Convey Programmers Guide

November 2010
Version 1.8
901-000003-000

© Convey Computer™ Corporation 2007-2010.
All Rights Reserved.
1302 East Collins
Richardson, TX 75081

The Information in this document is provided for use with Convey Computer Corporation (“Convey”) products. No license, express or implied, to any intellectual property associated with this document or such products is granted by this document.

All products described in this document whose name is prefaced by “Convey” or “Convey enhanced “ (“Convey products”) are owned by Convey Computer Corporation (or those companies that have licensed technology to Convey) and are protected by patents, trade secrets, copyrights or other industrial property rights.

The Convey products described in this document may still be in development. The final form of each product and release date thereof is at the sole and absolute discretion of Convey. Your purchase, license and/or use of Convey products shall be subject to Convey’s then current sales terms and conditions.

Trademarks
The following are trademarks of Convey Computer Corporation in the United States and other countries:

Convey Computer

The Convey Logo

Convey HC-1

Trademarks of other companies

Intel is a registered trademark of Intel Corporation

Adobe and Adobe Reader are registered trademarks of Adobe Systems Incorporated

Linux is a registered trademark of Linus Torvalds

PathScale is a trademark of PathScale LLC.

Xilinx and Virtex are registered trademarks of Xilinx, Inc.

Revisions

Version	**Description**
1.0	May 2008. Original printing.
1.1	August 2008. Renamed CNY_PERSISTANT -> CNY_PERSISTENT, documented **cpm** tool.
1.2	July 2009. Convey compiler support for the coprocessor.
1.3	Minor corrections. New FENCE directives/pragmas. New include file for C programs using support routines.
1.4	Beta release of the Financial Analytics personality.
1.5	Jan 2010. Production release of development tools, math libraries, and FPGA images, with assorted bug fixes and performance improvements. Production release of the Financial Analytics personality.
1.6	April 2010. Production release of development tools, math libraries, and FPGA images, with assorted bug fixes, enhancements, and performance improvements.
1.7	July 2010. Production release of development tools, math libraries, and FPGA images, with assorted bug fixes, enhancements, and performance improvements. Added user-defined intrinsic and memory fence documentation.
1.8	November 2010. Production release of development tools and math libraries, containing assorted bug fixes. New "fence" directive, "no_loop_directive" without a symbol list, improved vectorization. PDF support for table of contents navigation (Adobe Reader: click on *bookmarks* in the navigation panel, for evince (linux systems), enable "*Side Pane*" under "*View*", and select "*Index*")

Table of Contents

1 Introduction 7
  1.1 Intended Audience 7
  1.2 Other Sources for Additional Information 7
    1.2.1 Online FAQs and Support Bulletins 7
    1.2.2 Other Suggested Documents 7
  1.3 Typographical Conventions 8
2 Overview 9
  2.1 Definitions 9
  2.2 Convey HC-1 Architectural Overview 12
  2.3 Introduction to the Convey Programming Model 13
    2.3.1 Programming Model Examples: 14
  2.4 Porting Applications to a Convey Hybrid-Core Server --Overview 16
    2.4.1 Brief Overview of the Convey Programming Model 17
3 Convey's Software Products for Programmers 18
  3.1 Convey Compilers and Assembler 19
    3.1.1 cnycc 20
    3.1.2 cnyCC 20
    3.1.3 cnyf95 20
  3.2 as – Convey's Assembler 21
  3.3 Object File Compatibility 21
    3.3.1 Compatibility with Open64 and PathScale Products 21
    3.3.2 Compatibility with Intel and GNU Products 21
4 Porting Applications for the Convey Coprocessor 23
  4.1 Porting Existing Applications Overview 23
  4.2 Host Code and Coprocessor Code Interactions and Limitations 25
  4.3 Support for Portable Applications 26
    4.3.1 Running Portable Applications that Utilize the Convey Coprocessor on a non-Convey System 28
  4.4 How to Customize Source Code for Convey's Compilers 29
  4.5 Pointer Aliasing Model and Vectorization 29
    4.5.1 Default Alias Models 29
    4.5.2 Additional Aliasing Models 30
    4.5.3 Specifying An Aliasing Model 31
  4.6 Coprocessor Routines and Regions 33
  4.7 Naming Conventions for Routines in Coprocessor Regions 34
    4.7.1 Assembly Language Routine Names 34
  4.8 User-Defined Intrinsics 35
  4.9 Compiler flags 36
  4.10 Convey Directives and Pragmas 44
    4.10.1 Convey Coprocesser Code Generation Directives and Pragmas 45
    4.10.2 Directive and Pragma Examples 49
  4.11 Coprocessor Code Generation Inhibitors 50
  4.12 Memory Model 52
    4.12.1 Host and Coprocesser Memory Granularity 52
    4.12.2 Convey Memory Locality Directives and Pragmas 53

4.12.3 Initial Memory Allocation 53
4.12.4 Runtime Memory Migration 54
4.12.5 When to Migrate Memory 55
4.13 Coprocessor Exceptions 55
4.14 Automatic Vectorization 55
4.15 Vectorization Report 56
5 Coprocessor Code Examples 58
5.1 Vectorizing Loops Example 58
5.2 Loop Unrolling / Vector Reduction Example 62
6 Personality Specific Features 64
6.1 Financial Analytics Personality 64
6.1.1 Cumulative Normal Distribution Function 65
6.1.2 Mersenne-Twister Random Number Generators 65
6.1.3 SOBOL Random Numbers 69
6.1.4 Financial Analytics Example (using optimized intrinsics) 70
7 Linking an Executable that Contains Coprocessor Code 72
8 Executing Applications that Contain Coprocessor Code 73
8.1 Executing a Program using the Convey Coprocessor Simulator 73
8.2 Executing a Program using the Convey Coprocessor 75
8.3 Process Groups 76
8.4 MPI Example 76
9 User Callable Support Routines 80
10 Assembler Usage 87
10.1 Important Assembler Usage Notes 88
10.2 Sample coprocessor assembly code 89
11 Coprocessor Assembly Language Calling Conventions 90
11.1 C/C++ Calling Conventions 90
11.2 Fortran Calling Conventions 90
12 Environment Variable Summary 92
12.1 Convey Programming Model Environment Variables 92
13 Debugging Coprocessor Code with gdb 98
13.1 Selecting Convey Coprocessor Architecture or Host Architecture 98
13.2 Debugging user written assembly language 98
13.3 Convey coprocessor gdb enhancements 99
13.4 Convey Single Precision Personality Registers 101
13.5 Convey Financial Analytics Personality Registers 102
13.6 Convey Custom Personality Registers 102
13.7 Cautions When Using Multiple Architectures 103
14 Personalities, Signatures, and Signature Resolution 104
14.1 Signature Definition 105
14.2 Partial Signatures, Zero-valued Version Numbers, and Signature Upgrades ... 105
14.2.1 Zero-valued Version Numbers 105
14.2.2 Version Number Upgrade Attributes 106
14.3 Signature Resolution 106
14.3.1 Compile Time Signature Resolution 106
14.3.2 Runtime Signature Resolution Overview 107
14.3.3 Controlling Signature Resolution 109
14.4 Personality Tools 111
14.5 Canonical Instruction Set Version Number 111
14.6 Object Code Version Number 111

15 Customer Support Procedures 112
Appendix A How To Take Advantage Of The Convey Coprocessor 113
Convey Mathematical Library 113
Using Convey directives and source code restructuring 113
Automatic Vectorization 114
Developing a custom personality 114
Appendix B Optimizing Performance When The Coprocessor Is Unavailable 116
Appendix C Less Commonly Used Compiler Flags 117
Appendix D Less Commonly Used Environment Variables 123
Appendix E Coprocessor Intrinsics 124
Appendix F User Provided Dual Target Profitability 126
Appendix G Low Level Interfaces to Coprocessor Routines 128
C language interface routines 128
Fortran language interface routines 129
Simple Coprocessor Interface Routine Example 130
Sample coprocessor routine call in C 131
Sample coprocessor routine call in Fortran 131
Appendix H Process Group Tutorial 133
Appendix I Fence Checking with the Convey Simulator and Controlling Compiler Generated Fences 137
Introduction 137
Overview of the FENCE Instruction 137
The Convey `no_fence` Directive 139
The Convey Simulator and Fence Checking 140
Example Using Fence Checking 141

1 Introduction

The Convey Programmers Guide describes how to use the following Convey software development tools:

- the Convey compilers (C, C++, and Fortran),
- the Convey Assembler, Linker, and runtime libraries,
- the Convey-enhanced *gdb* debugger,
- the coprocessor performance analysis tool (*spat*), and
- the Convey coprocessor simulator

The latest revision of this document is always available at **conveysupport.com**.

Please note that this document contains clickable hyperlinks. If you are browsing this document on Convey's web site, or downloaded ALL of Convey's documents to the same directory, those clickable hyperlinks will work. If you only download a single document, clicking on the hyperlinks will fail.

1.1 Intended Audience

This guide is recommended reading for users who are developing or porting software for Convey hybrid-core servers.

This guide is also suggested reading for any user developing a custom personality with the Convey Personality Development Kit.

1.2 Other Sources for Additional Information

Most of the web pages and documents described below are located on the Convey Support website, and a Convey provided username and password is required to access them. Usually, the necessary username and password are provided to the technical contact provided when a Convey Server or Convey SW Development toolset is ordered. You may also contact support@conveycompter.com to request a username and password.

1.2.1 Online FAQs and Support Bulletins

Convey provides assorted additional information in the FAQ and Support Bulletin portions of the Convey Support website:

- **http://conveysupport.com/protected/faqs**
- **http://conveysupport.com/protected/bulletins**

1.2.2 Other Suggested Documents

Convey Reference Manual – describes the Convey coprocessor architecture and instruction sets

Convey Mathematical Libraries Guide – describes the higher level mathematical functions available in the Convey Math Libraries, that are optimized to take advantage of the Convey coprocessor when using Convey's vector personalities

Convey System Administration Guide – describes Convey software installation and packaging, first boot customization instructions, and customer support procedures

Convey PDK Reference Manual – for users of the Convey Personality Development Kit

Convey Spat Users Guide – describes how to use the Simulator Performance Analysis Tool, for examining the performance characteristics of coprocessor code

All the above documents are available at conveysupport.com, and in `/opt/convey/doc` on systems where the corresponding software has been installed.

Several additional Convey documents describing various Convey developed personalities are also available under a non-disclosure agreement, when that corresponding personality is purchased. Email support@conveycomputer.com for more information.

1.3 Typographical Conventions

A `brown fixed width font` is used for things a user might type, including shell commands, program names, command flags, filenames, file contents, source code, etc. A `dark green fixed width font` is used for C/C++ specific compiler flags, while a `purple fixed width font` is used for Fortran specific compiler flags.

A `black fixed width font` is used for system-generated output.

Italicized text is used to identify key concepts.

Bold text is used to draw attention to paragraph and chapter headings, and when italicized, to mark the first time a key concept or idea is presented.

`A dark red fixed width font` is used for Convey provided items, including Convey specific compiler directives or flags, Convey library routine names, Convey `include` filenames, and Convey programs/scripts.

Underlined blue text is used for clickable hyperlinks, both internal links (within this document) and external links. Convey's online documentation is typically provided in Adobe's® Portable Document Format (PDF). Most web browsers with the Adobe Reader® plugin installed can display Convey manuals and follow the embedded hyperlinks.

2 Overview

2.1 Definitions

Convey Server Terminology

hybrid-core server	The Convey *hybrid-core server* combines a host x86-64 processor and a Convey coprocessor. The host processor and coprocessor share a cache coherent global memory address space. A program can execute code on both processors concurrently.
coprocessor	The Convey *coprocessor* is an attached processor that can be loaded with different personalities, including both Convey-defined and user-defined personalities.
personality	A *personality* is an instruction set that is implemented on the Convey coprocessor. Personalities may be swapped during execution of a program. All personalities, including user-defined personalities, share a common core scalar instruction set. Convey currently provides a single precision vector personality and several double precision vector personalities.
signature	A *signature* identifies one specific version of a personality, and a corresponding coprocessor firmware image. Signatures contain a personality name/number, major and minor version numbers, and a coprocessor hardware model number. Signatures may be fully resolved or partially resolved. A fully resolved signature is a signature that has non-zero version numbers, and is installed and enabled. A partially resolved signature is converted into a fully resolved signature as described later in this document. The following are some sample signatures:

single	A partial signature selecting the single precision vector instruction set
sp	Another nickname for **single**
double.2.1	A partial signature selecting the double precision vector instruction set, version 2.1
2.1.1.3	A complete (fully resolved) signature, specifying personality number 2, version 1.1, and hardware model number 3.

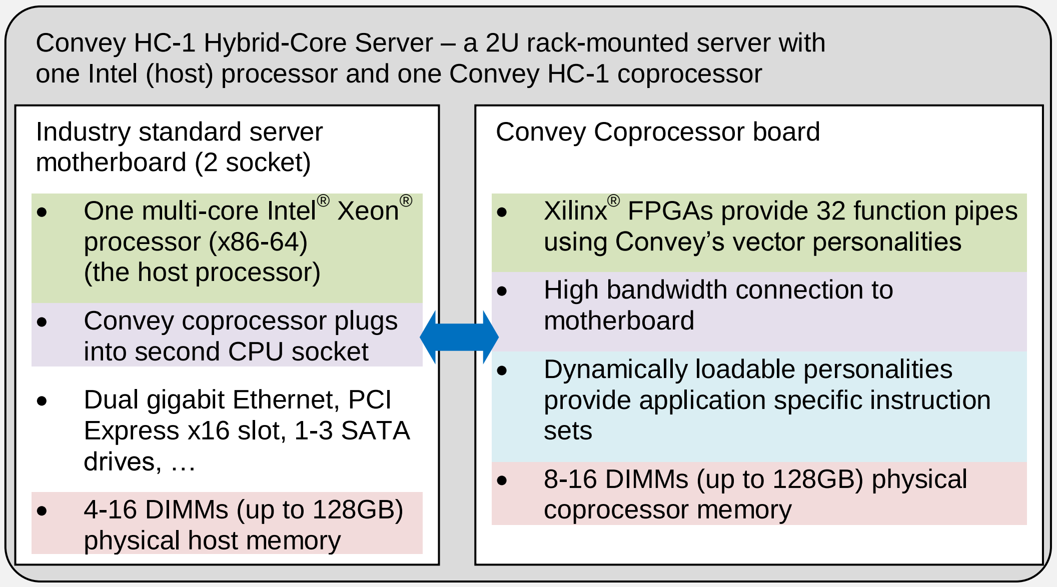
coprocessor image	A *coprocessor image* is the FPGA instruction image that is downloaded into the Convey coprocessor whenever coprocessor code for the corresponding signature is about to be executed. The image implements the desired instruction set. Each installed signature has its own coprocessor image.
signature resolution	When a user compiles a routine and specifies a partial signature, that signature will undergo two levels of *signature resolution* to select an actual installed signature, controlled by various system defaults and environment variables. Compile time signature resolution resolves a partial signature to select a particular signature for the compiler and assembler to use, but permits a different but compatible signature to be used when executing the program. Runtime signature resolution allows programs to be compiled using a particular signature, and then later to be executed using a compatible signature, such as a newer release of a personality, different coprocessor hardware model number, or a system administrator 'approved' version. Runtime signature resolution also allows a single runtime signature to be used instead of several different but compatible compile time signatures, reducing the time spent loading coprocessor images.

Convey Coprocessor/Compiler Terminology

attached coprocessor	The Convey coprocessor must be *attached* to a process before it can be used. Only one process or Linux process group (i.e. an MPI process group) can attach the coprocessor at a time. All the threads in such a process are free to use the coprocessor. Convey's Linux operating system typically attaches the coprocessor (if it is available) when an application that expects to use the coprocessor starts execution. Some applications that utilize the Convey Mathematical Libraries don't attach to the coprocessor until the first library routine that uses the coprocessor is called.
coprocessor code	A sequence of Convey coprocessor instructions -- Coprocessor code can only be executed on the Convey coprocessor or the Convey coprocessor simulator. Coprocessor code may be dispatched from host code, or called directly from other coprocessor code.

coprocessor region	A block of source code within a routine that is (typically) compiled for **both** the host processor (an x86-64 processor) and the Convey coprocessor. At runtime, when a coprocessor region is executed by the host processor, the host code will first check to see if the Convey coprocessor is attached to this process and (optionally) determine if executing that region on the coprocessor is expected to be profitable (faster). If so, the coprocessor version of that region is dispatched to the coprocessor. Otherwise, the equivalent host code for that region is executed. Coprocessor regions may be created explicitly with Convey's directives/pragmas and compiler flags, or implicitly with the `-mcny_auto_vector` flag.
coprocessor routine	An entire routine can be compiled for the coprocessor with the `-mcny_dual_target`, `-mcny_dual_target_nowrap`, or `-mcny_pure` flags. The assembler can also be used to create a routine that is entirely coprocessor code.
host code	x86-64 executable code that executes on the host processor.
host processor	The x86-64 processor. Refers to both the x86-64 processor on a Convey hybrid-core server as well as the x86-64 processor on any generic x86-64 Linux system.
coprocessor dispatch	The host processor can *dispatch* the coprocessor version of the code region, that is, initiate execution of coprocessor instructions. Coprocessor code will only be dispatched when the coprocessor is currently attached to the current process or process group. If the coprocessor is attached to the current process (or process group) but is busy, the dispatch is queued, and will start as soon as the coprocessor is free.
simulator	The Convey *Simulator* is a functional simulator that emulates executing instructions on the Convey coprocessor. It directly supports Convey's personalities (single precision vector, double precision vector, etc.). It also provides hooks for simulating user defined instructions, as defined by a custom personality, using user provided emulation code. The Convey simulator is used by setting the `CNY_SIM_THREAD` environment variable (i.e. `export CNY_SIM_THREAD=libcpSimLib2.so`). When `CNY_SIM_THREAD` is set, the Convey simulator is used instead of the coprocessor.
spat	The Convey Simulator Performance Analysis Tool (**`spat`**) is a simulator that emulates the instruction timing of programs executing on the Convey coprocessor. It takes instruction traces from the Convey Simulator and generates plots and visual displays showing instruction flow through the coprocessor.

2.2 Convey HC-1 Architectural Overview



Convey HC-1 Hybrid-Core Server – a 2U rack-mounted server with one Intel (host) processor and one Convey HC-1 coprocessor

Industry standard server motherboard (2 socket)

- One multi-core Intel® Xeon® processor (x86-64) (the host processor)
- Convey coprocessor plugs into second CPU socket
- Dual gigabit Ethernet, PCI Express x16 slot, 1-3 SATA drives, …
- 4-16 DIMMs (up to 128GB) physical host memory

Convey Coprocessor board

- Xilinx® FPGAs provide 32 function pipes using Convey's vector personalities
- High bandwidth connection to motherboard
- Dynamically loadable personalities provide application specific instruction sets
- 8-16 DIMMs (up to 128GB) physical coprocessor memory

Common, cache coherent, virtual address space: Within an application, the host processor and the coprocessor use the same virtual address to access a particular memory location, regardless of the physical location of the memory.

The host processor can access its own memory quicker than coprocessor memory. Similarly, the coprocessor can access its own memory faster than host memory. Virtual pages can be allocated in either host or coprocessor memory, and migrated from one to the other during execution of an application to improve performance.

The additional instructions an HC-1 coprocessor provides to an application are defined by the coprocessor personality currently loaded. Different coprocessor personalities, each with different additional instructions, can be loaded into the coprocessor as needed during the execution of an application. Host (x86-64) code can *dispatch* a coprocessor code segment, passing it arguments and addresses. See the programming model discussion below for more details.

Convey provides single precision, double precision, and financial vector personalities. These personalities provide a common scalar instruction set, and a vector instruction set similar to that found on a vector supercomputer, tuned for a particular data type and/or application area. Convey will add additional personalities, each designed for a particular type of application, in the future.

The user can also define and implement a custom personality. A custom personality contains the same scalar instruction set as Convey's vector personalities, and one or more additional user defined instructions. Implementing a new custom personality requires designing and implementing an FPGA image using Convey and Xilinx FPGA tools. The new user defined coprocessor instructions can be executed from C, C++, Fortran, or coprocessor assembly language.

Convey also provides custom personality design and implementation services.

The **Convey Reference Manual** provides an extensive description of the HC-1 architecture.

2.3 Introduction to the Convey Programming Model

This introduction to the programming model is intended to show a few key concepts and capabilities. A complete description of Convey's Programming Model is provided in chapter 4 **Porting Applications for the Convey Coprocessor**.

Key features and differences compared to other attached coprocessor programming models include:

- **Applications are coded in standard C, C++, and Fortran.** Additional directives and pragmas are available to guide the compilers and improve performance.
- A routine coded in C, C++, or Fortran, or a portion thereof, may be compiled to execute on the host processor, the coprocessor, or both. No special library calls or memory relocation actions are required.

  The primary restrictions on code compiled for the coprocessor are that it cannot:

  - perform I/O,
  - call an OS (operating system) routine, or
  - call a host routine.
- The coprocessor's extended instruction set is defined by the currently loaded **personality**.
  - Personalities can be dynamically loaded as needed, and each personality provides a particular instruction set geared towards a particular application or set of operations.
  - Convey provides a set of floating point vector personalities. Each vector personality provides a common set of scalar instructions, some integer vector operations, and a highly optimized set of floating point vector operations for one particular floating point data type. Since these vector personalities are optimized for floating point vector operations, loops consisting mainly of these operations are the best candidates for execution on the coprocessor.
  - Custom personalities may also be developed that implement a set of user-defined instructions. This capability requires some advanced skills to program the FPGA. These user-defined coprocessor instructions may be invoked directly from C, C++, and Fortran, as well as in coprocessor assembly language routines.
- Lastly, the Convey Mathematical Library (CML) contains routines to perform common mathematical functions on the coprocessor, including
  - Dense vector and matrix operations, including the Level 1, 2, and 3 BLAS
  - Sparse vector operations, including the sparse BLAS
  - Linear equation solution, including LAPACK
  - Eigenvalue solution, including LAPACK
  - Discrete Fourier Transforms, including 1-D, 2-D, 3-D and multiple 1-D

  Some applications that were originally linked with Intel's MKL shared library can utilize certain routines in the Convey Mathematical Library without recompilation or re-linking.

2.3.1 Programming Model Examples:

Consider the following simplified saxpy routine (`saxpy.c`):

```
void saxpy (int n, float a, float x[], float y[])
{
  int i;
  for (i = 0; i < n; i++) {
    y[i] = a * x[i] + y[i];
  }
}
```

Each of the following examples below show a different capability provided by the Convey programming model for use with one of Convey's vector personalities. One or more of these capabilities will typically be used when porting an application to use the Convey coprocessor. The "**VECTOR LOOP**" line in each example is the compiler's vectorization report, indicating which loop was vectorized, and how many vector operations of various types were generated (report format described below).

User directed coprocessor code generation for loops

A specific region of code can be compiled for the coprocessor by using the **begin_coproc**/**end_coproc** pragmas:

```
void saxpy (int n, float a, float x[], float y[])
{
  int i;
#pragma cny begin_coproc
  for (i = 0; i < n; i++) {
    y[i] = a * x[i] + y[i];
  }
#pragma cny end_coproc
}
```

Compile with the **-mcny_vector** and **-std=c99** flags to produce a coprocessor region for the `for` loop, and the compiler will vectorize the loop:

```
VECTOR LOOP in saxpy0 at 5: L 2 S 1 B 2 U 0 I 0 M 0
```

Compiling whole routines for the coprocessor

An entire function can be compiled for the coprocessor by using the **dual_target** directive/pragma:

```
void saxpy (int n, float a, float x[], float y[])
{
#pragma cny dual_target(1)
  int i;
  for (i = 0; i < n; i++) {
    y[i] = a * x[i] + y[i];
  }
}
```

Compile with the **-mcny_vector** flag, or, alternatively, compile the original saxpy routine (without any pragmas) with the **-mcny_dual_target**, **-mcny_vector**, and **-std=c99** compiler flags, to produce both a host and a coprocessor version of the `saxpy` routine.

```
$ cnycc -c -std=c99 -mcny_vector saxpy.c

VECTOR LOOP in cny_saxpy at 5: L 2 S 1 B 2 U 0 I 0 M 0
```

Auto-vectorization of loops

The saxpy routine can be compiled with the the `-mcny_auto_vector` flag, which enables automatic vectorization of loops.

```
$ cnycc -c -std=c99 -mcny_auto_vector saxpy.c
VECTORIZED STMT in saxpy0 at 4: L 2 S 1 B 2 U 0 I 0 M 0
```

In the vectorization reports above, the numeric value after each capital letter indicates the number of that type of operation the compiler produced:

L	vector loads
S	vector stores
B	binary vector operations (+, -, * …)
U	unary vector operations (unary - …)
I	vector intrinsic routines called
M	miscellaneous vector operations

In this case, the compiler produced two vector loads, a store, and two binary operations. Note that the compiler will attempt to combine an adjacent "+" and "*" operation into a single coprocessor FMA (fused multiply add) instruction, but the FMA instruction will show as two binary operations in the vectorization report.

Note that the above examples did not address the placement or migration of data, which is also required to maximize the effectiveness of the Convey coprocessor.

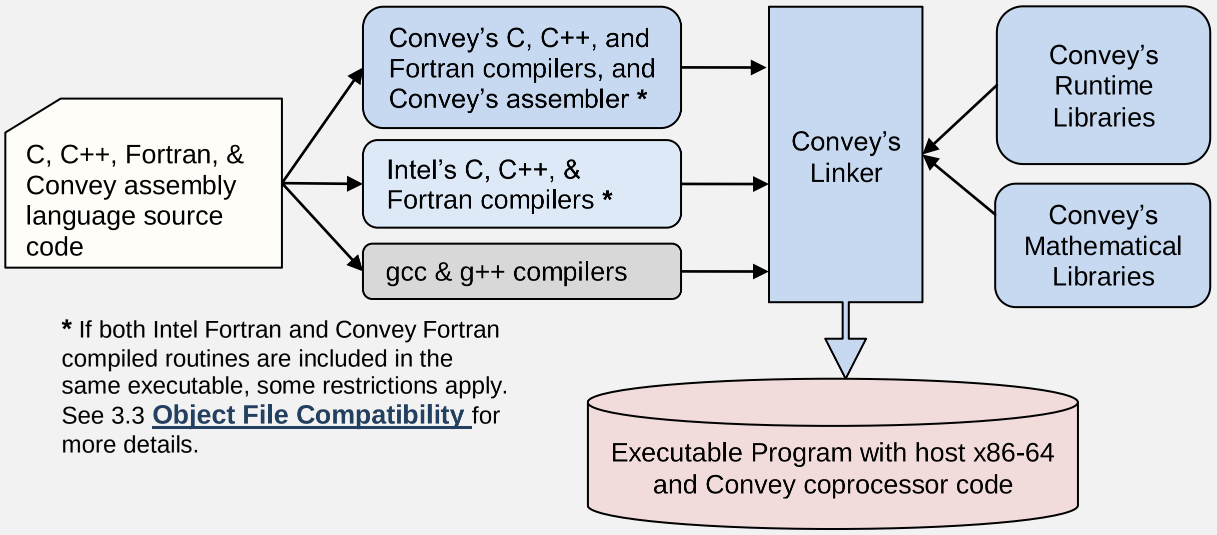
See chapter 4 **Porting Applications for the Convey Coprocessor** later in this document for more details as well as a description of the benefits and restrictions of each approach described above.

