPLEX(8) where C is of type COMPLEX(8) result is of type COMPLEX(8)

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interface allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Note: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

Index

Program Number: 5799-TRG
5799-TRH

Printed in USA

SC10-4310-00