In addition to the examples above, there are two other approaches for utilizing the Convey coprocessor. They are:

- Calling a routine in the Convey Mathematical Libraries (see the **Convey Mathematical Libraries Guide**), and
- Developing a custom personality to provide part or all of the functionality provided by the equivalent host code (see the **Convey PDK Users Guide**)

2.4 Porting Applications to a Convey Hybrid-Core Server -- Overview

Applications may be developed on a Convey hybrid-core server or any x86-64 system with the Convey Software Development Kit installed (requires a compatible Linux OS)


C, C++, Fortran, & Convey assembly language source code
Convey's C, C++, and Fortran compilers, and Convey's assembler *
Intel's C, C++, & Fortran compilers *
gcc & g++ compilers
Convey's Linker
Convey's Runtime Libraries
Convey's Mathematical Libraries
Executable Program with host x86-64 and Convey coprocessor code

***** If both Intel Fortran and Convey Fortran compiled routines are included in the same executable, some restrictions apply. See 3.3 **Object File Compatibility** for more details.

The resulting program can be run on a variety of different systems, including non-Convey systems

Convey hybrid-core server (with attached coprocessor)

Coprocessor code is executed on the Convey coprocessor, or on the Convey simulator, or equivalent host code is executed on the x86-64 ‡‡

x86-64 Linux system with Convey Software installed

Coprocessor code is executed on the Convey simulator, or equivalent host code is executed on the x86-64 ‡‡

Generic x86-64 Linux system

Programs execute equivalent x86-64 host code instead of coprocessor code. Typically, no Convey specific libraries are required.

‡‡ under user control

Programs can be debugged with Convey's enhanced `gdb` debugger that handles both host and coprocessor code. The same debugging interface and capabilities are provided for code executed via the Convey simulator, on the Convey coprocessor, and on the host x86-64 processor.

The Convey simulator is appropriate for most program development and debugging. It does not require access to an actual coprocessor. Note that the Convey simulator is approximately four orders of magnitude slower than the actual coprocessor.

2.4.1 Brief Overview of the Convey Programming Model

This overview is intended to provide a suitable context for reading the rest of this guide. A more complete discussion of the programming model may be found later in this document, in chapter **4** **Porting Applications for the Convey Coprocessor**.

Key Points:

- The Convey coprocessor extends the host processor's (x86-64) instruction set. These new instructions can only be executed by the Convey coprocessor, and the host processor "dispatches" a coprocessor code segment, and typically waits for the coprocessor to "return". The extended instruction set is defined by a dynamically loadable "personality". Convey provides several personalities that can be used as is, or the user can develop their own personality.
- Within an application, the host processor and Convey coprocessor share a common, cache-coherent address space. Code executing on the host processor may dispatch coprocessor code, and may also call a coprocessor procedure passing in arguments and addresses. Coprocessor code has complete access to the application's address space.
- Coprocessor code cannot perform I/O, call OS kernel routines, call host routines, nor call Linux supplied library routines (most math intrinsics are allowed). Therefore, loops and routines that only contain numeric computations are the best candidates for executing on the coprocessor, assuming that the compiler can generate vector coprocessor code for those loops and routines.
- Although the host and coprocessor share a common address space, physical memory is present on the host processor's motherboard and on the coprocessor board. The coprocessor can access its own memory faster than host memory, and the host processor can access its own memory faster than coprocessor memory.

One or more of the following 3 approaches can be used to port an application to use the Convey coprocessor.

- Use Convey's compilers to generate coprocessor code, for those routines, or portions thereof, that will benefit from executing on the Convey coprocessor, using one of Convey's vector personalities (i.e. single precision vector, double precision vector …). Coprocessor code can be generated by
  - adding directives/pragmas to the source code to indicate which routines, or portions thereof, that might execute faster on the Convey coprocessor,
  - by specifying compiler flags to compile entire routines for the Convey coprocessor, or to automatically vectorize loops in those routines, and
  - by writing coprocessor assembly language routines.

  Directives are also provided to allocate static memory in host and coprocessor memory, and to migrate memory pages between the host and coprocessor.
- Substitute Convey's Mathematical Libraries to replace another high-level mathematical library (does not necessarily require re-compilation or re-linking). Many Convey unique routines are also available. See the **Convey Mathematical Libraries Guide** for more information.
- Develop a custom personality for the FPGA's on the Convey coprocessor, implementing one or more custom instructions. These new instructions can be called from C, C++, Fortran, or Convey assembly language routines. See the **Convey PDK Reference Manual** for more information.

3 Convey's Software Products for Programmers

Convey provides the following software development products for the Convey hybrid-core server. The following products support both the x86-64 and Convey coprocessor architectures.

- C, C++, and Fortran compilers
- Convey Math Libraries
- Convey Assembler (Linux assembler enhanced with coprocessor support)
- Convey Linker (Linux linker enhanced to support coprocessor sections and alignment requirements)
- Convey Debugger (enhanced gdb)

In addition, the `spat` performance analysis tool provides detailed insight into the performance of coprocessor code.

These products are provided in both `.rpm` and `.deb` packages on the Convey website, conveysupport.com. Some of the packages require a valid Convey Software License for that product to be used on a system.

The various Convey enhanced software products are typically installed in `/opt/convey`. After installing the initial Convey software package (and starting up a new shell/terminal), the `PATH` environment variable for users of `ksh`, `bash`, `csh`, `tcsh`, and related shells should automatically include `/opt/convey/bin`.

On cross-development systems (customer supplied Linux systems), `/opt/convey/bin`, by default, will appear after `/usr/bin` and `/bin` in the `PATH` environment variable. The system administrator of a cross-development system can execute the script `/opt/convey/sbin/preferoptconveybin` to permanently move `/opt/convey/bin` before `/usr/bin` and `/bin` in the `PATH` environment variable.

On Convey servers, `/opt/convey/bin` will appear before `/usr/bin` and `/bin` in the `PATH` environment variable.

Note that only *login* shells will automatically set the `PATH` environment variable to include `/opt/convey/bin`.

If your `PATH` environment variable does not include `/opt/convey/bin`[1], please add it. The Convey compilers, the Convey assembler (`as`), the Convey linker (`ld`), the Convey debugger (`gdb`), and the simulator performance analyzer tool `spat` are typically installed in `/opt/convey/bin`.

[1] possibly because **PATH** is set to a specific value in **~/.profile**, **~/.bashrc**, **~/.kshrc**, **~/.zshrc**, or **~/.cshrc**, or you are using a non-login shell (the *terminals* that startup on a graphical desktop on many Linux distributions are not login shells)

Note that the Convey assembler, linker, and debugger (`as`, `ld`, and `gdb`) have the same names as the standard Linux® programs, but are in a different directory. Since the default `PATH` environment variable on Convey Servers includes `/opt/convey/bin` before `/usr/bin`, any existing Makefiles, scripts, or commands entered by the user that invoke `as`, `ld`, or `gdb` directly will use the Convey version of that tool. Whenever the original version of `gdb`, `as`, or `ld` is required, you should invoke it with a fully qualified path (i.e. `/usr/bin/as`) or update your `PATH` environment variable to put `/opt/convey/bin` after `/bin` and `/usr/bin`. The Convey compilers will always use the Convey assembler and linker, regardless of the directories in the `PATH` environment variable.

Two scripts are included in `/opt/convey/bin` to modify the `PATH` environment variable for users that want to use the Convey versions of the assembler, linker, and gdb debugger without having to specify the full pathname. These scripts add `/opt/convey/bin` ahead of the system default paths (`/bin` and `/usr/bin`).

bash, **sh**, and **ksh** shell users can execute the command

```
. /opt/convey/bin/convey.sh
```

while **csh** and **tcsh** users can execute the command

```
source /opt/convey/bin/convey.csh
```

These commands can also be run from `.bashrc`, `.kshrc`, `.cshrc`, `.tcshrc`, `.profile`, or `.login` as appropriate.

The Convey runtime libraries are typically installed in `/opt/convey/lib`, and the Convey linker automatically includes this directory in the library search path when linking a program.

Personality related files are typically installed in `/opt/convey/personalities`. The system administrator and each user can configure some aspects of personality signature selection as described in the Convey System Administration Guide and later in this document, in chapter 14.3.3 Controlling Signature Resolution below.

On a Convey HC-1 server, the usual Linux software development tools are also available:

- Text editors (`emacs`, `vi`, …)
- Version management tools (`svn`, `RCS`, …)
- Profilers (`prof`, `gprof`, …) (these only support the host x86-64 processor on a Convey hybrid-core server)

3.1 Convey Compilers and Assembler

The following table lists the default preprocessor (if any) and the presumed source code language, based on a filename's suffix. The selected preprocessor can be overridden by explicit compiler flags.

`.c`	C preprocessor, C source code
`.C`, `.cc`, `.cpp`, `.cxx`	C preprocessor, C++ source code
`.f`	No preprocessor, Fortran fixed-form source code

`.f90`, `.f95`	No preprocessor, Fortran 95 free-form source code
`.F`	C preprocessor, Fortran 95 fixed-form source code
`.F90`, `.F95`	C preprocessor, Fortran 95 free-form source code
`.s`	No preprocessor, assembly language source code (for the x86, x86-64, or Convey coprocessor architectures)

3.1.1 cnycc

`cnycc` is Convey's C compiler. It is ISO/IEC 9899:1990 (C89/C90) compliant, ISO/IEC 9899:1995 (C95) compliant, and has partial support for ISO/IEC 9899:1999 (C99).

Key features:

- `cnycc` is source code compatible with the GNU C compiler (`gcc`) v4.2
- `cnycc` is binary (calling sequence) compatible with `gcc` 4.2
- Supports OpenMP 2.5

3.1.2 cnyCC

`cnyCC` is Convey's C++ compiler. It implements most of ISO/IEC 14882:1998 (C++98), and many of the changes in ISO/IEC 14882:2003 (C++03).

Key features:

- `cnyCC` is source code compatible with the GNU C C++ compiler (`g++`) v4.2
- `cnyCC` is binary (calling sequence) compatible with `g++` 4.2, including name mangling
- Supports OpenMP 2.5

3.1.3 cnyf95

`cnyf95` (also accessible as `cnyf90`) is Convey's Fortran compiler. It is ISO/IEC 1539:1997 (Fortran 95) compliant, and contains limited support for ISO/IEC 1539:2004 (Fortran 2003). Regardless of the filename suffix of the source file being compiled, `cnyf90` and `cnyf95` always provide full Fortran 95 support. Note that the Fortran 90 features removed in the Fortran 95 standard are still supported by Convey's Fortran compiler.

Key features:

- All common industry standard extensions to Fortran 77, including Cray Pointers, are supported
- With a few minor exceptions, `cnyf95` is also compliant with the Fortran 77 and Fortran 90 standards. These exceptions are required by the Fortran 95 standard, and are documented in the Conformance section (1.5) of the Fortran 95 standard.
- Applications compiled with Intel's Fortran (and C/C++) compiler may be linked with some Convey compiled Fortran routines (see chapter 3.3.2 **Compatibility with Intel and GNU Products** for a list of limitations), providing a quick and easy way to port a large existing application to take advantage of the Convey coprocessor.
- Support for OpenMP 2.5

3.2 as – Convey's Assembler

Convey's assembler supports the x86, x86-64, and Convey's coprocessor architectures. Convey recommends using `cnycc` to compile all assembly language routines.

```
$ cnycc  -cpp  -c  file1.s  file2.s …
```

The `–cpp` flag causes the assembly language source code files to be processed by the C preprocessor before passing them on to the Convey assembler.

The Convey Reference Manual describes the coprocessor's scalar instruction set shared by all personalities, and the vector instruction sets for Convey's single and double precision personalities.

3.3 Object File Compatibility [2]

3.3.1 Compatibility with Open64 and PathScale Products

Convey's compiler suite is compatible with Open64's compiler suite and PathScale's compiler suite, except for the following:

- Convey's Fortran compiler implicitly sets two flags for every Fortran source file compiled, to improve compatibility with Intel's compilers:
  - `-ff2c-abi`, and
  - `-fno-second-underscore`
  - To be more compatible with Open64 and PathScale, `-fno-f2c-abi` or `-fsecond-underscore` may be specified
- Convey's OpenMP runtime support is not compatible with any other vendor's OpenMP runtime support. When compiling source code containing OpenMP directives, with both Convey's and another vendor's compiler, the `–openmp` flag (or equivalent) should only be used with either Convey's compilers or the other vendor's compilers, and the application should be linked with the appropriate OpenMP library.
- Convey Fortran `.mod` files are not compatible with any other vendor's `.mod` files

3.3.2 Compatibility with Intel and GNU Products

Object files produced by newer releases of the C and C++ compilers from Convey, GNU, and Intel may be freely mixed.

Mixed language applications that use both Convey and non-Convey compilers are subject to the following restrictions:

- Objects compiled with Intel's or GNU's C++ compiler may be linked with objects compiled with Convey's C++ compiler with the following restrictions:
  - templates, exception support (`try`/`catch`), and OpenMP runtime support routines are not compatible
- Objects compiled with Intel's Fortran compiler may be linked with objects compiled with Convey's Fortran compiler with the following restrictions:
  - Fortran I/O will require some explicit libraries to be linked in, to satisfy I/O library routines for both vendors' I/O libraries.

[2] Convey has tested object file compatibility with a number of example programs and other vendors compilers, but other undocumented incompatibilities may exist.

- routines compiled by Intel and Convey Fortran compilers can both execute Fortran **`WRITE`** statements for the “*” unit. More complicated I/O, such as **`READ`**s or **`WRITE`**s to files that were **`OPEN`**ed, direct access files, unformatted sequential access files, and pipes, is likely to produce incorrect results or internal library errors.
- Fortran MODULEs cannot be shared between Intel and Convey compiled objects files.
- When a subprogram and the routine that invokes it are compiled by different compilers, any dummy arguments for that called routine may not
  - be assumed shape arrays, or
  - be variables with the **`POINTER`** or **`ALLOCATABLE`** attribute.

  Dummy arguments with these characteristics require passing an array descriptor, instead of a simple address. Convey’s array descriptors are not compatible with any other vendor’s array descriptors, except Open64/PathScale’s. Note that dummy arguments with the prohibited characteristics are exactly those dummy arguments that require an explicit visible interface for the called routine when that routine is called.

  If an actual argument itself:
  - is an assumed shape array,
  - has the **`POINTER`** attribute, or
  - is a potentially non-contiguous array expression,

  and the corresponding dummy argument is not one of the prohibited types listed above, then the compiler will:
  - create a contiguous array temporary,
  - copy the actual argument into the temporary,
  - pass the simple address of that temporary, and
  - possibly copy the temporary back into the original actual argument when the called routine returns.

  Although very inefficient for large arrays, this calling mechanism is compatible between Convey’s Fortran compiler and other vendor’s Fortran compilers.
- Common blocks shared by object files produced by different compilers may not contain an object that:
  - has the **`POINTER`** or **`ALLOCATABLE`** attribute
  - has a complicated derived type (a simple sequenced derived type is likely to work as expected)
- If present, compiler options that control implicit underscores added to external symbol names must be compatible
- If present, compiler options that control sizes of intrinsic types must be compatible

4 Porting Applications for the Convey Coprocessor

Porting an application to take advantage of the Convey coprocessor requires using one or more of the following approaches:

- Use the standard math solvers in the Convey Mathematical Libraries.
  See the **Convey Mathematical Libraries Guide** for a description of the available routines and how to call them.
- Compile one or more routines with the appropriate Convey compiler(s), using one or more of the following capabilities:
  - using automatic vectorization to automatically select and vectorize DO/for loops for execution on the coprocessor, using one of Convey's vector personalities
  - inserting directives/pragmas in the source code to explicitly indicate which parts of a routine should execute on the coprocessor
  - compiling an entire routine (or set of routines) for execution on the coprocessor
  - writing a coprocessor assembly language routine, and calling it from C, C++, or Fortran
  - writing a coprocessor assembly language routine that the compiler treats as an *intrinsic routine*. Calls to such a routine would contain scalar arguments, inside a loop/loop nest, and the compiler would vectorize calls to the *user-defined intrinsic* just as it does for language defined intrinsic such as *sin*.
- Develop a custom personality (instruction set) for the coprocessor, and execute those custom instructions within a C, C++, Fortran, or coprocessor assembly language routine.

  See the **Convey PDK Reference Manual** for instructions on developing and deploying a custom personality.

  A custom personality allows for greater performance gains at the cost of additional specialized development effort, namely, designing one or more new instructions for the Convey Coprocessor and implementing the FPGA image that executes those instructions.

See Appendix A **How To Take Advantage Of The Convey Coprocessor** for a discussion of the advantages and tradeoffs of each approach.

4.1 Porting Existing Applications Overview

When porting an existing application previously built with the Intel, PathScale, Open64 or gcc/g++ compilers, Convey recommends that only those portions of the application that will benefit from executing on the coprocessor be recompiled with Convey's compilers, although other routines may also have to be compiled with Convey's compilers due to object file compatibility issues. Recompiling a minimal set of routines with Convey's compilers may reduce the porting effort significantly as well as reduce the chances of introducing changes in behavior.

Compiling an application using a profiling flag such as `-pg` (see the man page for `gprof`) can be useful in identifying hot spots in the application that might be good candidates for execution on the Convey coprocessor.

Convey hybrid-core servers are capable of running applications entirely on the host processor. In order to improve the performance on an application, portions of the application must be ported to run on the Convey coprocessor. The coprocessor supports multiple personalities, each of which is suitable for a particular set of operations. An application can use one or more personalities to improve performance of a few key routines.

Convey Provided Vector Personalities

Personality Nicknames	Description
single_precision, sp, single, single_vector, sp_vector	a general purpose single precision floating point vector arithmetic personality
double_precision,dp, double, double_vector, dp_vector	a general purpose double precision floating point vector arithmetic personality
financial, fap, finance, fin	a specialized double precision personality, with additional support for fast vector random number generation (mersenne twister), and a few fast intrinsics (sqrt, e^x, sine, cosine, log, reciprocal approximation)

The Convey compilers can generate coprocessor code for these personalities, both automatically and via directives.

The Convey Math Libraries also support these three vector personalities. See the **Convey Mathematical Libraries Guide** for more information.

Convey's vector personalities have the following characteristics that affect the performance of applications that utilize the coprocessor:

- The coprocessor's vector instructions typically have a modest startup cost (before producing the first result), and then produce additional results relatively rapidly. Therefore, long vector lengths are key to improving performance.
- The results of one vector instruction can often feed the next vector instruction, as soon as the first vector element is available. Therefore, code segments containing many vector instructions are more likely to improve performance.
- Although the coprocessor supports both a scalar and a vector instruction set, coprocessor scalar instructions are significantly slower than their corresponding host processor equivalents. The coprocessor scalar instructions are provided to enable small amounts of scalar code to be mixed with vector instructions to allow larger code segments to execute entirely on the coprocessor.
- Since dispatching a coprocessor code segment has some overhead, small coprocessor code segments (those with just a few vector instructions) may not provide a performance advantage compared to host code.
- The vector personalities include instructions or library routines for all basic arithmetic operations (+, -, *, /, x^y), for the indicated type of floating point data, and for 64 bit integers.

- Vector versions of most language provided floating point intrinsic procedures are provided, enabling vectorization of loops containing these operations (including many trig functions, sqrt, exp, $\log_e$, $\log_{10}$ …). Vector versions of some integer intrinsics are also provided, including modulo and some type conversion functions.
- Users can also define and implement their own user-defined intrinsic routines, similar to the language defined floating point intrinsics like *sin*, *cos*, …
- Switching personalities, either to a completely different personality, or to a different version of a particular personality, is a slow operation. Frequent personality switching within an application should be avoided.

Therefore, loops that perform either single or double precision array computations, particularly on large arrays, are usually the best candidates for execution on the Convey coprocessor using one of Convey's personalities. Note that a loop containing both single precision and double precision array references will not vectorize.

A custom personality can provide one or more user defined custom instructions. Custom personalities also provide the same scalar instruction set provided by the vector personalities, as well as the additional custom instructions. See the **Convey PDK Reference Manual** for instructions on developing and using a custom personality.

4.2 Host Code and Coprocessor Code Interactions and Limitations

An application that has been written or ported for use on a Convey hybrid-core server will contain host code, as well as some coprocessor code. Below is a summary of how host and coprocessor code may interact.

- The host processor and the Convey coprocessor share a common physical and virtual memory space that is cache coherent. Therefore, data addresses (such as arguments passed by address/pointer, and addresses of common blocks or **`extern`** variables …) may be freely passed from host code to coprocessor code.
- Host code can dispatch a coprocessor code segment or call a coprocessor routine, but coprocessor code cannot call a host routine. Therefore, any routines that are compiled "dual-target" should only call other procedures that were either compiled "dual-target" or have the coprocessor version of that procedure provided via a compiler flag (`-mcny_pure`) or an assembly language procedure.
- Coprocessor code can call other coprocessor routines (subject to the coprocessor stack limit of 8 active coprocessor routines, including Convey's mathematical intrinsics).
- Coprocessor code cannot perform I/O. Convey provides a low level message mechanism (for tracing purposes) that lets coprocessor code queue up messages to be processed when the coprocessor routine returns to its host caller (see **Message Routines**).
- Coprocessor code cannot call any host library routines, such as libc routines.
- Coprocessor code cannot call any Linux kernel interfaces (system calls).
- For Convey's vector personalities, coprocessor versions (scalar and vector) of all the standard mathematical intrinsics for C, C++, and Fortran (many of the libm routines) are provided. These include trig functions, sqrt, $\log_e$, $\log_{10}$, exp, and

type conversion functions. See Appendix E **Coprocessor Intrinsics** for a list of supported intrinsics.

- Although the host architecture supports a strongly ordered memory model, the coprocessor supports a weakly ordered memory model. From the assembly language programmer's point of view, this means that instructions that write to memory actually queue up the write requests. Those requests are not necessarily processed in the same order as the instructions that were executed might imply. Lastly, the coprocessor hardware does not guarantee[3] that a store to a memory location will complete before a subsequent load of that location loads the value into a coprocessor register. If you are writing assembly and need to make sure a write is complete before a later load, see **Fence Checking with the Convey Simulator and Controlling Compiler Generated Fences**.
- Applications entirely written in C, C++, and Fortran, including those that call routines in the Convey Mathematical Libraries, are insulated from the weakly ordered memory model. The compiler inserts `FENCE` instructions to preserve ordering semantics based on compile time analysis of the vectorized loop.

  User control over FENCE instruction generation is described in **Fence Checking with the Convey Simulator and Controlling Compiler Generated Fences**.
- Applications that call coprocessor assembly language routines from within a coprocessor region need to understand these issues and protect certain memory operations with a `FENCE` instruction.
- A dispatch of a coprocessor routine from host code provides an implicit `FENCE` operation as the dispatch completes. See the **Convey Reference Manual** (Memory Model) for a detailed discussion of the weakly ordered memory model.

4.3 Support for Portable Applications

One of the key features of the Convey compiler suite and Mathematical Libraries is the ability to speed up an application by utilizing the Convey coprocessor, while still being able to run that same executable on any compatible Linux system without a Convey coprocessor. This portability is provided by a number of mechanisms that always produce equivalent host code sequences for any coprocessor code produced by the compiler. These mechanisms are:

- Coprocessor regions – regions of code within a procedure that have both host code and coprocessor code versions on the region. Coprocessor regions can be created with directives/pragmas embedded in the source code, or by enabling automatic vectorization, for Convey's vector personalities.
- Dual-target routines – routines that have both host code and coprocessor code versions of that same routine. The `-mcny_dual_target` compiler flag or the `dual_target` directive/pragma can be used to create dual-target routines. Note that the `-mcny_dual_target` flag is equivalent to compiling with `-mcny` and placing a `dual_target(1)` directive/pragma at the top of the file. The coprocessor version of the routine is named `cny_xxx`, where `xxx` was the original routine name, so the two versions can coexist in the same executable. When the coprocessor version of a dual-target routine calls another routine, it prepends `cny_` to the procedure name.

[3] The coprocessor usually forces such a load operation to wait until the store is complete, except when different data paths in the coprocessor are used for the store and load operations.

The host version of a dual-target routine resulting from compiling with the `-mcny_dual_target` flag includes a *dispatch-wrapper*. A *dispatch-wrapper* causes the host version of a routine to dispatch the coprocessor version of that routine, if the coprocessor is attached to the current process/process group.

Dual-target routines can also be created with the `-mcny_dual_target_nowrap` flag or `dual_target_nowrap` directive/pragma. The `dual_target` and `dual_target_nowrap` directives/pragmas and flags differ in the sense that `dual_target` compiled routines include a dispatch-wrapper while `dual_target_nowrap` compiled routines do not include this wrapper. Routines compiled with `dual_target_nowrap` can only have their coprocessor versions called from other coprocessor code. See the **dual_target directive/pragma** for more details.

- The Convey assembler supports both host and coprocessor assembly language. An application can provide both host and coprocessor routines with the same apparent name and the appropriate one will be called as needed, via the dual-target routine `cny_` naming convention.
- The Convey runtime startup code queries the presence, or lack thereof, of the coprocessor, which allows the application to execute the host version of all coprocessor regions when a coprocessor is absent or cannot be attached.
- Lastly, the Convey Mathematical Libraries utilize these mechanisms to ensure that an application linked with the Convey Mathematical Libraries will run on any compatible x86-64 Linux system, subject to the constraints listed below.

4.3.1 Running Portable Applications that Utilize the Convey Coprocessor on a non-Convey System

An application that utilizes the Convey coprocessor can also run on any non-Convey system (hereafter refered to as the target system) with an x86-64 processor and a compatible Linux distribution, without a Convey coprocessor, subject to the following constraints:

- Any runtime **shared**[4] libraries required by the application must be installed, including:
    - Vendor provided compiler support shared libraries, for I/O, memory management, MPI support, math routines, etc. For example, if an application is compiled with Intel's C or Fortran compilers, the appropriate Intel runtime libraries need to be installed on the target system.
    - An application that uses one or more of the features/products listed below *may* require one or more of Convey's runtime libraries to be installed:
        - Convey's Mathematical Library routines that are not part of Intel's MKL library or some other 3rd party shared library
        - Explicit calls to personality support routines, coprocessor exception mask routines, or some low level coprocessor support routines

          Calls to the memory migration routines are silently ignored if the underlying Convey library is not installed
        - Explicit calls to `copcall` routines and coprocessor messaging routines, **if** the application requires executing code on the Convey simulator to produce correct behavior

          Calls of the `copcall` and coprocessor messaging routines should be enclosed in a runtime check for the coprocessor, i.e.

          ```
                  if (cny$coprocessor_ok)…
          ```
          to avoid a runtime abort if the coprocessor is not available.
- There are several situations where Convey runtime shared libraries do not need to be installed on a non-Convey system:
        - The Convey Mathematical Libraries (CML) are not required if all the CML library routines called are calling sequence compatible with the same named routine in a 3rd party library, such as Intel's MKL, and that 3rd party library is loaded and is specified in the `LD_PRELOAD` environment variable when the application is executed.
    - The C and C++ runtime packages for the target system's Linux distribution must be installed. These packages must be upwardly compatible with `gcc` 3.2 and GNU™ `libc` 2.5.
    - Any application provided or 3rd party shared libraries that were linked with the application will need to be installed on the target system.

[4] Static libraries (named with the suffix **.a**) used by an application do not need to be installed to run that application. Convey's Fortran compiler uses static libraries to support all standard Fortran language features, including I/O and the coprocessor versions of all intrinsic procedures. Similarly, Convey's C/C++ compilers use static libraries for the coprocessor versions of all libm procedures.

    - The Linux distribution's `libm` and `libc` libraries must be installed and must be compatible with the application (any compatible Linux distribution should automatically provide these).

- Coprocessor assembly language routines may be present, but they should ONLY be called from a coprocessor region, and equivalent host versions of the coprocessor routines must be available (host versions without the `cny_` prefix). These restrictions ensure the coprocessor assembly language routines won't ever be referenced during execution of the application.
- If any coprocessor code in the application is going to be executed, either on the coprocessor or the simulator, only Convey's `gdb` can be used to debug coprocessor code.

4.4 How to Customize Source Code for Convey's Compilers

Convey's C and C++ compilers always define `__CONVEY`, for use with `cpp` preprocessor directives. Convey's Fortran compiler also defines `__CONVEY` when the `-cpp` flag is specified.

4.5 Pointer Aliasing Model and Vectorization

Convey's compilers support multiple pointer aliasing models (hereafter referred to as *aliasing models*). These models affect the code produced by the compilers by changing what assumptions the compiler makes about pointers and under what conditions two different pointers or pointer expressions may reference the same memory location.

Note that an application compiled with an inappropriate aliasing model may produce incorrect answers.

4.5.1 Default Alias Models

Convey's C, C++, and Fortran compilers each have their own default aliasing model. Those default aliasing models are unique to each language, and conform to the requirements imposed by each language's ISO/ANSI standard.

- **c89**, the C compiler's default aliasing model makes very few assumptions about pointers. In general, the compiler assumes different pointers and pointer expressions can reference the same memory location. This default aliasing model and the somewhat more aggressive **typed** model are usually not appropriate when you want the coprocessor version of a region to be vectorized.

  When compiling C/C++ code and attempting to vectorize a region for execution on the coprocessor, the **c99**, **restrict** and **disjoint** aliasing models are more likely to produce a vectorized region. You should only use these aliasing models when the source code conforms to the assumptions for the aliasing model used.

- **fortran-default**, the Fortran compiler's default aliasing model, is safe, standard conforming, and usually appropriate when vectorized coprocessor regions are desired. Dummy arguments that aren't `TARGET`s or `POINTER`s are assumed to not overlap any other dummy arguments or variables in `COMMON`. When multiple Fortran 90 variables with the `POINTER` or `TARGET` attribute appear in a loop, the more aggressive **no_f90_pointer_alias** model may be required, assuming the loop/routine conforms to the assumptions of that aliasing model.

4.5.2 Additional Aliasing Models

The additional aliasing models listed below allow the compiler to make other assumptions about pointer usage in an application, typically allowing for more code optimizations and better vectorization.

The Convey compilers support the following additional aliasing models, listed in order of least aggressive to most aggressive optimization assumptions:

Aliasing Model Name	What the compiler assumes
`c89`	The compiler assumes that different pointers (and pointer expressions) can point to the same memory location.

`typed`	The compiler assumes that pointers of different types cannot point to the same location in memory.
	For example, given:
	`int   * p;`
	`float * q;`
	the compiler assumes `p` and `q` never point to the same memory location.
	Two different pointer expressions are assumed to possibly reference the same memory location.

`c99`	The compiler assumes that ANSI C99 pointer aliasing rules apply: • `void *` and `char *` pointers can alias anything • any other two distinct pointer types are assumed not to be aliases (similar to **restrict**)
`fortran-default`	The compiler assumes that dummy arguments without the `TARGET` or `POINTER` attribute cannot overlap (if either is modified in a routine). Two variables with the `TARGET` or `POINTER` attribute, of the same base type, are assumed to possibly reference the same memory location.

Aliasing Model Name	What the compiler assumes

`restrict`	The compiler assumes that distinct pointers cannot point to the same memory location, i.e. two pointers, even those of the same type, cannot point to the same memory location.
	
	For example, given:
	
	`float * p;`
	`float * q;`
	
	the compiler assumes `p` and `q` never point to the same memory location.

`disjoint`	The compiler assumes that all pointers and pointer expressions point to different memory locations
	
	For example, given:
	
	`float * p;`
	`float * q;`
	…
	`*q = p[i] + p[j];`
	
	the compiler assumes `p[i]`, `p[j]`, and `*q` never point to the same memory location.
	
	This is **not** standard conforming and some valid programs will produce wrong answers when compiled using this aliasing model.

`no_f90_pointer_alias`	The compiler assumes that all variables with the `POINTER` or `TARGET` attribute reference distinct non-overlapping memory locations. This is **not** standard conforming and some valid programs will produce wrong answers when compiled using this aliasing model.

Different source files in an application may be compiled with different aliasing models.

If your application contains source code that violates the assumptions of the default aliasing model, you may need to compile those source files at a lower optimization level (`-O1` or `-O0`).

4.5.3 Specifying An Aliasing Model

There are several compiler flags that affect the aliasing model:

- Language dialect flags (`-std=c89`, `-std=c99`)

  These C/C++ flags affect both the aliasing model and which dialect of C the compiler will accept. Not all C programs will compile when `-std=c99` is used.

- Alias model flag (`-OPT:alias=`*`model`*)

The `-OPT:alias=`*`model`* flag affects both host and coprocessor code generation.

- Coprocessor only alias model flags (`-mcny_alias_c89`, `-mcny_alias_c99`)

  These flags only affect the aliasing model used when generating coprocessor code for a region. The corresponding host code for that region (and all host code not in a region) will be compiled using the default aliasing model or the aliasing model specified in a `-OPT:alias` flag. These flags are particularly helpful when some parts of a C/C++ routine that will only execute on the host processor require the **typed** aliasing model, but the regions (loops) that you want to execute on the coprocessor (using vector instructions) can be safely compiled with the **restrict** aliasing model. `-mcny_alias_c99` enables the **restrict** aliasing model for coprocessor code generation.

Note that the **fortran-default** aliasing model cannot be explicitly specified via a compiler flag.

The following table lists those compiler flags that affect the aliasing model, grouped by aliasing model.

Compiler Flag	Alias Model	Effect
(no aliasing flag) or -std=c89	**c89** for the C/C++ compilers **fortran-default** for the Fortran compiler (for both host and coprocessor code)	The default aliasing models are the safest aliasing models. Many Fortran codes can be vectorized using the default aliasing model, but C/C++ codes usually need to be compiled with the **c99**, **restrict** or **disjoint** aliasing models in order to vectorize loops in coprocessor regions. When the –std=c89 flag is specified, the C compiler will only compile ANSI C 89 compliant source code.
`-mcny_alias_c89`	**typed** (for coprocessor code)	Coprocessor code will be generated using the **typed** aliasing model.
`-OPT:alias=typed`	**typed** (for both host and coprocessor code)	Host and coprocessor code will be generated using the **typed** aliasing model.
`-OPT:alias=restrict`	**restrict** (for both host and coprocessor code)	Host and coprocessor code will be generated using the **restrict** aliasing model.

-std=c99	**restrict** (for both host and coprocessor code)	In addition, when `-std=c99` is specified, the C compiler will only compile ANSI C 99 compliant source code.
`-mcny_alias_c99`	**restrict** (for coprocessor code)	Coprocessor code will be generated using the **restrict** aliasing model.
`-OPT:alias=disjoint`	**disjoint** (for both host and coprocessor code)	Host and coprocessor code will be generated using the **disjoint** aliasing model.
`-OPT:alias=no_f90_pointer_alias`	no_f90_pointer_alias	Host and coprocessor code will be generated using the **no_f90_pointer_alias** aliasing model.

4.6 Coprocessor Routines and Regions

The Convey compilers can generate coprocessor code for an entire routine, or for a region within a routine, such as a vectorizable loop nest. Typically, whenever coprocessor code is generated for a region or a routine, host (x86-64) code is also generated. The host code checks to see if the coprocessor is attached to the current process/process group, and if it is, the host code will dispatch the corresponding coprocessor code for that routine or region. If the coprocessor is not attached, the host code corresponding to that region or routine is executed instead.

Coprocessor code can be generated for entire routines with the following compiler flags and their equivalent directives and pragmas:

Compiler flag	Directive/pragma
`-mcny_dual_target`	`dual_target(x)`
`-mcny_dual_target_nowrap`	`dual_target_nowrap(x)`
`-mcny_pure`	No equivalent

The `(x)` argument must be

- In C/C++ code, a constant expression evaluating to 0 or 1,
- In Fortran code, a literal or named logical constant (`.true.`, `.false.`)

If `(x)` is zero/`.false.`, the mode is disabled; if one/`.true.`, the mode is enabled, for the rest of the source file, until another similar directive/pragmas appears.

Part of a procedure (a region) may be compiled for execution on the Convey coprocessor (or just the host) with the following directives/pragmas and compiler flags:

Compiler flag	Directive/pragma
-mcny_auto_vector	begin_coproc and end_coproc
No equivalent	begin_host and end_host

The -mcny_auto_vector flag also enables vector code generation for coprocessor regions.

Choosing the best granularity for coprocessor code generation is critical to maximizing performance of your application. See Appendix A How To Take Advantage Of The Convey Coprocessor for more information.

4.7 Naming Conventions for Routines in Coprocessor Regions

The Convey compilers use the procedure naming conventions described below for generating equivalent routines callable by either host or coprocessor code. When all of the source code for an application is written in C, C++, and Fortran, the application developer can ignore these naming conventions, except that routine names that begin with "**cny_**" should be avoided, particularly for routines that are compiled with the -mcny_dual_target, -mcny_dual_target_nowrap, or -mcny_pure flags[5].

When one of the Convey compilers is generating coprocessor code (within a coprocessor region), and a procedure call (a function call or a Fortran subroutine call) is encountered, the compiler generates a reference to cny_xxx, where xxx was the original procedure name. Similarly, when the compiler is compiling an entire routine and the -mcny_dual_target, -mcny_dual_target_nowrap, or -mcny_pure flag was specified, the coprocessor version of that routine is named cny_xxx, where xxx was the original procedure name. The host version of that routine retains the original name. This procedure renaming allows the host and coprocessor versions of an external routine to co-exist, and both may be referenced by their original name. This procedure renaming mainly affects developers of Convey assembly language source code.

For C++ compilations, the internal name is appropriately modified by inserting cny_ into the mangled name. For example, a routine named foo will generate, for dual_target compilation, two internal routines named:

```
_Z3foov
_Z7cny_foov
```

4.7.1 Assembly Language Routine Names

If a routine written in Convey assembly language is called from a compiler generated coprocessor region, then the assembly version of that routine must declare the external name of the routine with the "cny_" prefix.

Note that procedure names that appear as an argument in one of the low level copcall interface routines are not subject to renaming, and the cny_ prefix must be provided explicitly.

[5] An application can contain a routine named **cny_xxx**, as long as it does not also have a routine named **xxx**. If both names exist, and the routine named **xxx** is compiled with -mcny_dual_target (or a similar) flag, the linker will fail when trying to link the application.

4.8 User-Defined Intrinsics

A function call may be treated as a vectorizable intrinsic. The user is responsible for writing the intrinsic in coprocessor assembly language, and specifying the intrinsic with the compilers `-mcny_vec_rtn` flag.

The user defined vector intrinsic must return a vector result in the v0 register. In the call to the vector intrinsic, all arguments to the original function are converted to vectors passed in as v0, v1, ..., vN.

All arguments to the original function must be scalar values.

Example compilation steps:

```
cnycc -c user_def.s

cnycc -mcny_sig=sp -mcny_vec_warn -mcny_vector \

                -mcny_vec_rtn:dsign=__vecdsign \

                -mcny_vec_rtn:rsign=__vecrsign \

       call_dsign1.c   user_def.o
```

Example File:

```
#include <stdlib.h>
#include <stdio.h>

    int b[10];
    float c[10];

#pragma cny dual_target_nowrap(1)
float dsign(int i, float j) { return i*j+1;}
int rsign(float i, int j) { return i*j+2;}

#pragma cny dual_target_nowrap(0)
int main() {
   int i,j;
   int n=10;

#pragma cny begin_coproc
  for (j=0;j<4;j++) {
   for (i=0; i<n; i++) {
     b[i] = dsign(i,1.0f + j) + rsign(1.1f+i,j);
   }
  }
#pragma cny end_coproc
   for (i=0; i<n; i++) {
      printf("%d\n",b[i]);
   }
   return 0;
}
```

Additional messages can be produced by the compiler when the

-`mcny_vec_warn option` is enabled. Acceptance of the compiler option can be verified by the presence of these messages:

```
Info: user defined vector function: dsign -> __vecdsign
Info: user defined vector function: rsign -> __vecrsign
Info: user defined vector function: dsign -> __vecdsign
Info: user defined vector function: rsign -> __vecrsign
```

The replacement of the user function call by the vector intrinsic can be verified by these messages:

```
"call_dsign1.c", line 20, Info: Replaced cny_dsign with vector
function __vecdsign
"call_dsign1.c", line 20, Info: Replaced cny_rsign with vector
function __vecrsignThe sample assembly file:
```

The intrinsic implementation in coprocessor assembly language follows:

```
$ cat user_def.s
        .section .ctext, "ax",@progbits
        .section .ctext
        .signature sp
        .weak   __vecdsign
        .type   __vecdsign, @function
__vecdsign:
        cvt.sq.fs   %v0,%v0
        mul.fs   %v0,%v1,%v0
        cvt.fs.sq   %v0,%v0
        add.sq   %v0,$1,%v0
        cvt.sq.fs   %v0,%v0
        rtn

        .weak   __vecrsign
        .type   __vecrsign, @function
__vecrsign:
        cvt.sq.fs   %v1,%v1
        mul.fs   %v0,%v1,%v0
        cvt.fs.sq   %v0,%v0
        add.uq   %v0,$2,%v0
        rtn
```

Note the `.signature` assembler directive is specific to the personality the intrinsic is designed for.

4.9 Compiler flags

The following table lists commonly used flags supported by Convey's compilers, grouped by function, and sorted alphabetically within each group, except that the "`no`" form of a flag appears with its non-`no` version (i.e. `-noinline` appears immediately after `-inline`). Less commonly used flags are documented in the Appendix C **Less Commonly Used Compiler Flags**.

Color-coding is used for a flag's textual description to indicate which languages support a particular compiler flag:

- Black: C, C++, and Fortran shared flags
- Purple: Fortran only flag
- Green: C/C++ only flags
- `Red:` Convey specific compiler flags, typically affecting coprocessor code generation, are supported by Convey's C, C++ and Fortran compilers

For users reading a monochrome copy of this document, the "`Lang`" column also indicates when a particular flag is only available for some languages. When the "`Lang`" column is empty, that flag is supported in C, C++, and Fortran.

In general, Convey's compilers are invoked in the same manner as other Linux compilers, and accept the same syntax and some of the flags used by other vendor's compilers, particularly those related to linking.

These flags are also described in the *man* pages for the various Convey compilers (`cnycc`, `cnyf95`, `cnyCC`).

Coprocessor Machine Model and Code Generation Flags		Lang

`-mcny`
 	
`-mno-cny`
	Enable coprocessor code generation for coprocessor regions (i.e. a `begin_coproc` … `end_coproc` block) in addition to generating x86_64 (host) code. Note that most flags that begin with `–mcny_` (i.e. `-mcny_vector`) also imply `-mcny`.
	
	Disable coprocessor code generation for coprocessor regions.	

`-mcny_alias_c89`
 	
`-mcny_alias_c99`
	Set alias model to c89 for vectorization (default)
	
	Set alias model to c99 for vectorization
	
	Important Note: the c99 aliasing model is usually required in order to vectorize a loop in C/C++ code.	

`-mcny_auto_vector`	Automatically select `coproc` regions and vectorize loops. This is intended as an advisory tool, to identify those loops that can be vectorized. This enables the user to analyze those loops to determine if they should be vectorized. Vectorizing all vectorizable loops will usually cause an application to run slower. The `-mcny_auto_vector` option also enables `-mcny_vec_info` which lists indicates the routine name and line number of loops vectorized.	

-mcny_dual_target	Compile all procedures for both the host processor and the Convey coprocessor. The host version of the routine will include a dispatch-wrapper at the beginning of the routine. The dispatch-wrapper will dispatch the coprocessor version of the routine if the coprocessor is attached by the current process/process group; otherwise, the host version of the routine is executed. See the **dual_target directive/pragma** for more details. Warning: The -mcny_dual_target flag should not be used to compile C++ files containing templates that are used in more than one file. Doing so will result in duplicate symbols at link time. This restriction may be removed in a future release.	
-mcny_dual_target_nowrap	Similar to -mcny_dual_target, but no dispatch-wrappers are generated. Warning: The -mcny_dual_target_nowrap flag should not be used to compile C++ files containing templates that are used in more than one file. Doing so will result in duplicate symbols at link time. This restriction may be removed in a future release.	
-mcny_pure	Only generate coprocessor code for a procedure. No entry points for the x86-64 code space are compiled. Procedures compiled with this flag cannot be called from host code, unless some other host version of the routine was provided elsewhere. The usual cny_ prefix will be pre-pended to routine names.	
-mcny_sig=xyz	Generate coprocessor code for the *xyz* personality. '*xyz*' may be a nickname (*sp, dp, pdk*) or a personality #. '*xyz*' may be a partial or fully resolved signature.	

`-mcny_vector`
 	
`-mno-cny_vector`
	Enable vectorization of loops within coproc regions and routines.
	
	Disable vectorization of loops within coproc regions and routines.
	
	When enabled, vector coprocessor code will be generated for regions that contain loops that were vectorized. If no loops vectorized, scalar coprocessor code will only be generated when the flag `-LNO:allow_vec2scalar=1` is specified. Coprocessor code will always be generated for routines compiled with `-mcny_dual_target`, even if no vectorization occurred.
	
	IMPORTANT NOTE: the `c99` aliasing model (or a more aggressive aliasing model) is usually required in order to vectorize a loop in C/C++ code.	

`-mcny_vec_info` or `-mcny_vector_info`
 	
`-mno-cny_vec_info` or `-mno-cny_vector_info`
	Enable the vectorization report. This flag is implicitly turned on by the `-mcny_vector` and `-mcny_auto_vector` flags.
	
	Disable the vectorization report.	

`-mcny_vec_warn` or `-mcny_vector_warn`
 	
`-mno-cny_vec_warn` or `-mno-cny_vector_warn`
	Emit warning messages for loops that fail to vectorize. Useful when trying to improve performance of loops when using a vector personality.
	
	Do not emit warning messages for loops that fail to vectorize. (default behavior)	

Coprocessor Code Generation Optimization flags		Lang
`-LNO:allow_vec2scalar=x`	When ***x*** is 1, allows a region (a `begin_coproc` block) to generate coprocessor code for `-mcny_vector` compiled code even when nothing in the region could be vectorized. When ***x*** is 0, no coprocessor code will be generated for a region unless a loop in that region is vectorized. The default is 1.	
`-LNO:migrate_advise=x`	When ***x*** is **1**, the compiler only advises migration for arrays involved in vector loads and stores. When ***x*** is **2**, the compiler will suggest other arrays for migration, including loop invariant arrays and arrays used in scalar coprocessor loops. The default is no migration advice is provided.	
General Flags (driver behavior, filenames, linker options, …)		Lang
`-ar`	Create a static archive (`.a` library).	
`-auto-use <mod>[,<mod>]`	Implicitly USE the named module(s) in all routines.	**Ftn**

-c	Generate object code (.o file) for each source file (do not link the .o's to make an executable) (default: link all .o's to make an executable).	
-C	Keep comments after preprocessing.	C,C++
-C -ffortran-bounds-check	Turn on runtime subscript range checking. Set the F90_BOUNDS_CHECK_ABORT environment variable to YES to cause the program to abort when a bounds error is encountered.	Ftn
-copyright	Prints various copyright notices for the compiler	
-E	Just run the cpp preprocessor (not the compiler or linker), and write the output to stdout.	
-g[0|1|2]	Generate debug information for gdb (0->none, 2->maximum), –g implies –g2	
-I<path>	Add <path> to the preprocessor's directory search path.	
-L<path>	Add <path> to the linker's library search path.	
-l<libname>	Search <libname> for unresolved external names (i.e. -lc searches for libc.a/libc.so in all directories in the linker's library search path).	
-module <dir>	Create .mod files in <dir>.	Ftn
-o <filename>	Write the output (typically object code or the executable program) to <filename>.	
-pg	Generate code so gprof can profile this program.	
-profile	Similar to -pg, but includes libraries.	
-S -keep	Generate symbolic assembly language instead of .o file(s). The -keep flag can also be used to keep the assembly file around while still producing the .o file.	
-shared	Create a shared library (.so library).	
-v -show -###	List subprocess commands executed during compilation. Similar to -v, except no compiling or linking is actually done.	

-version	List version numbers (and other miscellaneous information) on **stderr**.	
-dumpversion	List the compilers version number on **stdout**.	
-w	Disable warning messages.	
-Wall	Enable all warning messages.	C/C++
-W<c>,<args>	Pass <**args**> to subprocess <**c**>.	
Pre-processor Flags		Lang
-cpp **-nocpp**	Run cpp before compiling, regardless of the filename's suffix (default: run cpp unless the file's suffix is .f, .f90, .f95, or .s). Do not run cpp.	
-D<name>[=xxx]	Set the macro named <**name**> to **xxx** (**xxx** defaults to **1**).	
-fcoco	Use Fortran's ISO/ANSI standard conditional compilation facility.	**Ftn**
-mfortify_source **-mno-fortify_source**	Enable gcc's automatic buffer overflow detection. Causes undefined external symbols on some Linux distributions (i.e. Ubuntu 9). Disable automatic buffer overflow detection. This is the default.	
-U<name>	Undefine <**name**> in the preprocessor.	
Code Generation Flags		Lang
-m32	Disabled. 32-bit ABI is not supported.	
-m64	Compile code for 64-bit ABI (default).	
Optimization Flags		Lang
-apo	Enable auto-parallelization.	
-ffast-math **-fno-fast-math**	Faster floating point performance thru relaxed ANSI/IEEE rules. Conform to ANSI/IEEE rules.	
-ffast-stdlib **-fno-fast-stdlib**	Use fast stdlib functions. Use normal stdlib functions.	

`-ffloat-store`	Do not keep floating point variables in extended precision x86 f.p. registers.	
`-finline` `-fno-inline`	Enable function inlining (same as –inline). Disable function inlining.	
`-finline-functions` `-fno-inline-functions`	Enable function inlining (same as –inline). Disable function inlining.	C,C++ C,C++
`-fkeep-inline-functions`	Generate code for all functions, even if fully inlined.	C,C++
`-fmath-errno` `-fno-math-errno`	Set ERRNO after instruction that implements a math function. Don't set ERRNO after instruction that implements a math func.	
`-help` `-help:string`	List supported compiler flags. List compiler flags that contain "**string**".	

`-INLINE`
`-inline`
`-noinline`
	Request inline processing.
	Enable function inlining.
	Disable function inlining.
		
		C,C++
		C,C++

`-ipa`	Turn on inter-procedural analysis.	
`-noexpopt`	Don't optimize exponentiation operations.	
`-O[0|1|2|3]`	Set the optimization level (0->none, 3->max, -O is the same as –O2 and is the default).	
`-Ofast`	Same as –O3 –ipa –OPT:Ofast –fno-math-errno –ffast-math.	
`-std=[c89|c99|c9x|gnu89|gnu99|gnu9x]`	Set the C aliasing model to the specified model. The default is **c89**. **IMPORTANT NOTE:** Most C/C++ code will not vectorize unless the C aliasing model is **c99**. The aliasing model should only be set to **c99** when the code being compiled conforms to this model. See Pointer Aliasing Model and Vectorization for more details about the aliasing model.	
Language Semantic Flags (size of intrinsic types, external naming conventions, …)		Lang
`-byteswapio`	Swap bytes during unformatted I/O.	Ftn
`-colN`	Set fixed form line length to **N** (**N** must be 72, 80, or 120).	Ftn
`-convert`	Swap bytes during unformatted I/O.	Ftn

`-d-lines`	Compile lines with a 'D' in column 1 (in fixed form).	Ftn
`-d8`	use double=8 and dcomplex=16.	Ftn
`-default64`	Same as specifying –r8 –i8.	Ftn
`-fdirectives` `-fno-directives`	Recognize Fortran compiler directives inside comments. Don't recognize Fortran compiler directives inside comments.	Ftn Ftn
`-extend-source` `-noextend-source`	Enable 132 column Fortran fixed form source. Disable 132 column Fortran fixed form source.	Ftn Ftn
`-i4` `-i8`	Set length of default integer type to 4 bytes. Set length of default integer type to 8 bytes.	Ftn Ftn
`-no-intrinsic=<name>`	Treat <name> as EXTERNAL, not an intrinsic procedure.	Ftn
`-pad-char-literals`	Pad character literal used as actual argument to size of default integer.	Ftn
`-r4` `-r8`	Set length of default real type to 4 bytes. Set length of default real type to 8 bytes.	Ftn Ftn
`-fsecond-underscore` `-fno-second-underscore`	Append second underscore to external names that already have one. Don't append second underscore to external names that already have one (default).	Ftn Ftn
`-funderscoring` `-fno-underscoring`	Append underscores to external names (default). Don't append underscore to external name.	Ftn Ftn
`-static-data`	Treat local data as static.	Ftn
`-trapuv`	Trap references to uninitialized floating point variables (generates a Floating Point Exception when referenced) (host code only).	
`-zerouv`	Zero uninitialized variables (host code only).	
Language Dialect Flags		Lang
`-ansi`	Print an error message for non-standard constructs.	
`-backslash`	Treat backslash as a normal character (Fortran does this by default).	
`-fgnu-keywords` `-fno-gnu-keywords`	Recognize the **`typeof`** keyword (default). Do not recognize the **`typeof`** keyword.	

-fixed-form -free-form	Assume all Fortran source is fixed form (statements start in column 7, labels in 1-5, statement continuation in 6). Assume all Fortran source is free form.	Ftn Ftn
-nobool	Disable the `bool` keyword.	C,C++

4.10 Convey Directives and Pragmas

The Convey C/C++ pragmas and Fortran directives described below give the user precise control over which routines and what parts of a routine will be candidates for execution on the Convey coprocessor, as well as which variables will be placed or migrated into host/coprocessor memory.   There are also a few other Convey directives/pragmas to guide the vectorization phase of the compiler when generating vector code for loops (for one of Convey's vector personalities).

Equivalent C/C++ pragmas and Fortran directives exist for each of the table entries below. For example, the C/C++ **begin_coproc** pragma is specified as

```
#pragma cny begin_coproc
```

The equivalent Fortran directive is specified as

```
!$cny begin_coproc
```

Fortran directives may be written in upper or lower case.

If the source code being compiled is fixed-form Fortran, the initial part of a Fortran directive may also be written as **c$cny** (instead of **!$cny**).

The Convey compilers will ignore the Convey directives/pragmas (and only generate host code) unless a compiler flag that begins with **-mcny…** is specified. When porting an application, a useful debugging technique for isolating Convey compiler and coprocessor bugs is to not specify any **-mcny…** flags, which will disable all coprocessor code generation.

The **-mcny_vector** flag should be used whenever any DO/for loops in a coprocessor region should be vectorized for use with Convey's single and double precision vector personalities. Otherwise, only scalar coprocessor code will be generated.

4.10.1 Convey Coprocesser Code Generation Directives and Pragmas

Region Based Directives and Pragmas	
begin_coproc[(cond)]	A coprocessor region may be created for the source code between a begin_coproc directive/pragma and the matching end_coproc directive/pragma. The coprocessor region is only created when it is both technically feasible and at least one loop within the region was vectorized. When the -mcny_vector compiler flag is specified, DO/for loops in the coprocessor version for the region are also candidates for vectorization. When the host processor enters a coprocessor region created by this directive, and the optional (cond) expression was specified, that expression is evaluated at runtime: • If (cond) evaluates to false (zero for C/C++), or the coprocessor is not attached, the coprocessor version of the region is not dispatched, and the equivalent host code is executed. • If (cond) evaluates to true (non-zero for C/C++) or was not specified, the coprocessor version of the region will be dispatched if the coprocessor is attached. Note that (cond) must have type LOGICAL in a Fortran source file. See the definition of coprocessor region above for a description of when regions are executed on the coprocessor.
end_coproc	The matching end for a begin_coproc directive/pragma.
begin_host	Code between a begin_host directive/pragma and the matching end_host directive/pragma will not be compiled as a coprocessor region, even when automatic vectorization is enabled.
end_host	The matching end for a begin_host directive/pragma.

Routine Based Directives and Pragmas	
`array(x[i][j], …)`	Tells the C/C++ compilers to treat the named object(s) as an array for dependency analysis and code generation purposes. This allows an object declared as pointer to pointer to value (or higher dimensional object) to be treated as a contiguous multi-dimensional array and allows code containing references to such objects to be vectorized. This pragma must appear after the data declaration for the object ("`x`" in this example), and before any references to the object.
`dual_target(x)`	When '`x`' is true (non-zero), the procedure containing this directive/pragma, and all procedures thereafter (in the same source file) are compiled for both the host processor and the Convey coprocessor, until another `dual_target` directive/pragma is encountered where '`x`' is false. When '`x`' is true, the host version of the routine will include a dispatch-wrapper at the beginning of the routine. The dispatch-wrapper will dispatch the coprocessor version of the routine if the coprocessor is attached to the current process/process group; otherwise the host code for the routine is executed. The coprocessor version of the routine is renamed as described in chapter 4.7 **Naming Conventions for Routines in Coprocessor Regions** All procedure calls within a coprocessor region are also automatically mapped to the coprocessor version of that procedure (`cny_...`). When '`x`' is false (zero), only host code is generated for the affected procedures. '`x`' must be either an integer constant (in C/C++ code) or, in Fortran, a logical constant (`.true.` or `.false.`) (a named constant is permitted). This dual target with wrapper capability allows a routine to be called from either host or coprocessor code (by the same apparent name), and also allows the application to be run on a generic x86-64 system without a Convey coprocessor or simulator. Note: specifying the compiler flag `-mcny_dual_target` is equivalent to putting a `dual_target(1)` directive on the first line of the source file and compiling with the `-mcny` flag. The behavior described above can be modified with the `-mcny_lib_select` flag. This flag allows a user written profitability procedure to be called before deciding whether or not to dispatch the coprocessor version of the routine. If the profitability routine returns true, or does not exist, the coprocessor version of the routine is dispatched, otherwise the host version is executed. See Appendix F **User**

	Provided Dual Target Profitability for more details.
dual_target_nowrap(x)	Similar to **dual_target** above, except the host version of the routine does not include the dispatch-wrapper. If the host version of the routine is called, it will execute solely on the host. Note: specifying the compiler flag **-mcny_dual_target_nowrap** is equivalent to putting a **dual_target_nowrap(1)** directive on the first line of the source file and compiling with the **-mcny** flag.

The **-mcny_vector** flag should be specified when compiling a routine that might benefit from executing on the coprocessor, so that any DO/for loops inside a coprocessor region will be candidates for vectorization. The compiler will report which loops were successfully vectorized. Loops that are not vectorized should not be executed on the coprocessor, since scalar coprocessor code generally executes slower than the equivalent host processor code.

Warning: The **dual_target** and **dual_target_nowrap** directives should not be used in C++ files that contain templates that are used in more than one file. Doing so will result in duplicate symbols at link time. This restriction may be removed in a future release.

The following directives/pragmas can be used to guide the vectorization phase of the compiler when generating vector coprocessor code for loops (for one of Convey's vector personalities). These loop based directives and pragmas should be placed immediately before the loop they are intended to act upon.

Loop Based Directives and Pragmas	
fence	Forces a coprocessor fence to be generated where this directive/pragma occurs. Typically used inside a loop that has a no_fence directive.
no_fence	Disables automatic generation of coprocessor fence operations within and around a vectorized loop. Still allows fences to be created around a loop nest. Use with extreme caution, since intermittent wrong answers may result if memory stores initiated by the coprocessor do not complete before the host processor loads that memory location. The compiler attempts to only automatically insert fences where analysis shows they are needed.
no_fence(*where*)	Disables the automatic generation of coprocessor fence operations as specified in *where*. *where* should contain one of more of these keywords (comma separated): all — Suppresses all fences. around, around_loop — Suppresses fences before or after a loop before, before_loop — Suppresses fences before a loop after, after_loop — Suppresses fences after a loop between, between_deps — Suppresses fences between (potential) data dependencies within, within_loop — Suppresses fences within a loop Use with extreme caution, since intermittent wrong answers may result if memory stores initiated by the coprocessor do not complete before the host processor loads that memory location.
no_loop_dep (varlist)	The variables in varlist are assumed to have no loop carried dependencies in the following loop/loop nest (the value assigned to a variable or array element in one iteration isn't used in another iteration). When (varlist) is not specified, the compiler assumes all variables used in the loop, including compiler introduced temporary variables, do not have cross-iteration dependencies.

unroll (n)	Loop unrolling of inner vectorized loops can be performed by placing an unroll directive/pragma before the desired loop. **unroll** is only honored when the loop is vectorized. The loop will be unrolled **n** times (**n** specifies how many vectorized strips to unroll the loop by). For example, given a vector length of 1024, unrolling by 2 will result in an unrolled body that will operate on 2048 elements (2 vector strips). For loops that have small bodies, performance can be increased by unrolling the loop body to increase the amount of computation per iteration, increase instruction scheduling overlap opportunities, and amortize the loop overhead. Care must be taken to not unroll too much as this also increases register pressure (and can result in spills if too much unrolling is performed). Note: a negative unroll count is valid, and the loop is unrolled |**n**| times, unless the **-TARG:cny_loop_unroll** compiler flag is specified, in which case the value specified in the compiler flag will be used as the unroll count.
no_vector	Used to prohibit vectorization of a loop.
prefer_vector	Used to indicate which loop in a loop nest should be vectorized, if possible.

The following directives/pragmas provide further control over the code generated for the coprocessor.

Miscellaneous Directives and Pragmas	
personality("signature")	Used when you need to change personalities in the middle of a source file. The value for **"signature"** is a string and must be specified with double quotes.

4.10.2 Directive and Pragma Examples

In the following example, **begin_coproc … end_coproc** directives are used to define a coprocessor region, for which the compiler will generate both host and coprocessor code. The vectorization report shows that the compiler successfully vectorized the inner loops in the coprocessor code version of the regions.

```
$ cnyf90 -c -O3 -mcny_sig=sp -mcny_vector -cpp stencil.f
VECTORIZED STMT in MAIN__0 at 69: L 7 S 1 B 10 U 0 I 0 M 0
VECTORIZED STMT in MAIN__1 at 84: L 7 S 1 B 10 U 0 I 0 M 0

$ cat -n stencil.f
    …
    66  !$cny BEGIN_COPROC
    67        do k=2,NZ-1
```

```
          do j=2,NY-1
            do i=2,NX-1
              p2(i,j,k) = sp*p1(i,j,k)
     &                    + sx*(p1(i+1,j  ,k  )-p1(i-1,j  ,k  ))
     &                    + sy*(p1(i  ,j+1,k  )-p1(i  ,j-1,k  ))
     &                    + sz*(p1(i  ,j  ,k+1)-p1(i  ,j  ,k-1))
            enddo
          enddo
        enddo
!$cny END_COPROC



!$cny BEGIN_COPROC
        do k=2,NZ-1
          do j=2,NY-1
            do i=2,NX-1
              p1(i,j,k) = sp*p2(i,j,k)
     &                    + sx*(p2(i+1,j  ,k  )-p2(i-1,j  ,k  ))
     &                    + sy*(p2(i  ,j+1,k  )-p2(i  ,j-1,k  ))
     &                    + sz*(p2(i  ,j  ,k+1)-p2(i  ,j  ,k-1))
            enddo
          enddo
        enddo
!$cny END_COPROC
…
```

The vectorization report format is described in chapter 4.15 Vectorization Report.

4.11 Coprocessor Code Generation Inhibitors

The following language features are a partial list of features that prevent coprocessor vector code generation for a region or routine:

- In Fortran code:
  - Computed `goto`s/assigned `goto`s
  - `select case` constructs
  - Multiple entry routines
  - Subscript range checking flag (-C / `-ffortran-bounds-check`)
- In C/C++ code
  - `switch` statements

References to the following routines, statements, and compiler flags, by default, prevent coprocessor code generation (vector or scalar) for a region or routine. These features should only be used in code that will execute on the host.

- C/C++ routines
  - C stdio functions
    scanf, sscanf, fscanf, printf, sprintf, fprintf, fpopen, fclose, fwrite, fread, gets, fgets, fseek, ftell, puts, fputs
  - system I/O calls
    open, close, read, write, lseek, pipe, ioctl, select, pselect

- file management
  rename, link, unlink, chmod, symlink, path_resolution
- C/system memory functions
  malloc, calloc, free, mmap, munmap, brk, sbrk
- system process control
  fork, vfork, wait, _exit, exit, execv, execvp, execle, execlp, execl, system
- signal processing
  signal, kill, alarm, pause, sigaction, sigpending, sigprocmask, sigqueue, sigsuspend, killpg, raise, sigsetops, sigvec
- coprocessor support routines
  cny_core_id, cny_cp_attributes, cny_cp_free, cny_cp_interleave, cny_cp_malloc, cny_cp_mem_size, cny_cp_numa_id, cny_cp_pagesize, cny_cp_realloc, cny_dump_faddr, cny_f_coprocessor_ok, cny_f_finalize, cny_f_get_signature, cny_memory_locale, cny_numa_id, cny_page_toucher, cny_str_locale
- low level coprocessor support routines
  cny_abort, cny_se_set_mask, cny_se_get_mask, cny_ae_set_mask, cny_ae_get_mask, cny_ae_set_mask_bits, cny_ae_clear_mask_bits, cny_se_set_mask_bits, cny_se_clear_mask_bits, cny_set_vpm, cny_coproc_precise, cny_cit_read, cny_cit_read_fence, cny_fence, cny_mcount, cny_cp_montr, cny_cp_montr_init, cny_cp_montr_disp, cny_cp_montr_call, cny_cp_montr_loop, cny_cp_montr_bb, cny_cp_montr_rtn, cny_cp_montr_v_init, cny_cp_montr_v_disp, cny_cp_montr_v_call, cny_cp_montr_v_loop, cny_cp_montr_v_bb, cny_cp_montr_v_rtn

- C++ features
  - Exception handlers (**try**/**catch** blocks)
- Fortran routines, statements, and flags
  - All Fortran I/O statements (**READ, WRITE, OPEN, CLOSE, BACKSPACE** …)
  - **STOP** and **PAUSE** statements
  - **ALLOCATE** and **DEALLOCATE** statements
  - Any array valued intrinsic that the compiler cannot inline
    - i.e. **SUM(X,N)** – sum X across dimension # N (where N is not a constant)
  - assorted **CHARACTER** valued expressions
  - argument checking flag
  - array shape conformance checking flag
  - calls to posix, magtape, signal support, memory allocation, and other library routines

Note: there may be some language constructs or routine calls that do not inhibit coprocessor code generation, but the necessary coprocessor library routines do not exist. In such cases, an unresolved external symbol will occur.

There are two compiler flags which can be used to specify a file containing a list of routine names that control whether or not those routines names may be called from coprocessor code in a region.

- `-mcny_exclude_names=filename`: Exclude all named routines in 'filename' from being called from coprocessor code. This *excluded* list of names is in addition to the predefined *excluded* list described above (syscalls, I/O, ...). A region containing a call to one of these excluded routine names will not generate a coprocessor version of that region, unless that name is also specified via the `-mcny_callable_names` flag or the `CNY_CALLABLE_NAMES` environment variable.
- `-mcny_callable_names=filename`: Allow all named routines in 'filename' to be called from coprocessor code, if that name is in one of the excluded lists.

If a routine name is not in one of the *excluded* lists, it is assumed to be callable.

In addition to these two compiler flags, there are corresponding environment variables:

- `CNY_EXCLUDE_NAMES`, and
- `CNY_CALLABLE_NAMES`

which may be set to a filename containing a list of routine names.

4.12 Memory Model

See the **Convey Reference Manual** (Memory Model) for a complete description of the Convey server's memory system.

A Convey server provides two regions (pools) of physical memory: physical memory attached to the host processor, and physical memory attached to the coprocessor. To an application, these regions appear as a common linear cache coherent address space. Pages in an application's virtual address space can be mapped onto either memory region and can be migrated between regions on request.

All memory is accessible from either processor, and is cache coherent, however:

- The host processor can access host memory much quicker than coprocessor memory, just as the coprocessor can access coprocessor memory much quicker than host memory.
- **Therefore, to realize the performance potential of the Convey coprocessor, it is critical that all arrays referenced by the Convey coprocessor during a dispatch be in coprocessor physical memory for the duration of that dispatch.**

When programming in coprocessor assembly language, the coprocessor's weakly ordered memory model may occasionally require a *fence* instruction to ensure the coprocessor retrieves the correct value from coprocessor memory. See the **Convey Reference Manual** (Coprocessor Memory References) for a description of when a fence instruction is required.

4.12.1 Host and Coprocesser Memory Granularity

On a Convey hybrid-core server, there are two pools of physical memory.

- The host memory pool, up to 128GB, consists of memory physically located on the x86 motherboard.
- The coprocessor memory pool, up to 128GB, consists of either standard or scatter-gather DIMMs, and is located on the coprocessor board.

A process that has the coprocessor attached can use 4Kbyte to 4Mbyte size pages (all possible powers of 2). Other processes can only use 4Kbyte pages. When the OS needs to fault in an area of virtual memory, for the first time, it attempts to allocate the largest possible page size (in either host or coprocessor memory).

4.12.2 Convey Memory Locality Directives and Pragmas

The following directives control the initial allocation and runtime migration of data between host and coprocessor memory.

In the following table, '**`v`**' may be any variable or address expression, '**`sv`**' must be a statically allocated variable and '**`n`**' must be an integer byte count.

Memory Placement and Migration Directives and Pragmas	
`coproc_mem(sv)`	The static variable `sv` is initially allocated in coprocessor memory. To specify a Fortran common block name, enclose the name in '`/`'s, i.e. `!$cny coproc_mem(/cmnblkname/)`
`host_mem(sv)`	The static variable `sv` is initially allocated in host memory. To specify a Fortran common block name, enclose the name in '`/`'s, i.e. `!$cny host_mem(/cmnblkname/)`
`migrate_coproc(v,n)`	The physical pages containing `n` contiguous bytes starting at address **`v`** are migrated to coprocessor physical memory.
`migrate_host(v,n)`	The physical pages containing `n` contiguous bytes starting at address **`v`** are migrated to host physical memory.

4.12.3 Initial Memory Allocation

The `coproc_mem` and `host_mem` directives/pragmas instruct the compiler and linker to initially allocate the specified static variables in the specified memory. Statically allocated variables include the following:

- Fortran variables that are initialized or have the `SAVE` attribute,
- variables in a Fortran `COMMON` block,
- Fortran module variables (any variable declared in the specification part of a `MODULE` program unit), typically accessed via `USE` statement, and
- C/C++ static or global scope variables

The primary use of these directives/pragmas is to avoid unnecessary runtime migration of memory.

The `coproc_mem` and `host_mem` directives/pragmas should always be placed after the declaration for the variable.

Statically allocated variables that are referenced by a `coproc_mem` directive are typically placed in an appropriate linker section (`.cbss` or `.cdata`) that is automatically placed in coprocessor memory when pages from that section are initially paged into a process. Other statically allocated variables are not affected by this allocation scheme, except in the following cases:

- When a Fortran COMMON block variable appears in a `coproc_mem` directive, the entire COMMON block is allocated in coprocessor memory. This is equivalent to a `coproc_mem` directive that specifies the `COMMON` block name itself.
- When a Fortran MODULE variable appears in a `coproc_mem` directive, all of that module's variables are allocated in coprocessor memory.

4.12.4 Runtime Memory Migration

When runtime memory migration is requested, due to a `migrate_coproc` or `migrate_host` directive/pragma, one or more whole pages in the application's virtual address space are migrated, including the entire memory block specified in the directive/pragma.

- If a page to be migrated is currently resident in physical memory but is not in the desired memory pool, the operating system will copy that page to physical memory in the correct memory pool and then update the page table entries so the new copy of that page has the same virtual address as the original page.
- If a page to be migrated is not resident (i.e. has never been paged in, or is currently paged out) it will be paged into the desired memory pool when it is next referenced. No actual copying of the page occurs.

Since whole pages are migrated, if another variable (or portion thereof) shares a physical memory page with a location that is being migrated, it will also be migrated.

The `migrate_coproc(v,n)` and `migrate_host(v,n)` directives can be used with any variable, including local stack variables and malloc'ed memory. These directives/pragmas act like executable statements. When encountered in host code, they cause the memory pages containing the designated memory locations to be migrated to the designated memory pool. These directives are ignored when encountered in the coprocessor version of a coprocessor region. The `migrate` directives/pragmas should be used instead of the equivalent library routines because non-Convey compilers typically ignore Convey directives/pragmas, providing greater code portability.

Note: The `migrate_coproc` and `migrate_host` directives assume that the memory to be migrated is contiguous. If `v` is a Fortran variable with the POINTER attribute, and the associated storage is non-contiguous, not all of the memory locations referenced by `v` will be migrated.

Note that some pages may not be migrated successfully if the operating system cannot safely do so; however, the program is allowed to continue execution with no error indication from the migration request. Here is a partial list of conditions that inhibit page migration:

- there is pending asynchronous I/O for the page
- there isn't enough physical memory of the desired type currently free (i.e. due to high system load)
- another thread is performing I/O on part of a page
- the process has `mlock()`'ed the page

4.12.5 When to Migrate Memory

Assuming there is sufficient memory in the target memory pools to accommodate all migration requests, all **array data** that is going to be referenced (loaded or stored) during a coprocessor dispatch should be migrated to coprocessor memory before the dispatch.

After the dispatch completes, that array data should be migrated back to host memory if it will be referenced from the host processor before being referenced again in a subsequent dispatch. If there is insufficient free memory to accommodate all migration requests, some migration requests will silently fail, and the application will execute correctly but slower than expected. Failure to properly migrate all of the arrays that are referenced by coprocessor code will have drastic performance consequences.

Migration of scalar variables can also improve performance (but isn't as important), and must be done carefully since memory is migrated a whole page at a time. Frequently accessed scalar variables may be migrated to coprocessor memory, as long as the memory page containing those variables does not also contain variables accessed frequently from host code.

4.13 Coprocessor Exceptions

To conform to typical implementations of Fortran, C, and C++, the following coprocessor exceptions are masked (ignored) for both the AE and the IAS, unless the user requests otherwise:

Floating point underflow	An arithmetic operation computed a non-zero value too small to represent as a normalized floating point value.
Floating point denormalized operand	An arithmetic operation encountered a denormalized floating point input operand (a "denorm"). Such a denorm might have been generated on the host processor, or read in from a binary data file.
Integer Overflow	An integer arithmetic operation overflowed.
Store Overflow	A store of a 64 bit value from a coprocessor register into a smaller memory location truncated significant bits.

It is possible to set the exception mask explicitly using the Convey exception mask library interface (see **Exception MasksException MasksException Masks**) or via the **CNY_AE_MASK** environment variable (see **CNY_AE_MASK**).

4.14 Automatic Vectorization

Automatic vectorization of loop nests is provided as an advisory tool to help identify those loops nests that can be vectorized. This provides a list of vectorizable loops to the user to then decide which loops should actually be vectorized. Vectorizing all possible loops in an application will usually cause the application to run slower. In addition, once the set of desired loops is selected, careful initial allocation and runtime migration of data between host and coprocessor memory is needed to further improve performance.

The `-mcny_auto_vector` flag enables automatic vectorization of DO/for loops. The Convey compilers will generate coprocessor regions and vectorized coprocessor code for loops when those loops can be vectorized and they are likely to be profitable. Similar to coprocessor regions created via directives/pragmas, automatically vectorized loops have both a host code and coprocessor code block. If the coprocessor is not attached, or the coprocessor code block does not seem to be profitable, the host code version will be executed instead of dispatching the coprocessor code block.

Typically, a coprocessor region is created for the outermost loop in a loop nest, while the innermost loop is vectorized. The loop nest optimizer phase of the compiler may reorder loops within a loop nest to improve performance or aid in vectorizing the innermost loop.

The `-mcny_auto_vector` flag implicitly turns on the `-mcny` and `-mcny_vec_info` flags.

4.15 Vectorization Report

When either the `-mcny_vector` or the `-mcny_auto_vector` flag is specified, the compiler will generate vectorization reports with the following formats:

- **`VECTOR LOOP in <routine> at <line>: L # S # B # U # I # M #`**
  where the # symbols above will be a count of the number of times the following occurred in the vectorized loop:

  `L` – number of vector loads
  `S` – number of vector stores
  `B` – number of vector binary operations
  `U` – number of vector unary operations
  `I` – number of vector intrinsics
  `M` – number of vector miscellaneous operations

- **`VECTOR SPECIAL in <routine> at <line>:  C # S # R # M # U#`**
  where the # symbols above will be counts:

  `C` – number of Component operations
  `S` – number of Shift operations
  `R` – number of Rotation operations
  `M` – number of Match operations
  `U` – number of unrolled vector loops

- **`VECTOR IDIOMS in <routine> at <line>: R # L # S # C # X # P #`**
  where the # symbols above will be counts:

  `R` – number of Reduction operations
  `L` – number of Count operations
  `S` – number of Search operations
  `C` – number of Compress operations
  `X` – number of Expand operations
  `P` – number of Prefix operations

- **`LOOP TRANSFORMS in <routine> at <line>: D # I # X # N # S # A #`**
  where the # symbols above will be counts:

  `D` – number of loop Distribution operations
  `I` – number of loop Interchange operations

X – number of scalar eXpansion operations
N – number of Node splitting operations
S – number of Scalar renaming operations
A – number of Array renaming operations

Note that the routine name may be modified by the compiler, but will be recognizable.

5 Coprocessor Code Examples

5.1 Vectorizing Loops Example

This example is intended to show one approach to generating coprocessor code for an application. As with any of Convey's vector personalities, generating vector code for loops, coupled with proper memory migration, is key to speeding up an application. In this example, all the Convey specific directives and compiler flags are in dark red font.

The descriptions contained here for this example provide the minimum necessary information for beginning to port an application to utilize the Convey coprocessor. The rest of this guide provides additional detailed information about these topics, and describes other ways to take advantage of the Convey coprocessor.

This example was compiled with:

```
$ cnyf90  -mcny_vector  example.f90
```

and `example.f90` contained:

```
program stencil
  integer(8), parameter:: NX=1000, NY=1000, NZ=100, NI=20
  real   p1(NX,NY,NZ), p2(NX,NY,NZ)
  real  sp, sx1, sx2, sy1, sy2, sz1, sz2
  real  gbyte, gflop, size, arraysize
  real*4 t1, t2, ttot, seconds
! put all the relevant data into coprocessor memory
  common /cb/ p1, p2, sp, sx1, sx2, sy1, sy2, sz1, sz2
!$cny coproc_mem (/cb/)

  external seconds
  integer*8 i, j, k, iter

!  see coproc_mem above !$cny migrate_coproc(p1, ((nx*ny*nz*2)+7)*4)
! gigabytes and gigaflops per iteration
  arraysize=float(NX*NY*NZ)
  size=float((NX-2)*(NY-2)*(NZ-2))
  gbyte=2.0*8.0*4.0*1.e-9*size
  gflop=2.0*13.0*1.e-9*size

  write(6,1000) NI,NX,NY,NZ,gflop,gbyte
1000  format('iterations: ',i6/                     &
             'array size: ',i4,'x',i4,'x',i4/       &
             'gigaflops/iteration: ',e6.1/          &
             'gigabytes/iteration: ',e6.1)

!$cny begin_coproc
! initialize the arrays (on the coprocessor)
  p1 = 0.0  ! sets the whole array (vectorizable)
  p2 = 0.0

  p1(:,:,1) = 1.0
  p2(:,:,1) = 1.0
!$cny end_coproc
```

All variables used in the loops to be executed on the coprocessor have been put into a COMMON block, just to make it easier to place them all in coprocessor memory.

The coproc_mem directive places the COMMON block /cb/ into coprocessor memory at link time.

The similar migrate_coproc directive could have been used to migrate memory at runtime.

The migrate_coproc directive is less efficient than link time allocation of static data in coprocessor memory. If some memory locations are accessed frequently on the host before the coprocessor routine/region begins accessing those locations, then the memory should be initially allocated on the host and migrated to the coprocessor just before the coprocessor routine/region is invoked.

coprocessor region

contains both host and coprocessor versions of the region

The initialization code in this example is run on the coprocessor. The begin_coproc directive starts a <u>coprocessor region</u>, that contains both <u>host</u> and <u>coprocessor</u> versions of the code in that region. When this region is executed at runtime, the coprocessor version is dispatched if the coprocessor is attached; otherwise, the host version is executed.

```
  sp=0.0
  sx1=1.0/6.0 ; sx2=1.0/6.0
  sy1=1.0/6.0 ; sy2=1.0/6.0
  sz1=1.0/6.0 ; sz2=1.0/6.0

  ttot=0.0
```

This code is executed on the host. If the following **DO** loop didn't call a host only routine and didn't contain a **WRITE** statement, the coprocessor region above could have been expanded to include this code and the loop below, avoiding a second dispatch.

```
  do iter=1,NI
    t1=seconds()
!$cny begin_coproc
    call stencil_sub(p1,p2,NX,NY,NZ,sp,sx1,sx2,sy1,sy2,sz1,sz2)
    call stencil_sub(p2,p1,NX,NY,NZ,sp,sx1,sx2,sy1,sy2,sz1,sz2)
!$cny end_coproc
    t2=seconds()
    ttot=ttot+(t2-t1)
    write(6,1010) iter,t2-t1,gflop/(t2-t1)
  enddo

  write(6,1020) ttot,float(NI)*gflop/ttot
  do k=1,10
    write(6,1030) p1(NX/2,NY/2,k)   ! check a few answers
  enddo

1010 format('iteration ',i6,', ', f10.3,' seconds, ', f10.2,' gflop/sec')
1020 format('total:            ', f10.3,' seconds, ', f10.2,' gflop/sec')
1030 format(e10.2)
1040 format(a2)
end
```

In the <u>coprocessor</u> version of this region, the procedure calls to **stencil_sub** are replaced with calls to **cny_stencil_sub**. This naming convention is the same used by the **dual_target** directive / compiler flag.

This allows a **dual_target** routine to be called from either host or coprocessor code.

```
subroutine stencil_sub(p1,p2,NX,NY,NZ,sp,sx1,sx2,sy1,sy2,sz1,sz2)
!$cny dual_target(1)
  integer*8 NX,NY,NZ,i,j,k
  real*4 p1(NX,NY,NZ),p2(NX,NY,NZ), sp,sx1,sx2,sy1,sy2,sz1,sz2
  do k=2,NZ-1
    do j=2,NY-1
      do i=2,NX-1
        p2(i,j,k) = sp*p1(i,j,k) + sx1*p1(i-1,j  ,k  )      &
                                 + sx2*p1(i+1,j  ,k  )      &
                                 + sy1*p1(i  ,j-1,k  )      &
                                 + sy2*p1(i  ,j+1,k  )      &
                                 + sz1*p1(i  ,j  ,k-1)      &
                                 + sz2*p1(i  ,j  ,k+1)
      enddo
    enddo
  enddo
  return
end
```

The routine **stencil_sub** was compiled both for the host and coprocessor, by using the **dual_target** directive.

An alternatve approach is to put **stencil_sub** in its own souce file, and compile that file with the **-mcny_dual_target** flag.

The **dual_target** directive caused two versions of this routine to be created, **stencil_sub** (containing host x86-64 code), and **cny_stencil_sub** (containing coprocessor code). The host version of the routine (**stencil_sub)** also checks to see if the coprocessor is attached, and if so, dispatches **cny_stencil_sub**; otherwise, it executes the host code in **stencil_sub**.

```
real*4 function seconds()
!$cny dual_target(0)
  integer t,r
  call system_clock(count=t, count_rate=r)
  seconds = real(t)/ real(r)
end function
```

This **dual_target** directive (with the **(0)**) disables **dual_target** code generation, and only host code is generated for **seconds()**. Since **seconds()** calls a host only library routine (**system_clock**), we cannot compile this routine **dual_target**.

Notes on the example above:

- Statically allocated variables may be placed in host or coprocessor memory with the `host_mem` and `coproc_mem` directives. This includes:
    - Fortran `COMMON` blocks and variables therein, `SAVE`d variables, initialized variables (variables in `DATA` statements and variables declared in type statements with initial value specifiers), and module variables.
    - C/C++ external variables and `static` variables

  The coprocessor can access coprocessor memory quickly, while the host processor can access host memory quickly. Frequent memory accesses from either the host processor or coprocessor to the other processor's memory should be avoided.

- The `migrate_host` and `migrate_coproc` directives act like executable statements, migrating the specified number of bytes to the indicated memory. `!$cny migrate_coproc(p1, ((nx*ny*nz*2)+7)*4)` attempts to migrate the specified number of bytes, starting at `p1(1)`, into coprocessor memory.

  When migrating a batch of variables, it is important to note that each migration request, when successful, migrates whole pages, not just the indicated byte count. If some of those variables occupy the same page, and are migrated individually, the system will end up migrating some pages twice (or more). Migrating coprocessor resident pages to coprocessor memory is very slow, and should be avoided.

  Once solution to this problem is to group the batch of variables together (in a Fortran `COMMON` block, or in a C `struct`, for statically allocated variables) and make one migration request for the entire batch. For non-static variables, you can query which memory pool the first byte of each variable is in, and only migrate the variable if necessary. If the variable is a large array, please avoid querying the memory location of the entire array, since that is a very high overhead operation.

- A coprocessor region is enclosed by `begin_coproc` and `end_coproc` directives.
    - The region should only contain numeric computations, loop constructs, and calls to routines that were compiled with the `dual_target` directive/pragma or flag, or have the `cny_` coprocessor version of that routine provided by some other method. Mathematical intrinsic references are also permitted. The compiler will refuse to generate coprocessor code for a region for statements and routine calls that it knows are invalid in coprocessor code. Calls to a user procedure, such as "`x`", where a coprocessor version of that procedure does not exist will result in an undefined external reference to `cny_x`.
    - If the coprocessor is not attached, the region will execute the host version of the code for that region.

- For applications which do not perform a lot of work in a single dispatch, the dispatch (invoking a coprocessor code region or routine from host code) is a relatively expensive operation. Reducing the number of dispatches in such an application is one way to improve the performance of that application. Possible approaches include
    - If two coprocessor regions are close together, combining them into one region should be attempted:
        - If there is a small amount of scalar code between the two regions, executing that scalar code on the coprocessor, even though slower than executing that code on the host, may be faster due to fewer

        dispatches and possibly eliminate the need for additional memory migration requests between regions.

        - If there is some code between the two regions that cannot be compiled in a coprocessor region, such as I/O statements or calls to host only library routines, restructuring that part of the application to provide a single larger region could prove beneficial.

    - If a routine is called frequently from a loop executing on the host, and that routine is compiled **`dual_target`**, then each call of that routine from that loop causes a new dispatch. If possible, placing that loop in a coprocessor region (with a **`begin_coproc`** directive/pragma) will reduce the number of dispatches. Note that the called routine can still be called from host code elsewhere, since it was compiled **`dual_target`**.

    - If a routine compiled **`dual_target`** is called with both large and small array arguments,and the overhead of a coprocessor dispatch is larger than the performance improvement when that routine is called with small arrays, a user provided profitability analysis routine can be used to avoid dispatches based on the arguments passed into that **`dual_target`** routine (see Appendix F **User Provided Dual Target** Profitability).

    - A routine can be compiled dual_target_nowrap, forcing the host version of the routine to be used when called from host code, and the coprocessor version to be used when called from coprocessor code. If all calls to this routine that are expected to benefit from running on the coprocessor are put into a coprocessor region, and those calls that aren't expected to benefit are left in host only code, inappropriate dispatches can be avoided.

- Convey's vector personalities can provide a substantial performance boost for certain vectorizable loops, but executing large amounts of scalar code on the coprocessor should be avoided. Executing small sections of scalar code is acceptable, particularly if some memory migration requests or coprocessor dispatches can be eliminated.

- The more floating point vector operations executed per dispatch, the more likely that dispatch will execute faster that the equivalent host code. The compiler vectorizer attempts to fuse loops and reorder loop nests in order to generate larger coprocessor regions, but some applications will benefit from hand restructuring of loops and loop nests.

- Algorithm choices can affect the performance of the coprocessor. Solvers that work on large array sections will perform better on the coprocessor than solvers that work on varying size sections.

- Memory migration requests and checking which memory pool a block of memory is in are both relatively slow operations.

- If an application contains a *standard* solver, there may be an equivalent routine in the Convey Mathematical Libraries that is already optimized for the Convey coprocessor. The Convey Mathematical Libraries include:

    - Dense vector and matrix operations, including the Level 1, 2, and 3 BLAS
    - Sparse vector operations, including the sparse BLAS
    - Linear equation solution, including LAPACK
    - Eigenvalue solution, including LAPACK
    - Discrete Fourier Transforms, including 1-D, 2-D, 3-D and multiple 1-D

5.2 Loop Unrolling / Vector Reduction Example

For loops which have small bodies, performance can be increased by unrolling the loop body to increase the amount of computation, increase instruction scheduling overlap opportunities, and amortize the loop overhead. Care must be taken, however, to not unroll too much as this also increases register pressure (and can result in spills if too much unrolling is performed). For vectorized loops, the unroll factor is not currently selected automatically by the compiler but it is possible to control it manually. Loop unrolling of inner vectorized loops can be performed by placing an `unroll` directive/pragma before the desired loop:

```
#pragma cny unroll(4)
for (i=0; i < n; i++) {
    ...
```

The `unroll` directive/pragma is currently only honored when the loop is vectorized.

Loop unrolling directives/pragmas are honored by default, but if desired, the loop unrolling pragmas can be disabled with the following option:

```
-TARG:cny_loop_unroll=0
```

Alternatively, the unroll factor used in the pragmas can also be controlled from the command line (useful when tuning the performance of a given unrolled loop). This is accomplished by specifying a negative unroll factor in the pragma/directive:

```
#pragma cny unroll(-4)
```

and using the following flag (where `N` is the desired factor):

```
-TARG:cny_loop_unroll=N
```

The negative unroll factors will be overridden by the value given with the flag. When the flag is not specified, the absolute value of the negative unroll factor will be used.

OPTIMIZED VECTOR REDUCTIONS

Since a vector reduction (e.g. sum) is higher cost than the corresponding vector operation (e.g. add), reductions can be optimized by the compiler by using the corresponding vector operator within the loop (maintaining the intermediate results as a vector instead of a scalar) and performing the final vector to scalar reduction once after the loop is complete. This is accomplished in the compiler through the use of unrolling and a vector pseudo register (VPreg).

To enable this optimization (currently off by default), use:

```
-TARG:cny_reduce_vpreg
```

Since this optimization uses unrolling, note that if no unroll pragma is present, the vector loop containing the reduction will be unrolled with a factor of 1 to create a body which operates with the maximum vector length and a tail loop for the remaining iterations.

Below is an example which demonstrates the use of both the `unroll` pragma as well as the `-TARG:cny_reduce_vpreg` flag to optimize reductions.

```
$ cat reduce.c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define N  100000

int main(int argc, char **argv) {
   long i, n = N;
   float sum, chk, esp, diff;
   int rval = 0;

   if (argc > 1) {
       n = atol(argv[1]);
   }

   long size = sizeof(float)*n;
   float *a = (float*) malloc(size);
#pragma cny migrate_coproc(a, size)
   // initialize data
#pragma cny begin_coproc
#pragma cny unroll(4)
   for (i=0; i < n; i++) {
       a[i] = i;
   }
#pragma cny end_coproc

   // compute sum reduction
   sum = 0;
#pragma cny begin_coproc
#pragma cny unroll(4)
   for (i=0; i < n; i++) {
       sum += a[i];
   }
#pragma cny end_coproc

   // compute closed form sum (for above dataset) and allowed error
   chk = ((long)n*(n-1)) / 2;
   esp = chk * 0.0001;

   printf("sum = %f\n", sum);
   printf("chk = %f\n", chk);
   diff = fabs(chk - sum);
   if (diff < esp) {
       printf("PASSED\n");
   } else {
       printf("FAILED fabs(%f - %f) = %f > %f \n", chk, sum, diff, esp);
       rval = 1;
   }
   return(rval);
}
```

6 Personality Specific Features

Most of the features described in this guide are generally applicable to all Convey supplied vector personalities. This section describes those features that are particular to one particular personality.

6.1 Financial Analytics Personality

This personality provides all the general functionality of any vector personality, with the following additional features and exceptions:

- Additional vector instructions that allow very efficient computation of these common intrinsic functions (FORTRAN NAME, c double/c single names):
    - e^x (EXP, exp/expf)
    - natural log (ALOG, log/logf)
    - square root (SQRT, sqrt/sqrtf) with no more than a two bit error
    - reciprocal (1.0/x) with no more than a one bit error
    - sine and cosine (SIN, COS, sin/sinf, cos/cosf), with no more than two bits error when input value x is -64000<x<64000
    - error functions (ERF, erf/erff) and (**CDF**, **cdf**)
- Additional vector instructions that compute the cumulative distribution function very efficiently
- Additional vector instructions that generate Mersenne Twister random numbers very efficiently
- Additional vector instructions that generate Sobol random number sequences very efficiently

Note: the additional instructions provided to assist in computing reciprocals, sine, and cosine, use table lookups to generate a first approximation of the result. If the vector of input values provided to these table lookup instructions are not sufficiently random, the table lookup will suffer from memory bank collisions, incurring a substantial performance penalty. Compiler flags are available to force the use of an alternative instruction sequence that does not use table lookups (see **Less Commonly Used Compiler Flags**).

The standard Fortran/C/C++ intrinsics named above use some of these additional vector instructions when a vectorized loop contains those intrinsics and the compiler flag **-mcny_sig=fap** was specified. The special support for e^x and natural log requires additional compiler flags.

The remaining features of the financial analytics personality are supported via the new library functions described below. These routines may be called from C, C++, and Fortran.

6.1.1 Cumulative Normal Distribution Function

The Cumulative Normal Distribution Function (`cdf`), when called from a loop that was vectorized for execution on the coprocessor, will utilize the special instructions provided by the financial analytics personality to efficiently compute CDF values.

The `cdf` function is declared in the `cny_comp.h` include file, typically accessed with:

```
#include <convey/usr/cny_comp.h>
```

`cdf` returns a value of type `double`, and takes an argument of type `double`.

Example:

```
void calc_black_scholes(int no_opts, double *x, double *c, double *rate,
double *sigma, double *t, double *result)
{
  int i;
  double time_sqrt;
  double time_rate;
  double sigma_sqtime;
  double d1, d2;

  for(i=0;i<no_opts;i++){
     time_sqrt = sqrt(t[i]);
     time_rate = rate[i] * t[i];
     sigma_sqtime = sigma[i]*time_sqrt;
     d1 = (log(x[i]/c[i])+time_rate)/(sigma[i]*time_sqrt)+0.5*sigma_sqtime;
     d2 = d1-sigma_sqtime;
     result[i] = (x[i]*cdf(d1) - c[i]*exp(-time_rate)*cdf(d2));
  }
  return;
}
```

6.1.2 Mersenne-Twister Random Number Generators

There are three variants of the Mersenne-Twister random number generator:

- `double mersenne`(void);
- long `mersenne_64`(void);
- double `mersenne_tbl`(void);

`mersenne()` produces a uniform random double value between 0.0 and 1.0.

`mersenne_64()`produces a uniform random long value between `LONG_MIN` and `LONG_MAX`.

`mersenne_tbl()` produces a random double value from a table according to the distribution function used to load the table values. The default distribution is the Normal (or Gaussian) distribution. The default table size is 4M (4194304) entries. You can change the table size and/or use a custom function to load the table with these routines:

- `int fap_mt_set_table_size(FAP_MT_enumsize);`
- `int fap_mt_set_table_size_custom(FAP_MT_enumsize sz, double (*user_dist_func)(double));`

where `FAP_MT_enumsize` is:

```
typedef enum
  {FAP_MT_16K, FAP_MT_64K, FAP_MT_256K,
   FAP_MT_1M,  FAP_MT_4M,  FAP_MT_16M,
   FAP_MT_64M, FAP_MT_256M}
FAP_MT_enumsize;
```

These functions and enum type are also declared / defined in

the `cny_comp.h` include file, accessed with:

```
#include <convey/usr/cny_comp.h>
```

In all cases, changing the table size to any value other than the default of 4Meg causes a re-computation of values.

The `user_dist_func()` takes a double argument and returns a double value. This routine is called for each table index (as a percentage of the table size) and stores each table entry value, as in:

```
  for (idx = 0; idx < table_size; idx += 1) {
    double p = (double)(idx+idx+1) / (table_size * 2.0);
    table_array[idx] = (*user_dist_func)(p);
  }
```

The default cumulative Normal distribution is obtained when the `user_dist_func()` is the inverse normal distribution function.

Example:

```
#include <convey/usr/cny_comp.h>

double myrand(double d) {
   double f = drand48()+d;
   return f;
}

...

    int stat =  fap_mt_set_table_size_custom  (FAP_MT_16K, myrand);
    printf("fap_mt_set_table_size_custom stat=%d\n",stat);

#pragma cny begin_coproc
    for (i=0;i<N;i++) {
      db[i] = mersenne();
    }
#pragma cny end_coproc

...
```

The Mersenne-Twister internal seed/state can be changed with these host-only routines:

- `int fap_mt_load_state(FAP_MT_state_t);`
- `int fap_mt_save_state(FAP_MT_state_t);`

where `FAP_MT_state_t` is:

```
unsigned long FAP_MT_state_t[10][1024];
```

The default state can be used with a call to:

```
int fap_mt_use_default_state(void);
```

The default state can be reset with:

```
int fap_mt_init_default_state(void);
```

An example of the `mersenne_tbl()` function is the Monte-Carlo solution method of the above Black-Scholes example. Here we use `no_sims` random variables to estimate the solution for each option.

```
void calc_black_scholes_mc(int no_opts, int no_sims, double *x, double
*c, double *rate, double *sigma, double *t, double *result)
{
  int i, n;
  double R, SD;
  double inv_no_sims;
  double sum_payoffs;

  inv_no_sims = 1.0 / (double)no_sims;

  for(i=0;i<no_opts;i++){

    R = (rate[i] - 0.5 * sigma[i] * sigma[i]) * t[i];
    SD = sigma[i] * sqrt(t[i]);

    sum_payoffs = 0.0;
    for (n=0; n<no_sims; n++) {
      sum_payoffs += fmax(0.0,((x[i]*exp(R+SD*mersenne_tbl()))-c[i]));
    }

    result[i] = sum_payoffs * inv_no_sims;

  }

  for(i=0;i<no_opts;i++){
    result[i] = exp(-rate[i]*t[i]) * result[i];
  }

  return;
}
```

The following example shows how a Monte Carlo simulation might use the Mersenne Twister hardware in the Financial Analytics Personality to generate random numbers with a non-uniform distribution.

```
void montecarlo(rate, variance, t, x, c, result, no_opts, no_sims)
    double *rate;
    double *variance;
    double *t;
    double *x;
    double *c;
    double *result;
    int no_opts;
    int no_sims;
```

```
 {
#pragma cny migrate_coproc(rate, sizeof(double) * no_opts)
#pragma cny migrate_coproc(variance, sizeof(double) * no_opts)
#pragma cny migrate_coproc(t, sizeof(double) * no_opts)
#pragma cny migrate_coproc(x, sizeof(double) * no_opts)
#pragma cny migrate_coproc(c, sizeof(double) * no_opts)
#pragma cny migrate_coproc(result, sizeof(double) * no_opts)

  int i, n;
  double R, SD, exp_var, sum_payoffs;

#pragma cny begin_coproc

 for(i=0;i<no_opts;i++){

   R = (rate[i] - 0.5 * variance[i] * variance[i]) * t[i];
   SD = variance[i] * sqrt(t[i]);
   exp_var = exp(-rate[i]*t[i]);

   sum_payoffs = 0.0;
   for (n=0; n<no_sims; n++) {
     sum_payoffs += fmax(0.0,((x[i]*exp(R+SD*mersenne_tbl()))-c[i]));
   }

   result[i] = exp_var * (sum_payoffs * (1.0/no_sims));

 }
#pragma cny end_coproc


#pragma cny migrate_host(rate, sizeof(double) * no_opts)
#pragma cny migrate_host(variance, sizeof(double) * no_opts)
#pragma cny migrate_host(t, sizeof(double) * no_opts)
#pragma cny migrate_host(x, sizeof(double) * no_opts)
#pragma cny migrate_host(c, sizeof(double) * no_opts)
#pragma cny migrate_host(result, sizeof(double) * no_opts)

 return;
}
```

This routine can be compiled as follows:

```
$ cnycc -std=c99 -c  -mcny_sig=fap -mcny_vector montecarlo.c
VECTOR LOOP in montecarlo0 at 63: L 1 S 0 B 5 U 1 I 1 M 0
VECTOR IDIOMS in montecarlo0 at 63: R 1 L 0 S 0 C 0 X 0 P 0
```

In this exampe, the `mersenne_tbl()` intrinsic was used to obtain a random number in a non-uniform distribution. By default, it will use the Cumulative Normal Distribution with a default table size of 4M, as though

```
        fap_mt_set_table_size_custom(FAP_MT_4M, cdf());
```

had been explicitly called.

6.1.3 SOBOL Random Numbers

A sobol sequence of numbers can be generated with the sobol() function. Its usage is:

```
        double sobol(double );
```

The **sobol()** function is simply an identity function returning the value passed in. The values are created with a call to the host-only initialization routine:

```
        void fap_sobol_init(int dim, int n, double *rand_array);
```

dim is the number of spatial dimensions desired. **dim** must satisfy **1 <= dim <= 1111**.

n is the number of elements in each dimension of the (**dim * n)** multi-dimensional **rand_array**.

rand_array is a pointer to storage allocated by the caller which must be at least **dim*n*sizeof(double)** bytes.

Typically, a user may further massage the data, for example, producing a Gaussian sequence.

```
        #include <convey/usr/cny_comp.h>

          fap_sobol_init(2,(no_sims+1)/2,random_norms);


          random_norms2 = (double *)cny_cp_malloc((no_sims+1) * sizeof(double));
          // convert pair to gaussian ...
          for(i=0;i<no_sims;i+=2){

        convert_boxmuller(random_norms[i],random_norms[i+1],&random_norms2[i],
                          &random_norms2[i+1]);
          }

        void convert_boxmuller(double x,double y,double *result,double *result2)
        {    double R, T;
          /*
           * take x[m] and x[m+1] and convert to single gaussian result
           */
          R = -2.0 * log(x);
          T =  2.0 * M_PI * y;

           *result = sqrt(R) * sin(T);
           *result2 = sqrt(R) * cos(T);
          return;
        }
```

For more information on random number intrinsics provided by the Financial Analytics Personality, see the **libcny_fap** (3) man page.

6.1.4 Financial Analytics Example (using optimized intrinsics)

The following example illustrates the solution of the Black-Scholes options pricing model using standard intrinsics that utilize special Financial Analytics Personality instructions:

```
void blackscholes(int no_opts, double *x, double *c, double *t,
                  double *variance, double *rate, double *result)
{
  double time_sqrt;
  double time_rate;
  double var_sqtime;
  double d1, d2;
  int i;

#pragma cny migrate_coproc(x,        sizeof(double) * no_opts)
#pragma cny migrate_coproc(c,        sizeof(double) * no_opts)
#pragma cny migrate_coproc(t,        sizeof(double) * no_opts)
#pragma cny migrate_coproc(variance, sizeof(double) * no_opts)
#pragma cny migrate_coproc(rate,     sizeof(double) * no_opts)
#pragma cny migrate_coproc(result,   sizeof(double) * no_opts)

#pragma cny begin_coproc
#pragma cny unroll(4)
  for ( i = 0; i < no_opts; i++ ) {
    time_sqrt = sqrt(t[i]);
    time_rate = rate[i] * t[i];
    var_sqtime = variance[i] * time_sqrt;

    d1 = (log(x[i]/c[i])+time_rate)/(variance[i]*time_sqrt) +
          0.5*var_sqtime;
    d2 = d1-var_sqtime;

    result[i] = (x[i]*cdf(d1) - c[i]*exp(-time_rate)*cdf(d2));
  }
#pragma cny end_coproc

#pragma cny migrate_host(x,        sizeof(double) * no_opts)
#pragma cny migrate_host(c,        sizeof(double) * no_opts)
#pragma cny migrate_host(t,        sizeof(double) * no_opts)
#pragma cny migrate_host(variance, sizeof(double) * no_opts)
#pragma cny migrate_host(rate,     sizeof(double) * no_opts)
#pragma cny migrate_host(result,   sizeof(double) * no_opts)

  return;
}
```

This example can be compiled as follows:

```
$ cnycc -std=c99 -O3 -c -mcny_sig=fap -mcny_vector blackscholes.c
LOOP TRANSFORM in blackscholes0 at 19: D 0 I 0 X 5 N 0 S 0 A 0 U 0
VECTOR LOOP in blackscholes0 at 19: L 18 S 6 B 13 U 6 I 2 M 0
LOOP TRANSFORM in blackscholes0 at 17: D 0 I 0 X 5 N 0 S 0 A 0 U 1
```

In the source listing above, the **`sqrt()`**, **`exp()`**, **`log()`**, and **`cdf()`** functions are optimized, vector-based intrinsics provided by the Financial Analytics Personality. For additional performance, loop unrolling may be explicitly specified via a pragma at the top of the loop.

For more information about the intrinsics provided by the Financial Analytics Personality, please see the **`libcny_fap`**(3) manpage.

7 Linking an Executable that Contains Coprocessor Code

Convey's compilers should be used to link all applications that utilize the Convey coprocessor. If you have to use another vendor's driver to link an application, the following linker flags and arguments will need to be explicitly added to the link command, and Convey's linker[6] (**ld**) must also be used.

```
/opt/convey/lib/cny_initruntime.o
-ldl
-L/opt/convey/lib
-Wl,-rpath-link,/opt/convey/lib
-lcny_utils
```

The above flags and arguments are supplied by default when linking with Convey's compilers. They can be suppressed, when linking with Convey's compilers, via the option **–skipconveylink**.

The **–fPIC** flag should also always be used when linking objects that contain coprocessor regions, such as the Convey Mathematical Libraries.

[6] Convey's linker is required because the standard Linux linker doesn't handle the Convey added sections such as **.ctext**, **.cbss**, …

8 Executing Applications that Contain Coprocessor Code

8.1 Executing a Program using the Convey Coprocessor Simulator

A program containing Convey coprocessor routines may be executed just like any other program, as long as it was properly linked with the Convey compiler drivers (`cnycc`, `cnyCC`, or `cnyf90`).

If the `CNY_SIM_THREAD` environment variable is set to the Convey simulator's dynamic library (`libcpSimLib2.so`), the Convey simulator will be used to execute coprocessor routines, even if a Convey coprocessor is available.

```
export CNY_SIM_THREAD=libcpSimLib2.so     or
setenv CNY_SIM_THREAD libcpSimLib2.so
```

It may be necessary to set/modify the `LD_LIBRARY_PATH` environment variable to include `/opt/convey/lib` on some Linux distributions.

Note that the Convey simulator executes in a separate thread from the user program.

If the `CNY_SIM_WEAK_ORDER_CHECK` environment variable is set non-zero, the simulator will also check for potentially missing fences that may be needed due to the weak memory ordering characteristics of the Convey coprocessor. If you are getting correct answers using the Convey simulator, and (possibly intermittent) wrong answers when running the same executable on the actual coprocessor, try setting `CNY_SIM_WEAK_ORDER_CHECK` to "1" and running the application on the simulator again to see if any missing fence operations are detected. If so, this can be due to the compiler not properly detecting the need for a fence, or the program not conforming to a more aggressive aliasing model (`-std=c99`, `-mcny_alias_c99`, `-OPT:alias={restrict,disjoint,no_f90_pointer_alias}`), or use of the `no_loop_dep` directive/pragma where a dependence truly exists. If the program is compiled using any of the more aggressive aliasing models or any `no_loop_dep` directives/pragmas, make sure the program is truly obeying those assertions. If you are still having problems, please contact Convey Customer Support. Note that the simulator runs about 15% slower when checking for missing fences.

The sample program below (in the Assembler Usage chapter) was assembled, compiled, linked, and run with these commands:

```
$ /opt/convey/bin/as -bundling_off -g -o fib.o fib.s
$ /opt/convey/bin/cnycc -std=c99 -g -mcny_sig=sp  mainfib.c fib.o
$ ./a.out
 0  0
 1  1
 2  1
 3  2
 4  3
 5  5
 6  8
 7  13
 8  21
 …
```

The **-bundling_off** assembler flag is only needed when you want to single step one instruction at a time thru assembly language code with **gdb**. Turning off bundling may reduce coprocessor performance.

Here's an example debugging session with the same executable:

```
$ gdb a.out
(gdb) b fib_
Breakpoint 1 at 0x800000: file fib.s, line 19.
(gdb) r
Starting program: /mnt/bigdog/test/a.out
[Thread debugging using libthread_db enabled]
[New Thread 46912500013280 (LWP 5382)]
[New Thread 1075841344 (LWP 5385)]
[Switching to Thread 1075841344 (LWP 5385)]

Breakpoint 1, fib_ () at fib.s:19
19      fib_:   ldi.sq  $32, %s8        # loop counter
Current language:  auto; currently asm
(gdb) s
fib_ () at fib.s:22
22              ldi.sq  $0, %s1         # first term of the fibonacci series
(gdb) p $s8
$1 = {u64 = 32, f = {4.48415509e-44, 0}, d = 1.5810100666919889e-322}
(gdb) s
fib_ () at fib.s:24
24              ldi.sq  $1, %s2         # second term of the fibonacci series
(gdb) s
fib_ () at fib.s:27
27              ldi.sq  $1, %s9         # %s9 is 1, used to count down the
loop index
(gdb) s
fib_ () at fib.s:30
30              st.sq   %s1, $0(%a9)    # store the first term of the series
(gdb) s
fib_ () at fib.s:31
31              add.sq  %a9, %a7, %a9   # bump up the result array pointer
(gdb) s
fib_ () at fib.s:32
32              st.sq   %s2, $0(%a9)    # store the second term of the series
(gdb) s
fib_ () at fib.s:33
33              add.sq  %a9, %a7, %a9   # bump up the result array pointer
(gdb) s
loop () at fib.s:36
36      loop:   add.sq  %s1, %s2, %s3   # compute the next term  x[i+1]
(gdb) s
loop () at fib.s:37
37              st.sq   %s3, $0(%a9)    # store the next term into result
array
(gdb) p $s3
$2 = {u64 = 1, f = {1.40129846e-45, 0}, d = 4.9406564584124654e-324}
```

Note that when `gdb` stops execution at a breakpoint, that particular instruction or statement has not been executed yet.

The assembler requires a '%' to specify a register name, while gdb requires a '$'. **gdb** will show a '%' when disassembling code.

If assembly source code is assembled with instruction bundling on, the automatic disassembly of instructions at each statement boundary will only show one instruction, even though there may be two instructions executed when single stepping through coprocessor code. Instruction bundling will also reorder some instructions to facilitate bundling.

More details on debugging can be found in **Debugging Coprocessor Code with gdb**.

8.2 Executing a Program using the Convey Coprocessor

Executing a program compiled for the Convey coprocessor does not require any special invocation. Simply run the executable as you would normally:

```
$ ./a.out
```

Be sure, however, that the `CNY_SIM_THREAD` environment variable is not set to prevent the program from running on the Convey Simulator.

The behavior of the coprocessor can be further controlled by setting environment variables. The environment variables listed below are the most frequently used. If the default behavior is not what you desired, set them as described in 12 **Environment Variable Summary**.

- Report how many times the program dispatched code to the coprocessor:
  - `CNY_CALL_STATS`
- To run on the host processor only (e.g. to collect host performance, results, etc. for comparison against a coprocessor run):
  - `CNY_NO_COPROC`
- To control how the application behaves if the coprocessor isn't immediately available:
  - `CNY_INIT_TIMEOUT, CNY_COPROC_OR_FAIL`
- To control how the host waits for a coprocessor dispatch to complete:
  - `CNY_WAIT_STRATEGY, CNY_WAIT_STRATEGY_TIME`
- To control other aspects of the coprocessor:
  - `CNY_PERSISTENT, CNY_PRECISE_TRAPS, CNY_SE_MASK, CNY_AE_MASK, CNY_STACK_SIZE`
- To control sharing of the coprocessor by a group of cooperating processes:
  - `CNY_PROG_MODEL`

If your application was compiled by another vendor's compilers, and called any Intel® MKL library routines, and you want to use Convey's Mathematical Library (CML) routines instead, set the `LD_PRELOAD` environment variable appropriately. For example, to use the single precision version of the CML library:

```
export LD_PRELOAD=/opt/convey/cml/lib/libcml_linalg_sp.so
```
or

```
setenv  LD_PRELOAD /opt/convey/cml/lib/libcml_linalg_sp.so
```

See the **Convey Mathematical Libraries Guide** for more information about these libraries.

8.3 Process Groups

The environment variable `CNY_PROG_MODEL` may be set to `PROCESS_EXCLUSIVE` (the default behavior) or `PROCESS_SHARED`. When set to `PROCESS_EXCLUSIVE` (or not set at all), a process will either attach the coprocessor at program startup or not attach at all, and if attached, the coprocessor will not be shared with any other processes.

When `CNY_PROG_MODEL` is set to `PROCESS_SHARED`, a group of cooperating processes, in the same process group[7], may share access to the coprocessor, subject to the limitations described below.

- This first process in the process group that starts up will attempt to attach the coprocessor. If the coprocessor is already attached by a process not in the same process group, the first process will either wait for the coprocessor, continue executing with attaching the coprocessor, or abort, depending on the environment variables `CNY_INIT_TIMEOUT` and `CNY_COPROC_OR_FAIL`. The other processes in the process group should not be initiated until the first process acquires the coprocessor, or those other processes should set `CNY_INIT_TIMEOUT` sufficiently large (default timeout is five seconds when `CNY_PROG_MODEL` is `PROCESS_SHARED`). Note that the wait strategy for the initial coprocessor attach is `SPIN`, which means the host processor will continually check for availability of the coprocessor. If the coprocessor is not available and `CNY_COPROC_OR_FAIL` is not set, the application will continue executing (after the attach timeout period has elasped), and only execute host code.
- If the cooperating processes expect to share coprocessor state, such as register contents, the environment variable `CNY_PERSISTENT` should be set to `1`.
- A dispatch wait strategy (`CNY_WAIT_STRATEGY`) of `POLL` is recommended.
- A dispatch wait time (`CNY_WAIT_STRATEGY_TIME`) less than 100µsec is not recommended, especially if there are more than a few processes in the process group.
- When processes in a process group are sharing pages (mmap) and one process tries to migrate a shared page, that page will only be migrated if is safe to do so in every process sharing that page.

A "process group" is a Linux concept, and a tutorial explaining process groups is in Appendix H **Process Group Tutorial**. MPI manages the progress groups automatically, but non-MPI applications that want to share the coprocessor will need to manage the process group.

8.4 MPI Example

Convey provides an MPI implementation in the package named "`convey-openmpi`", which can be installed by your system administrator. This package may be installed on both Convey servers and on compatible cross-development machines.

[7] MPI implementations typically use process groups. A MPI process group can share the coprocessor simply by setting `CNY_PROG_MODEL` to `PROCESS_SHARED`. See 8.4 **MPI Example** for an MPI example program.

Here is a very short sample MPI program run using the bash shell:

Always set these environment variables first:

```
export MPI_HOME=/opt/openmpi-1.4.0
export OMPI_CC=cnycc
export OMPI_FC=cnyf90
export OMPI_CXX=cnyCC
```

Set the environment variable appropriate for an MPI job

```
export CNY_CALL_STATS=1
export CNY_PROG_MODEL=PROCESS_SHARED
export CNY_INIT_TIMEOUT=1000
```

Use the compiler wrappers which add libraries and header files for MPI

```
$MPI_HOME/bin/mpicxx
$MPI_HOME/bin/mpicc
$MPI_HOME/bin/mpif90
```

Example:

```
$MPI_HOME/bin/mpicc  -mcny hello_cny_mpi.c
```

Under the covers, here is what the wrapper does:

```
$ $MPI_HOME/bin/mpicc  hello_cny_mpi.c -show -std=c99 -mcny
cnycc -I/opt/openmpi-1.4.0/include -pthread hello_c.c
-L/opt/openmpi-1.4.0/lib -lmpi -lopen-rte -lopen-pal -ldl
-Wl,--export-dynamic -lnsl -lutil -lm -ldl -std=c99 -mcny
```

To run the executable we need a list of hosts …

```
$ cat listofhosts
baddog
baddog

$ $MPI_HOME/bin/mpirun -machinefile listofhosts -np 4  ./a.out
Hello, world, I am 3 of 4
Hello, world, I am 2 of 4
Hello, world, I am 0 of 4
Hello, world, I am 1 of 4
Hello, world, I am 3 of 4
Hello, world, I am 0 of 4
Hello, world, I am 2 of 4
Hello, world, I am 1 of 4
Hello, world, I am 3 of 4
Hello, world, I am 0 of 4
Hello, world, I am 2 of 4
Hello, world, I am 1 of 4
Hello, world, I am 3 of 4
Hello, world, I am 2 of 4
Hello, world, I am 0 of 4
```

```
Hello, world, I am 1 of 4
Hello, world, I am 3 of 4
Hello, world, I am 0 of 4
Hello, world, I am 2 of 4
Hello, world, I am 1 of 4
0 1
1 1
3 1
2 1
coprocessor calls: 5
coprocessor calls: 5
coprocessor calls: 5
coprocessor calls: 5
```

Below is the source file (`hello_cny_mpi.c`) used in the above example:

```
 /*
 * Copyright (c) 2004-2006 The Trustees of Indiana University and
Indiana
 *                           University Research and Technology
 *                           Corporation.  All rights reserved.
 * Copyright (c) 2006        Cisco Systems, Inc.  All rights reserved.
 *
 * Sample MPI "hello world" application in C
 */

#include <stdio.h>
#include "mpi.h"

int main(int argc, char* argv[])
{
    int rank, size;
    int array [100];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
  int i;
  for (i=0;i< 5; i++) {
    printf("Hello, world, I am %d of %d\n", rank, size);
    array[rank] = 0;
    sleep(1);
#pragma cny begin_coproc
    array[rank] = 1;
#pragma cny end_coproc
   }
    MPI_Barrier(MPI_COMM_WORLD);
    printf("%d %d\n",rank, array[rank]);
    MPI_Finalize();

    return 0;
}
```

The various sub-processes that MPI creates are all members of the same process group. If `CNY_PROG_MODEL` is set to "`PROCESS_SHARED`", all the MPI sub-processes can attach the coprocessor concurrently. Note that only one coprocessor routine or region can execute at a time, so careful consideration should be given to proper scheduling of the coprocessor, when to execute alternative host code for **`dual_target`** routines, appropriate memory migration, etc.

9 User Callable Support Routines

The following library routines are available to programs linked with the Convey compiler drivers (`cnycc`, `cnyCC`, `cnyf95`).

Many of these routines, when called from C, require the `<convey/usr/cny_comp.h>` header file to be included.

Some of these routines are only callable from host code (as indicated in the table below). If called from coprocessor code (due to using `-mcny_dual_target`), unresolved external references will result.

Notes on the Fortran callable versions of these routines (if not otherwise explicitly documented in the table):

- The Fortran callable version of most routines have the same apparent name as the C callable version (the Fortran compiler implicitly appends a trailing underscore to most routine names).
- The arguments to the Fortran callable version of a routine do not need any special treatment, such as %VAL, but must be the correct type (`int` and `size_t` C types are equivalent to `INTEGER*4`, and long is equivalent to `INTEGER*8`).
- Where a C structure is used as an argument, the Fortran routine should pass a sufficiently large `INTEGER` array, packed appropriately. See `/opt/convey/include/convey/sys/cny_user.h` for the definition of the C structure types used in these routines.
- When a C `enum` type is passed in or returned, Fortran code should use an INTEGER*4 variable/constant. The `enum` values are present in the `cny_user.h` include file.

`copcall` **Interface Routines** *Can only be called from host code*	Convey provides a variety of interface routines, callable from C and Fortran host code, which invoke a coprocessor routine, passing in arguments and returning a scalar result. See Appendix G **Low Level Interfaces to Coprocessor Routines**. A runtime invocation of a `copcall` interface routine makes the application non-portable to non-Convey systems, unless the (very slow) Convey simulator is used to execute coprocessor code.

Data Migration Routines *Can only be called from host code*	The `cny_migrate_data` routine gives hints to the kernel about future data access to the specified memory by the coprocessor. The kernel will attempt to migrate the specified memory to coprocessor memory before the next dispatch of a coprocessor routine or region. This routine may be called many times before a dispatch, and the kernel may attempt to coalesce various requests and migrate large contiguous areas when possible. All outstanding migration requests will be processed before the next dispatch is initiated. `cny_status_t cny_migrate_data (void * data, size_t size, cny_node_t location)` `data` is a pointer to the start of the data region to be migrated `size` is the size of the data region in bytes `location` is the target location for the migration (the node ID to migrate data to, either `CNY_LOCAL` or `CNY_COPROCESSOR`). Also see `cny_numa_id()` and `cny_cp_numa_id()` below. Note that using the compiler `migrate_coproc` or `migrate_host` pragmas is equivalent to calling `cny_migrate_data`.

Coprocessor Memory Allocation Routines
 	
↑
All of the coprocessor memory allocation routines can only be called from host code
↓
	`cny_cp_malloc` is similar to `malloc`, but allocates coprocessor memory (if possible). A call to `cny_cp_malloc` returns a pointer to the allocated memory. A `NULL` (zero) return value indicates the allocation failed.
	
	`void * cny_cp_malloc (size_t size)`
	`size` is the number of bytes of coprocessor memory to allocate
	
	`cny_cp_free` frees previously allocated coprocessor memory.
	
	`void cny_cp_free(cny_data_t data)`
	
	`data` is a pointer to the memory to be freed (must be a value previously returned by `cny_cp_malloc`)
	
	`cny_cp_realloc` changes the size of the memory block pointed to by `data` to `size` bytes. The initial contents of the memory block will be unchanged (from the start until the minimum of the old and new sizes); newly allocated memory will be uninitialized. If `data` is `NULL`, the call is equivalent to `malloc(size)`. If `size` is equal to zero, the call is equivalent to `free(data)`. Unless `data` is `NULL`, it must have been returned by an earlier call to `cny_cpmalloc()`.
	
	If the area pointed to by data was moved, a `cny_cp_free(data)` is done.
	
	`void * cny_cp_realloc(void* data, size_t size)`
	
	`data` is a pointer to the memory block to be reallocated
	
	`size` is the size of the new (reallocated) block (in bytes)

Signature Manipulation Routines

Fortran callable

```
integer*8  sig1, sig2
integer  stat
character *5 perName
…
call cny_f_get_signature(perName, sig1, sig2, stat)
```

perName should be a character string/variable, containing a partial signature, usually a personality name, such as "sp".

stat is set to non-zero if a signature matching the personality name or number was not found, and is set to zero if a matching signature was found and returned in sig1.

stat is set to 1 when the underlying Convey support library was not available.

The 64-bit signature returned in sig1 is suitable for use with the copcall routines. sig2 will contain a partially resolved signature, reserved for future use.

↑

Can only be called from host code

↓

C callable

```
#include <stdio.h>
#include <convey/usr/cny_comp.h>
cny_image_t sig1, sig2;
int stat;
…
cny_get_signature ("sp", &sig1, &sig2, &stat);
if (stat != 0)
  fprintf(stderr,"cny_get_signaure failed\n"), exit(1);
```

cny_get_signature returns a fully resolved signature in **sig1** that corresponds to the partially resolved signature passed in as the first argument.

stat is set to non-zero if a signature matching the personality name or number was not found, and is set to zero if a matching signature was found and returned in sig1.

stat is set to 1 when the underlying Convey support library was not available.

The 64-bit signature returned in sig1 is suitable for use with the copcall routines. sig2 is reserved for future use.

perName should be a character string/variable, containing a partial signature, usually a personality nickname, such as "sp".

Message Routines

Allow coprocessor code to create a message that is printed when the coprocessor routine returns to the host. Host versions also exist that print the message immediately, allowing these routines to be used within coprocessor regions.

The following interfaces allow simple message and variable output from the coprocessor. The first argument (`msg`) is a string to prepend to the message. The second argument (`mlen`) is the number of characters of `msg` to prepend. The third argument (`code`) is the value to be printed (using an appropriate format) after the string `msg`.

The collection and display of data is enabled by setting the environment variable `CNY_SIMCALL_MSGTRACE=1`. The routines store the string/values to print in an internal buffer, and when the dispatched coprocessor code finishes, the host writes the buffered values to `stdout`.

C callable

These functions silently return if `CNY_SIMCALL_MSGTRACE` is not set.

```
void cny$mput (char *msg, int mlen) ;
void cny$mputi(char *msg, int mlen, int  code) ;
void cny$mputl(char *msg, int mlen, long  code) ;
void cny$mputf(char *msg, int mlen, float  code) ;
void cny$mputd(char *msg, int mlen, double  code) ;
void cny$mputv(char *msg, int mlen, long * code) ;
```

(`mputv` is like `mputl`, but takes an address)

Example:

```
int main() {
#pragma cny begin_coproc
  cny$mput("cny$mput",8);
  cny$mputi("cny$mputi",9,11);
  cny$mputl("cny$mputl",9,12L);
  cny$mputl("cny$mputv",9,13L);
  cny$mputf("cny$mputf",9,12.1);
  cny$mputd("cny$mputd",9,12.2);
#pragma cny end_coproc
  printf ("Done\n");
  return 0;
}
```

↑

Can be called from host or coprocessor code

↓

Fortran callable

For these Fortran versions of these routines, `msg` should be a quoted string or a CHARACTER variable, and `mlen` must be an integer value, the number of characters in `msg`.

```
integer ival
integer*8 lval
real rval
real*8 dpval
…
call cny$mput(msg, mlen)
call cny$mputi(msg, mlen, ival)
call cny$mputl(msg, mlen, lval)
call cny$mputf(msg, mlen, rval)
call cny$mputd(msg, mlen, dpval)
```

Precise Traps
 	
May be called from host and coprocessor code
 	
C callable
 	
Fortran callable
	Enable or disable precise traps on the coprocessor. Enabling precise traps can be a useful debugging tool, but coprocessor code runs significantly slower when precise traps are enabled.
	
	The simulator always generates precise traps.
	
	The argument to these routines should be zero or one (typed `long` or `integer*8`). Zero disables precise traps, one enables them.
	
	These routines do not return a value.
	
	`#include <convey/usr/cny_comp.h>`
	`cny_coproc_precise(1L);`
	
	and
	
	`call cny_coproc_precise(1_8)`

Exception Masks
 	
May be called from host and coprocessor code
	These routines get, set, or clear bits in the SE (scalar engine) and AE (application engine) exception masks.
	
	The "**get**" routines return the current mask. The "**set**" routines set the mask to the value passed in. The **clear_mask_bits** and **set_mask_bits** routines clear/set the bits in the exception mask where the corresponding bit in `value` is a one bit, preserving the original value for all other bits.
	
	`long se_get_mask()`
	`long ae_get_mask()`
	`void se_set_mask(long value)`
	`void ae_set_mask(long value)`
	`void se_clear_mask_bits(long value)`
	`void ae_clear_mask_bits(long value)`
	`void se_set_mask_bits(long value)`
	`void ae_set_mask_bits(long value)`
	
	The exception mask bits are defined in `/opt/convey/include/convey/usr/cny_coproc_prog.h` and `cny_coproc_f90prog.h`.

VPM register C and Fortran versions available	The coprocessor's VPM register may be set from coprocessor code with this routine: `void set_vpm(long value)` The host version of this routine does nothing.

Query Functions Except for `cit_read`, these can only be called from host code	`long cit_read ()` Returns the coprocessor interval timer value, when called from coprocessor code. The host version of this routine returns -1.
	`void cny_cp_attributes(cny_cp_attr_t * attr)` Returns the system attributes for a Convey node in `attr`.
	`cny_interleave_t cny_cp_interleave()` Returns the memory interleave in effect (a boot time option). One of `CNY_MI_UNDEF` (not running on a Convey server), `CNY_MI_BINARY`, or `CNY_MI_3131`
	`int cny_core_id ()` Returns the processor ID for the currently executing thread.
	`int cny_numa_id ()` Returns the node ID for the currently executing thread.
	`int cny_cp_numa_id ()` If a coprocessor exists on the system, this returns the node ID for the Convey coprocessor; otherwise, it returns -1.
	`int cny_cp_pagesize ()` If a coprocessor exists on the system, this returns the maximum supported page size for the coprocessor; otherwise, it returns the maximum page size for the host.
	`int cny_cp_mem_size ()` If a coprocessor exists on the system, this returns the amount of memory on the coprocessor board; otherwise it returns zero.

More Query Functions
 	
Except for **cit_read**, these can only be called from host code
	`cny_status_t cny_memory_locale (void * data, size_t size, cny_node_t * location)`
	
	Returns the locale(s) of the specified memory region in `location` (an `enum`, see `cny_user.h`)
	
	`data` is a pointer to the memory of interest
	
	`size` is the size of the region in bytes
	
	--- 	
	
	`char * cny_str_locale (cny_node_t x)`
	
	Returns a string describing the location of `cny_node_t` enum `x`.
	
	Commonly used with the enum value returned in `location` by the `cny_memory_locale` routine.
	
	Note: Although the Fortran callable version of this routine exists, it returns a C string. The Fortran caller should scan the returned string for a zero byte, and only access the string value before that zero byte.

10 Assembler Usage

The Convey assembler is named **as**, and can compile x86-64 assembly language as well as Convey coprocessor assembly language.

The Convey assembler is an enhanced version of the GNU assembler for the x86-64 architecture, and uses the same syntax.

The Convey assembler supports several new sections, named **.ctext**, **.cdata**, and **.cbss**. The new sections are equivalent to the traditional **.text**, **.data**, and **.bss**, except their corresponding physical memory pages will be loaded into coprocessor resident memory (when they are first paged into the process) for faster access by the coprocessor. **.ctext** sections typically contain Convey assembly language. These new sections should not be used for memory that is primarily accessed by host code, due to the performance impact.

The Convey assembler, although it compiles code for the x86-64 architecture, has numerous syntactical differences compared to the Intel® assembler. The most significant differences are:

- Immediate operands are preceded by a '$'
- Register operands are preceded by a '%'
- The destination operand in an instruction is the last operand, not the first
- For some instructions, the size of memory operands is determined by the last character of the instruction's opcode name

The GNU assembler manual (www.gnu.org/software/binutils/manual/gas-2.9.1) describes non-architectural features of the GNU assembler.

Intel® provides Instruction Set Reference manuals for the Intel® 64 architecture at www.intel.com/products/processor/manuals/index.htm.

The Convey instruction set is described in the Convey Coprocessor Reference Manual.

Commonly used Convey assembler flags:

-al	Turn on the assembly listing (written to stdout).
-bundling_not_allow	Turn off all instruction bundling, even when **.bundle** directives are present in the source code.
-bundling_off	Turn instruction bundling off, unless a **.bundle next2** directive is used. Useful when using **gdb** to debug assembly language routines.
-bundling_on	Turn on instruction bundling. This is the default bundling mode.
-g	Turn on debug information. Needed to support source level debugging with **gdb**.

`-o` *`filename`*	Uses ***`filename`*** as the output file, typically needed to produce a **`.o`** file (*filename* defaults to ***`a.out`*** if **`-o`** is not specified).

The following environment variable affects the Convey assembler:

Environment variable name	Permissible values	Effect
`CNY_GAS_BUNDLE`	<u>`on`</u> `off`	Turn instruction bundling **`on`** or **`off`**.

The underlined values are the default if the environment variable is not present and no equivalent assembler flags were present.

10.1 Important Assembler Usage Notes

When writing Convey assembly code, it is recommended that you use the general form of the sample coprocessor assembly code shown in chapter titled **Sample coprocessor assembly code**. In particular:

- Coprocessor instructions should always appear in a `.ctext` section.
- A `.signature` statement should always be included, particularly for custom personalities, to help the assembler diagnose misuse of instructions from other personalities. The debugger also makes use of the specified signature (indirectly, through the coprocessor's CDS register) to determine how to disassemble instructions in an assembly language routine.
- A `.globl` statement and a `.type` statement should be used to identify the external name of the routine.
- The coprocessor routine should always execute a `rtn` statement to return to its caller or complete a dispatch.
- The Convey compilers, when generating a call to a coprocessor routine, prefix the external procedure name with "`cny_`". This renaming does not occur when using the low level `copcall` interfaces.
- The assembler does not add the `cny_` prefix to any routine names referenced in assembly language source code. If an assembly language routine wants to call a coprocessor routine that was compiled with the C, C++, or Fortran compilers (using a flag such as `-mcny_dual_target`, or `-mcny_pure`), it must explicitly prefix the routine name with `cny_`. If that coprocessor routine was compiled with the Fortran compiler, a trailing "_" must also be appended to the name[8]. Similarly, the entry point to the assembly language routine must be renamed in the same manner if the assembly language routine is to be called from compiler generated code.

[8] Unless the **`-fno-underscoring`** flag is used

- The restrictions on coprocessor code described in chapter 4.2 **Host Code and Coprocessor Code Interactions and Limitations** apply to coprocessor assembly language routines as well.

10.2 Sample coprocessor assembly code

This routine may be assembled with the command

```
    $ /opt/convey/bin/as  -g  -o  fib.o  fib.s
```

```
#
# fib.s : fibonacci series example
#
#       compute the series where
#       x[1]   = 0
#       x[2]   = 1
#       x[i] = x[i-1] + x[i-2]          for i = 3..32
#
#       store the results in the external variable 'result_'
#
        .section .ctext, "ax",@progbits
        .section .ctext
        .signature    2  # any signature works for canonical instructions

        .globl  fib_
        .globl  result_
        .type   fib_, @function

fib_:   ldi.sq  $30, %s8        # loop counter
                                #     (32 terms less the two initial terms)
        ldi.uq  result_, %a9    # we'll store the answers indirect thru %a9

        ldi.sq  $0, %s1         # first term of the fibonacci series
                                #     %s1 will be x[i-1]
        ldi.sq  $1, %s2         # second term of the fibonacci series
                                #     %s2 will be x[i]

        ldi.sq  $1, %s9         # %s9 is 1, used to count down the loop index
        ldi.sq  $8, %a7         # %a7 is 8 (bytes), the data size we're storing

        st.sq   %s1, $0(%a9)    # store the first term of the series
        add.sq  %a9, %a7, %a9   # bump up the result array pointer
        st.sq   %s2, $0(%a9)    # store the second term of the series
        add.sq  %a9, %a7, %a9   # bump up the result array pointer

#       here's the main loop
loop:   add.sq  %s1, %s2, %s3   # compute the next term  x[i+1]
        st.sq   %s3, $0(%a9)    # store the next term into result array
        add.sq  %a9, %a7, %a9   # bump up the result array pointer
        mov     %s2, %s1        # shift x[i]   -> x[i-1]
        mov     %s3, %s2        #   and x[i+1] -> x[i]
        sub.sq  %s8, %s9, %s8   # decrement loop counter
        cmp.sq  %s8, %s0, %sc0  # compare count down loop counter against zero
        br      %sc0.gt,loop    # loop until counter reaches zero
        rtn
```

11 Coprocessor Assembly Language Calling Conventions

11.1 C/C++ Calling Conventions

An assembly language routine called from coprocessor code in a region compiled from C/C++ code should use the following conventions for accessing arguments, returning results, and preserving reserved registers:

- A0 and S0 are always zero
- A1-A7 are reserved for the compilers use, do not modify them
- Addresses and integer values are passed in A registers, starting with A8
- Floating point values are passed in S registers, starting with S1
- If the coprocessor routine returns an INTEGER value, it should return it in A8.
- If the coprocessor routine returns a REAL value, it should return it in S1.

Example:

If the coproc region looks like:

```
#pragma cny begin_coproc
        myint = 1;
        mycr(&myint, 2, "12345", 3.0, "abc");
#pragma cny end_coproc
```

then the coprocessor version of that region will call the coprocessor routine `cny_mycr` passing

- the address of the variable “one” in A8,
- the 64 bit integer value “2” in A9,
- the address of the null terminated string “12345” in A10,
- the floating point value “3.0” in S1,
- the address of the null terminated string “abc” in A11,

11.2 Fortran Calling Conventions

An assembly language routine called from coprocessor code in a region compiled from Fortran code should use the following conventions for accessing arguments, returning results, and preserving reserved registers:

- A0 and S0 are always zero
- A1-A7 are reserved for the compilers use, do not modify them
- Almost all arguments are passed *by address*. The first in A8, the second in A9, …
- INTEGER type arguments passed with `%VAL()` are passed in the A registers also, in sequence with the *by address* arguments, but are passed as a 64 bit value.

- `REAL` type arguments passed with `%VAL()` are passed in the S registers, starting with S1, as a 64 bit floating point value.
- For each character type dummy argument, a hidden extra argument is appended onto the end of the dummy argument list (in the order the character typed dummy arguments appeared in the actual procedure call), and the length of each character typed dummy argument is passed, by value, in an A register.
- If the coprocessor routine returns an `INTEGER` value, it should return it in A8.
- If the coprocessor routine returns a `REAL` value, it should return it in S1.
- To return an array result, pass in the address of the result array as an argument
- Assembly language routines called from a Fortran compiled region should not have a visible interface that includes any dummy arguments that
    - are assumed shape arrays,
    - have the `POINTER` or `TARGET` attribute,
    - are KEYWORD actual arguments,
    - are derived types,

  nor should those assembly language routines have an explicit interface that defines the procedure as
    - returning an array or a Fortran `POINTER`,
    - returning a dynamically sized character value,
    - a generic procedure,
    - implementing a user-defined operator or user-defined assignment

Example:

If the coproc region looks like:

```
!$cny begin_coproc
        call mycr(1, %val(2), "12345", %val(3.0), "abc")
!$cny end_coproc
```

then the coprocessor version of that region will call the coprocessor routine `cny_mycr` passing

- the address of the constant 1 in A8,
- the 64 bit integer value “2” in A9,
- the address of the string “12345” in A10,
- the floating point value “3.0” in S1,
- the address of the string “abc” in A11,
- the integer value “5” in A12 (length of “12345”), and
- the integer value “3” in A13 (length of “abc”)

12 Environment Variable Summary

The following environment variables are used by various Convey Software products. They are typically set by the user via a shell command[9], possibly in one of the shell initialization scripts[10].

Warning: `gdb` invokes a new shell to execute the program being debugged. Any environment variables that are initialized in `.cshrc`, `.bashrc`, or other similar shell initialization scripts may change the values of any environment variables set therein, leading to different behavior when using `gdb` compared to running the program without `gdb`. For this reason, Convey recommends that `.login`, `.profile`, and `.bash_login` (or other personal initialization scripts that are only executed by login shells) be used to initialize the following environment variables.

12.1 Convey Programming Model Environment Variables

This chapter lists the environment variables that control execution of a program containing Convey Coprocessor instructions, whether executing on an actual Convey coprocessor or the Convey simulator. Most of these variables are queried by the Convey runtime library when a process first starts up.

Unless explicitly described otherwise, these environment variables should be set to numeric values.

Environment Variable	Description (default value)
`CNY_AE_MASK`	Application engine coprocessor exceptions may be masked by setting this environment variable to one or more of the following exception names: **AEUIE AEFIE AEFDE AEFZE AEFOE AEFUE AEIIE AEIZE AEIOE AERRE AEURE AESOE AEERE** i.e. `export CNY_AE_MASK="AEFUE,AESOE,AEIOE"` Commas or spaces may be used to separate the exception names, or a single hex value may be specified (`"0x120"`). The various exception names are documented in the Convey Reference Manual. The default exception mask (**"AEFUE,AESOE,AEIOE,AEFDE"**) causes vector floating point underflows in the coprocessor to produce a 0.0 value, vector integer op and vector store overflows to be ignored, and Denorms to be ignored. Not all of these values are necessarily meaningful when using a custom or non-floating point personality.

[9] Bash & sh shells: **export variable=value**, csh & tcsh shells: **setenv variable value**

[10] Preferred initialization scripts are **.profile**, **.login**, **.bash_profile**, and **.bash_login**. Please avoid using **.bashrc**, **.cshrc**, **.kshrc**, **.tcshrc**, and other scripts that are executed by non-login shells upon startup.

Environment Variable	Description (default value)
CNY_ALWAYS_USE_COPROC	When non-zero, assume it is always profitable to use the coprocessor in a coprocessor region. i.e. `export CNY_ALWAYS_USE_COPROC=1` The default value is 0.
CNY_CALL_STATS	When non-zero, print out the number of dispatches performed during a process' lifetime (at process exit). i.e. `export CNY_CALL_STATS=1` The default value is 0.
CNY_CALLABLE_NAMES	If set, should be a filename containing routine names. See the `-mcny_callable_names` flag for more information.
CNY_COPROC_OR_FAIL	When non-zero, the process will abort if the coprocessor is not available at process startup. i.e. `export CNY_COPROC_OR_FAIL=1` The default value is 0.
CNY_DEFAULT_IMAGES	A list of signatures that are preloaded into the coprocessor's image cache on the management processor.
CNY_DISPATCH_METHOD	If 0, invokes a wrapper on every dispatch to set the AEC and CPC registers (includes VPM) and then jumps to the target routine. The default is 0. If 1, compiler assumes *persistence*, and will only call the wrapper for the first dispatch. Will cause aborts if *persistence* is not specified for every dispatch.
CNY_EXCLUDE_NAMES	If set, should be a filename containing routine names. See the `-mcny_exclude_names` flag for more information.
CNY_FORCE_SIG	When non-zero, use the specified signature for all dispatches. Default value is 0.
CNY_GAS_BUNDLE	When set to "**off**", disables instruction bundling in the assembler. Default value is "**on**". This affects all coprocessor code generation from Convey's compilers also.
CNY_IGNORE_SIGNATURE_CHECK	If non-zero, signatures will not be checked against the signature database nor subjected to upgrading. Default value is 0.

Environment Variable	Description (default value)
CNY_INIT_TIMEOUT	Specifies how long a process should wait, in seconds, at process startup, for the coprocessor to become available. Default is 0 seconds when CNY_PROG_MODEL is PROCESS_EXCLUSIVE and 5 seconds when CNY_PROG_MODEL is PROCESS_SHARED. If the allotted time has elapsed, and the coprocessor is still not available, the application, by default, will run without attaching the coprocessor, executing only host code. You can force the application to abort if the coprocessor is not available by also setting the CNY_COPROC_OR_FAIL flag.
CNY_LOAD_IMAGE	This environment variable specifies the initial signature image to be loaded into the Convey coprocessor for the next Convey program execution. This is only useful for programs requiring multiple signatures. In this case, CNY_LOAD_IMAGE specifies the signature image to be loaded first into the coprocessor. If this variable does not exist or is set to an invalid signature, the signature images required by the program will be cached on the coprocessor but none will be loaded. Loading will be performed by a first fault technique. All signature images required by a program are always cached on the coprocessor by default. Loading implies the physical loading of the AE FPGAs with the image contents. This environment variable has no affect on the Convey simulator. Example: `export CNY_LOAD_IMAGE=2.1.1.1.0`
CNY_MIGRATE_TRACE	When non-zero, enables tracing of the data migration calls; includes info on address, size, and location. Default value is 0.
CNY_NO_COPROC	When non-zero, all coprocessor regions/routines will execute the host version of the region/routine only. Default value is 0. Calls to any of the copcall routines will fail when this environment variable is set non-zero.
CNY_NO_CPMEM	When non-zero, no data will be allocated on or migrated to the coprocessor's memory pool. Default value is 0.
CNY_NO_SIM_COPROC_CHECK	When non-zero, suppresses the simulator's warning message that a coprocessor was detected, but you are running on the simulator. Default value is 0.

Environment Variable	Description (default value)
CNY_PDK_AE_MASK	If set, the 16bit mask value will be used to initialize the application engine's exception mask for a custom personality. Default value is 0.
CNY_PDK_CLIENT_MODE	This environment variable controls the startup of the custom personality client user simulation. If this variable has the value "**debug**", the custom personality client will start in a window under control of the **gdb** debugger. **gdb** and **xterm** must be in the user's path. If this variable has the value "**win**", the custom personality client will start in a window to allow print statements to be visible. **xterm** must be in the users path. If CNY_PDK_CLIENT_MODE is undefined, the client will start as a child process of the simulation.
CNY_PERSISTENT	When non-zero, allows the coprocessor's state to persist from one coprocessor dispatch to the next (within a single process). This allows the values of coprocessor registers set in one dispatched coprocessor region to be referenced during a subsequent dispatch. Default value is 0.
CNY_PERSONALITY_PATH	Specifies where to find the personality database and files. Default value is /opt/convey/personalities. Users of Convey's PDK should typically use the updateCustomDB script to enter a new PDK personality in the personality database, rather than use this environment variable.
CNY_PERS_VERBOSE	When non-zero, displays information about personality instruction files selected during compilation. Default value is 0.
CNY_PRECISE_TRAPS	When non-zero, the location where a coprocessor exception occurred (those that aren't ignored) will be reported more precisely (precise trap mode). Default is 0. Precise trap mode is primarily intended for use with the debugger. The trap model for the process can be changed at process startup via this environment variable. . At runtime the cny_coproc_precise() routine and the debugger can change the trap model. In precise trap mode, the coprocessor runs slower than in imprecise trap mode. The simulator always runs in precise trap mode.

Environment Variable	Description (default value)
CNY_PROG_MODEL	Controls whether an application requests shared or exclusive access to the Convey coprocessor. This environment variable may be set to "PROCESS_SHARED" or "PROCESS_EXCLUSIVE ". The default value is **PROCESS_EXCLUSIVE**. See chapter 8.3 **Process Groups** for a discussion of shared access to the coprocessor. This environment variable has no effect when using the Convey simulator.
CNY_RUNTIME_STARTUP_DEBUG	When non-zero, larger values produce increasing levels of debug output about startup of the application and coprocessor. Default value is 0.
CNY_SE_MASK	Scalar coprocessor exceptions may be masked by setting this environment variable to one or more of the following exception names: SUIE SFIE SFDE SFZE SFOE SFUE SIZE SIOE SRRE SURE SSOE SCOE SDIE The default value is "SFUE,SSOE,SIOE,SFDE" (floating point underflows produce a value of 0.0, integer overflows are ignored, store overflows are ignored, and Denorms are ignored). The various exception names are documented in the **Convey Reference Manual**.
CNY_SIMCALL_DBG	When non-zero, larger values produce increasing levels of debug output about dispatches (only on the simulator). Default value is 0. • 1 - minimal tracing • 2 - call parameters • 3 - more verbose
CNY_SIMCALL_MSGTRACE	When non-zero, enables the collection and dumping of any messages produced by coprocessor code at the end of execution. See chapter 9 **User Callable Support Routines** (Message Routines) for a description of the coprocessor messaging routines. Default value is 0.
CNY_SIM_THREAD	Name of the shared library containing the simulator - usually "libcpSimLib2.so". When this environment variable is set, the specified shared library is used as the coprocessor simulator. When not set, the real coprocessor is used, if available.

Environment Variable	Description (default value)
CNY_SIM_WEAK_ORDER_CHECK	When set to a non-zero value, the simulator will check for potentially missing fences that may be needed due to the weak memory ordering characteristics of the Convey coprocessor. Default is 0.
CNY_STACK_SIZE	Amount of memory to be allocated for the coprocessor runtime stack, in bytes. Default value is 4MB.
CNY_WAIT_STRATEGY	Specifies how the host processor should wait for a dispatch to complete. Available strategies are **SPIN**, and **POLL**. Default is **SPIN**. • **SPIN** causes the host processor to continually check for dispatch completion • **POLL** sleeps CNY_WAIT_STRATEGY_TIME micro-seconds between checks for dispatch complete, leaving the host free to do other work
CNY_WAIT_STRATEGY_TIME	How many microseconds to wait between checks for dispatch completion, if the dispatch wait strategy is POLL. The default value is 100µsec.

13 Debugging Coprocessor Code with gdb

Convey's enhanced gdb debugger is based on the GNU Project Debugger and can be used for debugging C, C++, Fortran, and assembly code that runs on the host processor, as well as coprocessor assembly code that runs on the coprocessor or the coprocessor simulator. Gdb contains an extensive built-in help system (type `help` or `info cny_help` at a command prompt). Many online resources are available.

- **http://www.gnu.org/software/gdb/documentation/** contains a very detailed description of gdb and all its commands and a User Manual on usage.

Convey's enhanced gdb allows debugging of code running on the Convey coprocessor hardware or the Convey coprocessor simulator. The remainder of this chapter deals with Convey specific commands and features, and assumes a general understanding of gdb.

13.1 Selecting Convey Coprocessor Architecture or Host Architecture

Convey gdb interfaces to the coprocessor via a thread that has knowledge of the architecture of the coprocessor, all other threads have the architecture of the host machine. The coprocessor thread is visible by using the `info thread` command and is normally thread three. Convey gdb will switch between the host and coprocessor architectures as needed. The user may switch between architectures by using the `thread N` command to switch to the coprocessor thread or any host thread. When the architecture is set to the coprocessor, only coprocessor registers and commands are accessible, when set to the host, only host commands and registers are accessible. There are two exceptions to this rule, the `breakpoint` command and the `disassemble` command inspect the address of the command (using the executable file) to determine the correct architecture to call to execute the command.

13.2 Debugging user written assembly language

When debugging user written assembly language with gdb, the assembly language routines should be assembled with the **-g** to allow line number breakpoints. When using a graphical interface such as the Data Display Debugger (DDD) on top of gdb to debug an application, the **-g** is not necessary.

When debugging user written assembly language, the following features of the Convey assembler should be considered:

- The **Convey Reference Manual** defines pseudo-instructions to simplify the writing of assembly code. For example

```
mov    %s3, %s2
```

  is a pseudo-instruction that is mapped to the equivalent coprocessor instruction

```
or     %s3,$0,%s2 .
```

  Gdb will display the actual coprocessor instruction instead of the pseudo-instruction.

Gdb commands that display the assembly language source file (i.e. **`step`**) will show what the user actually wrote (**`mov %s3,%s2`**), while commands that use gdb's disassembler (i.e. **`x/10i $ip`**) will show the actual instructions (**`or %s3,$0,%s2`**). Pseudo-instructions are marked in the Convey Reference Manual with '#'.

- Some instructions are 'bundled' together into an instruction packet. An instruction packet is executed as a single entity, therefore even when single stepping, both instructions will be executed at the same time.

  Similarly, some instructions are reordered in order to make bundling possible. Thus gdb may display instructions in a different order than they were written. Only adjacent independent instructions (instructions that do not depend on each other) are bundled.

  The assembler directive **`.bundle off`**, the assembler flag **`–bundling_off`**, and setting the environment variable **`CNY_GAS_BUNDLE`** to **`off`** may be used to turn off bundling. It is easier to single step through user written assembly language (with the **`step`** or **`stepi`** commands) when instruction bundling is turned off. Instruction bundling should only be turned off when debugging an application, due to its negative performance impact.

13.3 Convey coprocessor gdb enhancements

Convey gdb contains commands that are specific to the coprocessor,

- **`info cny_help`** – Display available register names and coprocessor commands that are common to all personalities. If a personality is loaded, register names and commands that are specific to the personality loaded will be displayed. These commands are available only if the coprocessor thread has been selected. Caution, commands and registers that are associated with the host processor are not available when the coprocessor thread is selected.
- **`set cny_verbose`** – Set or clear the verbose flag for **`cny`** commands.
- **`set cny_nop`** – Set or clear the **`nop`** display flag. If this flag is set, all **`nop`**s will be displayed in the output from a disassemble command.

and commands that are specific to the personality currently loaded.

- Single precision float
  - **`info cny_svm [vm_register_number [first_bit [last_bit]]]`** – display the single precision vector mask (for vpm = 0) lsb to msb
  - **`info cny_svr [vector_number [first_element [last_element]]]`** – display the single precision vector elements (for vpm = 0)
  - **`info cny_svmp [partition [vm_register_number [first_bit [last_bit]]]]`** – display the single precision vector mask (for vmp = 1,2) lsb to msb
  - **`info cny_svrp [partition [vector_number [first_element [last_element]]]]`** – display the single precision vector elements (for vmp = 1,2)

- Double precision float (the financial analytics personality)
  - `info cny_dvm [vm_register_number [first_bit [last_bit]]]` -- display the double precision vector mask (vpm = 0) lsb to msb
  - `info cny_dvr [vector_number [first_element [last_element]]]` -- display the double precision vector (vpm = 0)
  - `info cny_dvmp [partition [vm_register_number [first_bit [last_bit]]]]` -- display the double precision vector mask (vpm = 1,2) lsb to msb
  - `info cny_dvrp [partition [vector_number [first_element [last_element]]]]` -- display the double precision vector (vpm = 1,2)

- Custom
  - `info cny_aeg ae_unit [first_element [last_element]]` – display the aeg registers for the specified ae unit
  - `info cny_aeg_desc` -- show the loaded aeg register descriptions
  - `cny_load_aeg_desc file` – load the specified aeg register description file. See the PDK Reference Guide for information on describing the aeg registers to gdb.

The Convey coprocessor registers may be used with the usual gdb commands, such as `print` or `set`. The user can use the `info cny_help` command to display the available registers and the syntax used to access them. The Convey coprocessor contains some registers that are composed of several bit fields; these registers are displayed and modified as bit fields.

```
(gdb) print $wb
    $1 = {awb = 4, swb = 4, vwb = 4, vrrb = 0, vrrs = 0, vrro = 0, vmwb =
          4, vma = 0}
(gdb) set $wb.vwb = 17
(gdb) p $wb
    $2 = {awb = 4, swb = 4, vwb = 17, vrrb = 0, vrrs = 0, vrro = 0,
          vmwb = 4, vma = 0}
(gdb) p $wb.vwb
    $3 = 17
```

The a and s registers are a union of an unsigned long long (u64), an array of two single precision floats (f) and a double precision float.

```
(gdb) p $s4
$5 = {u64 = 0, f = {0, 0}, d = 0}
(gdb) set $s4.f[1] = 9.1
(gdb) p $s4
$7 = {u64 = 4688697569478443008, f = {0, 9.09999943}, d = 288358.25}
(gdb)
```

The Convey coprocessor **a** and **s** registers are accessed programmatically through the offsets contained in the window base register (**awb** and **swb**) allowing only 64 registers to be visible to the program at any one time. Gdb allows access the same way by using the **`$a`** and **`$s`** register names. When a **`$a`** or **`$s`** register is accessed, the correct **`wb`** register is selected (depending on the current frame) and the offset in the **`wb`** is added to the specified register (**`a0->a63`** and **`s0->s63`** are valid) to determine the absolute **`a`** or **`s`** register to access. Gdb also allows direct access to all coprocessor a and s registers by using the **`$aa`** and **`$sa`** register names (**`aa0->aa255`** and **`sa0->sa255`** are valid). This access bypasses the **`wb`** completely and accesses the registers directly.

13.4 Convey Single Precision Personality Registers

The Convey coprocessor single precision personality contains registers that are specific to the personality. These register names are shown by the `info cny_help` command and are accessed via the usual gdb `print` or `set` commands.

The vector mask register name is dependent on the current vector partition mode:

- **`vm[0->15]b[0->15]`** – (vpm=0) **`$vm3b5`** will access vector mask register 3 bitfield 5
- **`vm[0->15]b[0->3]p[0->3]`** – (vpm=1) **`$vm9b2p3`** will access vector mask register 9 bitfield 2 from partition 3
- **`vm[0->15]b[0]p[0->31]`** – (vpm=2) **`$vm1b0p13`** will access vector mask register 1 bitfield 0 from partition 13

The vm register is VL bits in length. The first 64 bits are displayed in bitfield 0 (lsb = vector element 0 mask), the second (if VL > 64) in bitfield 1 etc.

The command `info cny_svm` or `info cny_svmp` will show only the **`vm`** bits currently valid as a string of bits with the leftmost bit displayed being the mask bit for vector element 0.

The vector register name is dependent on the current vector partition mode:

- **`v[0->63]e[0->1023]`** – (vpm=0) **`$v3b5`** will access vector register 3 element 5
- **`v[0->63]e[0->255]p[0->3]`** – (vpm=1) **`$v9e2p3`** will access vector register 9 element 2 from partition 3
- **`v[0->63]e[0->31]p[0->31]`** – (vpm=2) **`$v1e0p13`** will access vector register 1 element 0 from partition 13

The commands `info cny_svr` and `info cny_svrp` will show all elements of the specified vector (or a subset specified by the command).

The register names **`$v`** and **`$vm`** and the commands `info cny_svm[p]` and `info cny_svr[p]` are all offset by the value in the **`vmwb`** and **`vwb`** portions of the current **`wb`** register.

Each vector register element is a union of an unsigned long long (u64) and an array of two single precision floats (f). The user may `print` the value of a vector register element in a specific format as shown:

```
(gdb) print $v2e44
$6 = {u64 = 4, f = {5.60519386e-45, 0}}
(gdb) print $v2e44.u64
$7 = 4
(gdb) print $v2e44.f[0]
$8 = 5.60519386e-45
(gdb) print $v2e44.f[1]
$9 = 0
```

```
(gdb) set $v2e44.u64 = 9
(gdb) print $v2e44
$10 = {u64 = 9, f = {1.26116862e-44, 0}}
(gdb)
```

While using single precision instructions which use one single precision float value (ld.fs, st.fs, add.fs etc) there are 64 directly accessible vectors, v0l->v31l, v0r->v31r. The user must use the vwb register to access v32l->v63l and v32r->v63r. To display v0l element 0 using gdb the user must use $v0e0.f[1] and to display v0r element 0 use $v0e0.f[0] (the left element is in f[0] and the right element is in f[1]).

While using single precision instructions which use two single precision float values (ld.cs, st.cs, add.cs etc) there are 64 directly accessible vectors v0->v63. When using gdb to display these vectors the value in f[0] is the real part and f[1] is the imaginary part.

Note: A vector should only be accessed with the vector length and vector partition mode set to the same value as the program that created them. If these values are not the same, invalid results will be displayed.

13.5 Convey Financial Analytics Personality Registers

The double precision personality register access is the same as the single precision register access with the following changes:

- The `info cny_svm` command is `info cny_dvm`
- The `info cny_svmp` command is `info cny_dvmp`
- The `info cny_svr` command is `info cny_dvr`
- The `info cny_svrp` command is `info cny_dvrp`
- The vector element is a union of an unsigned long long (u64) and a double precision float (d).

13.6 Convey Custom Personality Registers

The Convey coprocessor custom personality contains registers that are specific to the personality. These register names are shown by the `help cny_info` command and are accessed via the usual gdb `print` command. The register name **`aeg[0->3]e[0->Aeg_Cnt-1]`** accesses the **`aeg`** register contained in the **`ae(0->3)`** element **`(0->Aeg_Cnt-1)`**.

The command `info cny_aeg ae_unit [first_element [last_element]]` will display the **`aeg`** registers for the specified **`ae`** unit. It will display all **`aeg`** registers (default) or a subset of registers specified by the user.

13.7 Cautions When Using Multiple Architectures

There are some pitfalls to be aware of when using multiple architectures in the same gdb debugging session:

- If a Convey register is accessed while the architecture is set to the host, a convenience variable of the same name will be generated, example: **`set $v2e4 = 9`** will create a debugger variable named **`v2e4`** if the architecture is not the Convey coprocessor.
- You should not set the architecture with the command **`set architecture`**
- If you step/stepi/next while executing within a coprocessor region and the coprocessor finishes executing the current coprocessor region (by executing an **`rtn`** instruction), you will see:

  ```
  Cannot step into a coprocessor dispatch complete rtn
  You can only continue or quit from here
  ```

  and the coprocessor will not execute the command.
- If you set a breakpoint on a routine name, or a line number within a particular routine, gdb will attempt to set a breakpoint for both the named routine, and, if it exists, the **`cny_`** version of that routine.
  i.e. **`b myroutine:3`**
  will set a breakpoint at line 3 in **`myroutine`**, and also at line 3 in **`cny_myroutine`** (if possible)

14 Personalities, Signatures, and Signature Resolution

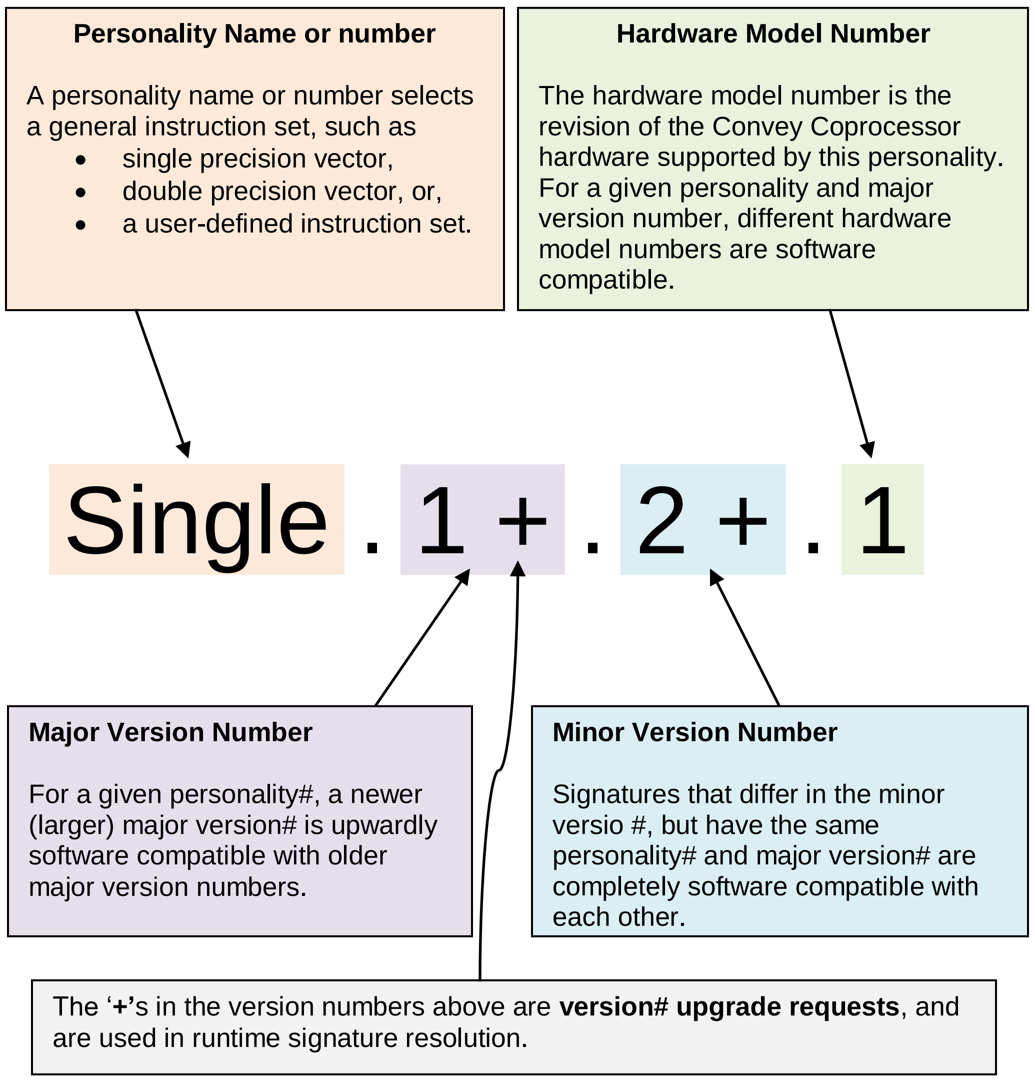
A signature identifies a particular personality for the Convey coprocessor, as well as version information for that personality.

When a routine is compiled, the user specified signature controls what coprocessor personality (and corresponding instruction set) the compiler will generate code for.

When a program executes a coprocessor routine, that same signature (after runtime signature resolution) also controls which particular firmware image to load into the coprocessor.

Signatures are composed of four fields, separated by periods. Most user provided signatures will only need to specify one or two of these fields.

What's in a signature?



Personality Name or number

A personality name or number selects a general instruction set, such as

- single precision vector,
- double precision vector, or,
- a user-defined instruction set.

Hardware Model Number

The hardware model number is the revision of the Convey Coprocessor hardware supported by this personality. For a given personality and major version number, different hardware model numbers are software compatible.

Major Version Number

For a given personality#, a newer (larger) major version# is upwardly software compatible with older major version numbers.

Minor Version Number

Signatures that differ in the minor versio #, but have the same personality# and major version# are completely software compatible with each other.

The '**+**'s in the version numbers above are **version# upgrade requests**, and are used in runtime signature resolution.

14.1 Signature Definition

Field Name	Commonly used signature fields
personality number and names	The personality number, and its associated names, identifies a general instruction set. Examples of a general instruction set include *single precision vector*, *double precision vector*, or a particular *user-defined instruction set*. A personality name can be used anywhere a personality number is allowed.
major version number	Two signatures that are identical except for the major version number are upwardly software compatible. Code compiled for a lower valued major version number can execute using a higher valued major version number, if all other signature fields are identical or compatible. New major versions of a personality typically indicate new instructions have been added to that personality, but all the old instructions are still supported.
minor version number	Two signatures that are identical except for the minor version number are completely software compatible with each other. Code compiled for such signatures can execute using either signature. Newer minor versions of a personality typically indicate a bug fix or performance improvement in the associated coprocessor image.
Field Name	**Less commonly used signature fields**
hardware model number	The hardware model number indicates which model Convey coprocessor the compiler should generate code for. For a given personality number and major version number, different hardware model numbers are software compatible. An executable program compiled for a particular hardware model number can be run on a coprocessor with a different hardware model number, without being re-compiled. Compiling with the correct hardware model number will likely improve performance. Most users will never specify a value for the hardware model number field. At compile time, the system's default hardware model number will be used. Runtime signature resolution forces all coprocessor routines to execute using the hardware model number from the system's *default base signature*, ensuring the proper coprocessor image is loaded into the coprocessor before the routine is actually executed.

14.2 Partial Signatures, Zero-valued Version Numbers, and Signature Upgrades

User specified signatures can be partial signatures, or fully resolved signatures. A fully resolved signature has non-zero values for the personality number, the major version number, the minor version number, and the hardware model number. A fully resolved signature must also be installed and enabled to be a valid signature.

14.2.1 Zero-valued Version Numbers

In a user specified signature, absent version numbers are always considered to be zero. In the discussions below, any action taken for a *zero-valued* version number value applies to absent version numbers as well. A signature with zero-valued version numbers is automatically upgraded to a compatible installed and enabled signature.

14.2.2 Version Number Upgrade Attributes

In a signature, the version number upgrade attribute allows the user to specify particular major and/or minor version numbers to use when compiling a routine, but allows those version numbers be possibly upgraded at runtime to a newer version. An upgrade attribute is a '+' immediately after the major or minor version number in a signature, for example, single.2+

A zero-valued version number automatically has the version number upgrade attribute, even if no '+' was specified.

This capability allows code to be compiled for the simplest version of a personality (such as major version number 1 of a particular personality), but run with the newest compatible signature available when the program is actually run. Sites with several hybrid-core servers or applications that are distributed to other customer sites can ensure maximum compatibility by compiling for a commonly installed personality version, but still benefit from future performance and bug fix releases without recompiling an application.

When the major version number has an upgrade attribute, that signature may not specify a non-zero minor version number, and the upgrade attribute is automatically applied to the minor version number.

Upgrade attributes can appear in the signature specified as a compiler flag, in the user specified list of default signatures, or in the system default signature list. During the compile time signature resolution process, once an upgrade attribute is applied to the major or minor version number, that version number keeps the upgrade attribute throughout the signature resolution process.

14.3 Signature Resolution

A partial signature supplied to the Convey compilers/assembler undergoes compile time signature resolution to select a compatible fully resolved signature for use by the compiler/assembler.

At runtime, all signatures undergo runtime signature resolution to update the hardware model number, and, if requested by the user, to possibly update the major/minor versions numbers.

14.3.1 Compile Time Signature Resolution

When a routine is compiled, the user typically specifies a partial signature. This partial signature undergoes compile time signature resolution, resulting in a particular signature that determines what coprocessor instruction set to use. The steps of compile time signature resolution are:

- A personality name, if any, is replaced with its corresponding personality number.
- A zero-valued personality number is replaced with the default personality number from the system's default base signature. If the system's default personality number is zero, the user must always specify a valid personality name or number.
- Zero-valued hardware model numbers are replaced with the default hardware model number from the system's default base signature.
- At the end of this step, the resulting signature, along with any *upgrade attributes*, is called the associated runtime signature. If this associated runtime signature is a partial signature, or any upgrade attributes are present, runtime signature

resolution will attempt to minimize the number of different signatures needed to execute the program.

- The user specified list of default signatures is scanned, looking for compatible signatures that will fill in any zero-valued major and minor version numbers.
- The system default signature list is scanned, also looking for compatible signatures that will fill in any remaining zero-valued major and minor version numbers.
- If the resulting signature is not fully resolved, the personality database is scanned to find the best compatible signature. When scanning the *personality database*, only entries for installed and enabled signatures are considered. The personality database only contains fully resolved signatures. Entries in the *personality database* are usually in reverse chronological order. Newer signatures (higher major and minor version numbers) appear before older signatures.

14.3.1.1 Compatible Signatures

When scanning the *user specified list of default signatures*, the *system default signature list*, and the *personality database*, an entry from those lists is compatible with a partial signature if

- the personality numbers match,
- the partial signature's major version number either matches the entry's major version number, or the partial signature requested a major version number upgrade and the entry's major version number is greater than or equal to the major version number in the partial signature, and
- the partial signature's minor version number either matches the entry's minor version number, or the partial signature requested a minor version number upgrade and the entry's minor version number is greater than or equal to the minor version number in the partial signature. An entry which is a fully resolved signature, but one that is disabled or not installed is never considered compatible.

When a compatible entry is found, the major and minor version numbers from the compatible entry replace the corresponding values in the partial signature. If the resulting signature is still a partial signature, scanning continues.

Note that the final signature produced by compile time signature resolution must be fully resolved, installed, and enabled; otherwise, an error occurs.

In addition, the double precision personality is compatible with the financial analytics personality, but the financial analytics personality is not compatible with the double precision personality. Coprocessor code compiled for the double precision personality can run on the coprocessor as long as either the financial analytics or double precision personality is available.

14.3.2 Runtime Signature Resolution Overview

Runtime signature resolution attempts to minimize the total number of different signatures (and corresponding firmware images) needed to execute a program. Since loading a new coprocessor firmware image is a relatively slow process, and each different signature has a different associated image, reducing the number of signatures a program executes with is very important.

There are two recommended methods to minimize the number of different firmware images required:

- Compile everything with the same fully resolved signature, or
- Compile all (or most) coprocessor code with the same partial signature, and let runtime signature resolution pick one compatible image to use.

The second method should be used when a program is linked with 3rd party precompiled code (i.e. an ISV supplied library), unless the 3rd party provider recommends a specific fully resolved signature.

The second method allows the newest version of a personality to be used each time a program is executed. If you want to guarantee that the program will always use the same signature and image, use a fully resolved signature to compile all coprocessor code.

At runtime, when a coprocessor routine is about to be executed:

- The associated runtime signature has its hardware model number replaced with the hardware model number from the system's *default base signature* (it identifies what model coprocessor is installed).
- If the associated runtime signature is compatible with the current image loaded on the coprocessor, that current image will be used, and runtime signature resolution is complete.
- If the associated runtime signature's major version number has the upgrade attribute:
  - That major version number is upgraded by scanning the user specified list of default signatures, the system default signature list, and the personality database (in that order), until an entry with the same personality number and an equal or larger major version number is found.
  - The major version number from that matching entry replaces the major version number in the associated runtime signature, and no further upgrading of the major version number occurs.
  - If the matching entry has a non-zero minor version number, that entry's minor version number replaces the minor version number in the associated runtime signature and runtime signature resolution is complete.
- If the associated runtime signature's minor version number has the upgrade attribute:
  - That minor version number is upgraded by scanning the user specified list of default signatures, the system default signature list, and the personality database (in that order), until an entry with the same personality, the same major version number, and a non-zero minor version number is found.
  - The minor version number from that matching entry replaces the minor version number in the associated runtime signature, and runtime signature resolution is complete.

The final associated runtime signature must be fully resolved and be an installed and enabled signature, or it is invalid.

The final associated runtime signature is used solely to select which coprocessor image to load before executing the coprocessor routine or code segment.

14.3.3 Controlling Signature Resolution

As described above, the system's default base signature, the user specified list of default signatures, the system default signature list, and the personality database all participate in signature resolution.

The ***system's default base signature*** provides:

- A system wide default personality number used whenever the user failed to specify a personality name/number, or specified a zero value for the personality number.
- A hardware model number that matches the installed coprocessor's model number. If no coprocessor is installed, the system administrator can provide a default value for the hardware model number. This hardware model number is always used when a coprocessor routine is about to be called, to ensure the proper firmware image for the system's coprocessor is loaded. This hardware model number is also used at compile time if the user did not specify a hardware model number.

The system's default base signature never participates in major or minor version number resolution.

The **Convey System Administration Guide** describes how to set a system's default base signature.

The *user specified default signature list* is contained in the environment variable **CNY_DEFAULT_PERSONALITIES**. If present, this environment variable should contain a list of signatures, separated by commas. Partial signatures are allowed, including nicknames and upgrade attributes. For example, a bash shell user could initialize **CNY_DEFAULT_PERSONALITIES** by a command such as

```
export CNY_DEFAULT_PERSONALITIES="2.1.0,1.2.1+"
```

Signatures in the *user specified default signature list* must have non-zero major version numbers, or they will have no effect.

The *system default signature list* contains system administrator defined signatures that affect signature resolution. If the user specifies a partial signature, and scanning the *user specified default signature list* doesn't result in a fully resolved signature, the *system default signature list* provides a way for the system administrator to guide signature resolution, usually by specifying a particular *approved* major version number for each personality.

Signatures in the *system default signature list* must have non-zero major version numbers, or they will have no effect.

See the Convey System Administration Guide for information on setting up the *system default signature list*.

The Convey supplied ***personality database*** is provided by Convey, and lists all available signatures (installed or not). The system administrator can enable or disable any installed signature. Disabled signatures are not considered compatible when scanning the *personality database*.

Both the user and system administrator have some control over signature resolution.

The system administrator can control signature resolution by:

- Disabling signatures with the **`/opt/convey/sbin/cpm disable`** … command. A disabled signature cannot be used to compile or execute code.
- The *system default signature list* allows the system administrator to *suggest* particular major and minor version numbers for a particular personality. If the user specifies a partial signature, and there is no compatible entry in the *user specified default signature list* that specifies a non-zero version number for major and minor version numbers, then the *system default signature list* can force a particular major or minor version number to be used (not necessarily the newest).

The user can control signature resolution by:

- Setting an entry in the *user specified default signature list* that specifies particular major and minor version numbers for a personality. When the user specifies a partial signature on the compilers command line, the signature undergoes compile time signature resolution, and the *user specified default signature list* is scanned before the *system default signature list*. When an entry in the *user specified default signature list* matches a partial signature, the non-zero values for major and minor version numbers from that matching entry are used.
- The user can specify a signature with non-zero major and minor version numbers when compiling a routine. Compile time signature resolution will not affect a fully resolved signature.

14.3.3.1 Controlling Signature Resolution Examples

If the system administrator wants to force users to only use major version number 1 of the double precision personality, but always get the latest minor version thereof, the system administrator should:

- Mark all other major versions of that personality disabled, using the **`/opt/convey/sbin/cpm disable`** … command, and
- add the entry double.1.1+ to the system's default signature list

If the system administrator wants to *suggest* that users should use major version 2 of the single precision personality:

- The system administrator should put the signature **`single.2`** or **`single.2+`** in the system's default signature list
- The user can override this suggestion by:
  - Specifying non-zero version numbers when compiling a routine, or
  - Adding an entry to the *user specified default signature list* that selects a particular major version number for the single precision personality (**`single.3`** or **`single.3+`** for example)

If the user wants to just specify the personality name **`sp`** when compiling a routine, but always wants to request version **`1.2`**, adding the entry **`sp.1.2`** to the *user specified* *default signature list* will map `sp` to `sp.1.2` unless `sp.1.2` has been disabled by the system administrator.

14.4 Personality Tools

The script `/opt/convey/sbin/cpm` (the Convey Personality Manager) allows the system administrator to enable and disable particular personalities and lets any user list all installed personalities. See the `man` page for `cpm` for more information.

14.5 Canonical Instruction Set Version Number

In addition to the personality and signature versioning capabilities, there is also a ***canonical instruction set version number***. When this version of the Convey Programmers Guide was published, the only valid *canonical instruction set version number* was zero. For now, there is no way to set or specify the *canonical instruction set version number*.

If Convey ever introduces an incompatible change in the canonical instruction set, Convey will provide tools to specify the *canonical instruction set version number*, and to set a system default value.

14.6 Object Code Version Number

There are hidden object code version numbers in all `.o` and executable files produced by the Convey compilers and assembler. When this version of the Convey Programmers Guide was published, the only valid object code version number was zero.

The object code version number is used to ensure the coprocessor object code format in all .o and executable files matches the format the actual coprocessor or coprocessor simulator expects.

Convey does not currently plan to make an incompatible change in the object code format for coprocessor code. If such a change is ever made, Convey will provide documentation, tools and compiler flags to manage the use of object code version numbers.

15 Customer Support Procedures

Please use email for questions, and the issue tracker web interface to report bugs and request enhancements to Convey products.

Email	**support**
Issue Tracker Web Interface	Browse to **http://rt.conveysupport.com**
Phone	Our toll-free phone # is **1-866-338-1768** -- pick "Customer Support" when prompted, and please leave a brief message and a call back # if you reach the automated voicemail system

Appendix A How To Take Advantage Of The Convey Coprocessor

The various characteristics of the Convey coprocessor described in chapter 4.1 **Porting Existing Applications** must be carefully considered when porting an application to utilize the Convey coprocessor. Although many different parts of a typical application can be ported to run on the coprocessor, running all possible portions of an application on the coprocessor is unlikely to improve its performance.

The following approaches to utilizing the Convey coprocessor are listed in order of ease of use. Not every approach is appropriate for a particular application.

Convey Mathematical Library

The **Convey Mathematical Libraries User's Guide** describes what routines are provided in Convey's math library. These routines utilize the coprocessor when it is available, and execute equivalent host code otherwise. The math library includes:

- Dense vector and matrix operations, including the Level 1, 2, and 3 BLAS
- Sparse vector operations, including the sparse BLAS
- Linear equation solution, including LAPACK
- Eigenvalue solution, including LAPACK
- Discrete Fourier Transforms, including 1-D, 2-D, 3-D and multiple 1-D

Using Convey directives and source code restructuring

For numeric applications, particularily those that operate on large arrays, vectorizing those loops that consume most of the application's runtime is the preferred method of utilizing the Convey coprocessor.

The `-mcny_vector` flag enables the generation of vector coprocessor code. The compiler will attempt to vectorize loops in coprocessor regions (source code enclosed in `begin_coproc/end_coproc` directives/pragmas) and in routines compiled with the `dual_target/dual_target_nowrap` compiler flags and directives/pragmas.

The `begin_coproc/end_coproc` directives/pragmas provide a simple way to request coprocessor code generation for a specifc loop. In general, the less scalar code in a coprocessor region, the faster that code will run on the coprocessor. Therefore, vectorizable loops with large iteration counts will provide the biggest performance boost. Restructuring code to avoid unvectorizable scalar loops in the coprocessor regions will improve performance of a coprocessor region.

Coprocessor regions (source code enclosed in `begin_coproc/end_coproc` directives/pragmas) will only execute on the coprocessor if it is attached. If the coprocessor is not attached, equivalent host code will be executed instead.

For loops that contain procedure calls, but are otherwise good candidates for running on the coprocessor, Convey provides the `dual_target` compiler flag and directive/pragma. These allow the called procedures to be compiled for both the host processor, and the coprocessor. The coprocessor version of the routine has its name changed to start with `cny_`. This allows that routine to be called from both host and coprocessor code in the same application. This is helpful when some calls to the routine are made from host code, while some are made from a coprocessor region.

The `dual_target` compiler flags and directives/pragmas also cause the host version of a routine to check to see if the coprocessor is attached, and if so, the host version of the routine will dispatch the coprocessor version of the routine, instead of executing the host version. If the coprocessor is not attached, the host version will be executed.

A hand coded assembly language coprocessor routine may be called from a coprocessor region or routine, provided it follows the `cny_` naming convention, and an equivalent host version is also provided. See chapter 11 **Coprocessor Assembly Language Calling Conventions** for a description of calling conventions for coprocessor code.

Routines compiled with the `dual_target` compiler flags and directives/pragmas must conform to the restrictions for coprocessor code, including no I/O statements, no unsupported language library calls, etc.

In addition to generating coprocessor code, proper memory allocation and migration (between the host and coprocessor) is critical to maximizing application performance. The `coproc_mem` and `host_mem` directives/pragmas are used to allocate static variables, at link time, in host or coprocessor memory. The `migrate_host` and `migrate_coproc` directives/pragmas are used to migrate variables at runtime to host or coprocessor memory.

Although link time allocation of variables to host or coprocessor memory is essentially free, runtime migration of variables is expensive, unless those variables are not currently paged into memory. Since migration is expensive, executing small loops on the coprocessor, even those with large iteration counts, may not improve performance, since the memory migration costs can offset any performance boost from the coprocessor.

Accessing large amounts of host memory from coprocessor code also incurs a relatively large performance penalty, and should be avoided. Referencing a few scalar variables occasionally is ok, but arrays whose elements are referenced more than once or twice in a loop should be initially allocated in or migrated to coprocessor memory before executing the coprocessor region or routine, and migrated back to host memory if necessary.

Automatic Vectorization

Convey's C, C++, and Fortran compilers provide the `-mcny_auto_vector` flag that enables automatic vectorization of `DO` and `for` loops. This capability is, for now, primarily intended to help identify what loops can be easily vectorized, so the developer can select those that will benefit the most from execution on the coprocessor.

The automatic vectorization capability, although functional, is not a recommended approach to getting the most performance from the coprocessor. Future releases of the compilers and firmware upgrades are expected to improve the capabilities of automatic vectorization and add support for automatic memory migration. For now, Convey recommends using the Convey directives/pragmas for controlling coprocessor code generation and memory migration.

Developing a custom personality

For some applications, especially those that aren't good candidates for vectorization, developing a custom personality may offer the best possible performance boost. Developing a custom personality is far more difficult than vectorizing an application, but a custom personality allows the coprocessor to be utilized in the most effective manner, and can result in much higher performance gains than otherwise possible.

See the **PDK Reference Manual** for more information on developing a custom personality.

Appendix B Optimizing Performance When The Coprocessor Is Unavailable

This chapter describes how to port an application that will run on a Convey server (with a coprocessor) but is expected to sometimes run when the coprocessor is in use by another process. In such a case, care must be taken to allocate memory appropriately, to minimize how much memory is allocated on the coprocessor when only the host processor will be accessing that memory.

The following guidelines should allow an application to execute efficiently on the host processor when the coprocessor is unavailable:

- If there is sufficient host memory available, all memory locations that are referenced more than once or twice should be allocated in or migrated to host memory.
- The application should migrate memory to the node ID of the current node, rather than explicitly migrating memory to the host or coprocessor node. This ensures that routines compiled using the **`dual_target`** capability will migrate memory appropriately when the coprocessor is not available.

Appendix C Less Commonly Used Compiler Flags

This appendix describes those Convey compiler flags that are not commonly used. See 4.8 User-Defined Intrinsics

A function call may be treated as a vectorizable intrinsic. The user is responsible for writing the intrinsic in coprocessor assembly language, and specifying the intrinsic with the compilers `-mcny_vec_rtn` flag.

The user defined vector intrinsic must return a vector result in the v0 register. In the call to the vector intrinsic, all arguments to the original function are converted to vectors passed in as v0, v1, ..., vN.

All arguments to the original function must be scalar values.

Example compilation steps:

```
cnycc -c user_def.s

cnycc -mcny_sig=sp -mcny_vec_warn -mcny_vector \

               -mcny_vec_rtn:dsign=__vecdsign \

               -mcny_vec_rtn:rsign=__vecrsign \

       call_dsign1.c   user_def.o
```

Example File:

```
#include <stdlib.h>
#include <stdio.h>

    int b[10];
    float c[10];

#pragma cny dual_target_nowrap(1)
float dsign(int i, float j) { return i*j+1;}
int rsign(float i, int j) { return i*j+2;}

#pragma cny dual_target_nowrap(0)
int main() {
   int i,j;
   int n=10;

#pragma cny begin_coproc
  for (j=0;j<4;j++) {
   for (i=0; i<n; i++) {
     b[i] = dsign(i,1.0f + j) + rsign(1.1f+i,j);
   }
  }
#pragma cny end_coproc
   for (i=0; i<n; i++) {
```

```
            printf("%d\n",b[i]);
        }
        return 0;
    }
```

Additional messages can be produced by the compiler when the

-mcny`_vec_warn option` is enabled. Acceptance of the compiler option can be verified by the presence of these messages:

```
Info: user defined vector function: dsign -> __vecdsign
Info: user defined vector function: rsign -> __vecrsign
Info: user defined vector function: dsign -> __vecdsign
Info: user defined vector function: rsign -> __vecrsign
```

The replacement of the user function call by the vector intrinsic can be verified by these messages:

```
"call_dsign1.c", line 20, Info: Replaced cny_dsign with vector
function __vecdsign
"call_dsign1.c", line 20, Info: Replaced cny_rsign with vector
function __vecrsignThe sample assembly file:
```

The intrinsic implementation in coprocessor assembly language follows:

```
$ cat user_def.s
        .section .ctext, "ax",@progbits
        .section .ctext
        .signature sp
        .weak   __vecdsign
        .type   __vecdsign, @function
__vecdsign:
        cvt.sq.fs   %v0,%v0
        mul.fs   %v0,%v1,%v0
        cvt.fs.sq   %v0,%v0
        add.sq   %v0,$1,%v0
        cvt.sq.fs   %v0,%v0
        rtn

        .weak   __vecrsign
        .type   __vecrsign, @function
__vecrsign:
        cvt.sq.fs   %v1,%v1
        mul.fs   %v0,%v1,%v0
        cvt.fs.sq   %v0,%v0
        add.uq   %v0,$2,%v0
        rtn
```

Note the `.signature` assembler directive is specific to the personality the intrinsic is designed for.

Compiler flags for the list of more commonly used compiler flags.

The compiler flags are listed in alphabetical order, except that the "`no`" form of a flag appears with its non-`no` version (i.e. `-noinline` appears immediately after `-inline`).

Flags listed here that are not listed elsewhere in this document are either internal compiler flags, or are considered less useful or dangerous. These additional flags are documented in case the customer support group, or an FAQ on the Convey Support website, recommends their use.

Color coding is used, for a flags textual description, to indicate which languages support a particular compiler flag:

- Black: C, C++, and Fortran shared flags
- Purple: Fortran only flag
- Green: C/C++ only flags
- `Dark red flags (Convey specific flags) are supported by all languages`

For users reading a monochrome copy of this document, the **Lang** column also indicates when a particular flag is only available for some languages. When the **Lang** column is empty, that flag is supported in C, C++, and Fortran.

Less Commonly Used Compiler Flags		Lang
`-CG:cny_dp_div=[0,1]`	When 1 (and compiling coprocessor code for the financial analytics personality) divide operations (real*8 / double) will use the double precision divide code sequence instead of the financial analytics divide code sequence. Default is 1. See the note (‡) at the end of this table.	
`-CG:cny_dp_recip=[0,1]`	When 1 (and compiling coprocessor code for the financial analytics personality) 1.0/x operations (real*8 / double) will use the double precision recipricol code sequence instead of the financial analytics recipricol code sequence. Default is 1. See the note (‡) at the end of this table.	
`-CG:cny_dp_sin=[0,1]`	When 1 (and compiling coprocessor code for the financial analytics personality), calls to the `sin` intrinsic will call the double precision `sin` instead of the financial analytics `sin`. Default is 0.	
`-CG:cny_dp_sqrt=[0,1]`	When 1 (and compiling coprocessor code for the financial analytics personality), calls to the `sqrt` intrinsic will use the double precision `sqrt` code sequence instead of the financial analytics `sqrt` code sequence. Default is 1. See the note (‡) at the end of this table.	

`-CG:cny_drop_exp_wait=[0,1]`
`-CG:cny_drop_log_wait=[0,1]`
`-CG:cny_drop_sqrt_wait=[0,1]`
`-CG:cny_drop_recip_wait=[0,1]`
	If 1, suppresses generation of "wait vector mask active" instructions for the indicated intrinsic (financial analytics personality only). This does not affect correctness, but can improve performance.	

`-clist`	Turn on source code listing (C/C++ compilers only). Recreates C/C++ source code from compilers internal intermediate code, after IPA inlining and loop-nest transformations. This is a compiler debugging option, and the generated source code may not be compilable.	C,C++
`-flist`	Turn on source code listing (Fortran compiler only). Recreates Fortran source code from compilers internal intermediate code, after IPA inlining and loop-nest transformations. This is a compiler debugging option, and the generated source code may not be compilable.	**Ftn**
`-mcny_callable_names=`*`fn`*	Allow all the routine names listed in the file '*fn*' to be called from coprocessor regions.	
`-mcny_dt_ignore_calls=`*`fn`*	Ignore all functions listed in *fn* when compiling a coprocessor region (coprocessor code generation will not be inhibited if one of the specified functions is called within a region)	
`-mcny_exclude_names=`*`fn`*	Exclude all the routine names listed in the file '*fn*' from being called from coprocessor regions. If a coprocessor region calls an excluded routine name, the compiler will only generate host code for that region.	
`-mcny_lib_select`	Enable user provided profitabiltity analysis. See **User Provided Dual** Target Profitability for a description of this feature.	

`-mcny_sig=all`
 	
`-mcny_sig=none`
	No particular signature is associated with the generated object file, and coprocessor instructions from all personalities are valid.
	
	No particular signature is associated with the generated object file, and no coprocessor instructions are valid.	

`-OPT:cny_nz_loop_trip=1`	Disables zero-trip DO loop test. All DO loops will execute at least one iteration. This was common practice in Fortran 66 compilers, and can improve short loop performance as long as there are no zero-trip DO loops.	
`-skipcny_routine=`*name*	Suppress coprocessor code generation for routine '*name*'.	
`-skipconveylink`	Link without `cny_initruntime.o -ldl` and `-llibcny_utils.a`	
`-skipconveyrdynamic`	Suppresses the `-Wl,--export-dynamic` flag usually passed to the linker.	
`-TARG:cny_auto_migrate=[0,1]`	If 1, automatically migrate arrays in loops vectorized by `-mcny_auto_vector`. Default is 0 (don't auto-migrate).	
`-TARG:cny_dynamic_select=[0,1]`	If 1, automatically insert dynamic selection tests for loops vectorized by `-mcny_auto_vector`. Default is 0 (don't insert dynamic selection tests automatically).	
`-TARG:cny_comp_ops=[0,1]`	If 1, enable component operation. Default is 0.	
`-TARG:cny_rotate=[0,32]`	Enable register rotation. Default is 1.	
`-TARG:cny_vshift=[0,32]`	Enable vector shift. Default is 0.	
`-TARG:cny_vls_opt=[0,1]`	If 1, enable vector length/stride opts. Default is 1.	
`-TARG:cny_vector_match=[0,1]`	If 1, enable vector match. Default is 0.	
`-TARG:cny_rra_move=[0,1]`	If 1, enable moves rather than rotates. Default is 0.	
`-TARG:cny_reduce=[0,1]`	If 1, compiler will generate reductions in coprocessor code. Default is 1.	
`-TARG:cny_redcnt=n`	Tells the compiler to generate HW reductions specialized to a user-specified reduction count. Default is 4.	
`-TARG:cny_loop_distrib=[0,1]`	If 1, enable loop distribution and all other dependence breaking transformations. Default is 1.	

-TARG:cny_loop_inter=[0,1]	If 1, enable loop interchange for performance and isolating dependences. Default is 1.	
-TARG:cny_loop_unroll=[0,1, n]	If 1, enable loop unrolling and use the value specified in an unroll directive / pragma for the unroll count. If 0, disable loop unrolling. If greater than 1, use the specified value as the loop unroll count for loops with a negative valued unroll directive / pragma. Default is 1.	
-TARG:cny_loop_unroll_pretail=[0,1]	If 1, perform vector loop unrolling using a VLmax pre-tail. Default is 1.	
-TARG:cny_array_rename=[0,1]	If 1, enable array renaming to break dependences. Default is 1.	
-TARG:cny_scalar_rename=[0,1]	If 1, enable scalar renaming to break dependences. Default is 1.	
-TARG:cny_scalar_expand=[0,1]	If 1, enable scalar expansion to break dependences. Default is 1.	
-TARG:cny_reduce_vpreg=[0,1]	If 1, enable vector pseudo-registers for unrolling. Default is 1. See the loop unrolling example for more details.	
-TARG:cny_expand_vpreg=[0,1]	If 1, allows the use of vector registers for scalar expanded references in a loop. Default is 1.	
-TARG:cny_node_split=[0,1]	If 1, enable node splitting to break dependences. Default is 1.	

‡ NOTE: The indicated operations (divide, sqrt, and recipricol) are, by default, implemented using the code sequences for the double precision personality, even when compiling for the financial analytics personality. If the financial analytics versions of these operations are enabled, and the input values are not sufficiently randomly distributed, a substantial performance penalty will be incurred, due to memory bank conflicts when doing table lookups.

Appendix D Less Commonly Used Environment Variables

`CNY_CPMALLOC_GRANULARITY`	Specifies the minimum number of bytes to request from the OS when requesting coprocessor memory. Must be a multiple of the page size (4096, …). Default value is the page size on the node memory is allocated from. The value can be a hex or decimal number.
`CNY_CPMALLOC_MMAP_THRESHOLD`	Specifies the request size threshold for using **`mmap`** to directly service a request. Requests of at least this size that cannot be allocated using already-existing space will be serviced via **`mmap`**. Using **`mmap`** segregates relatively large chunks of memory so that they can be individually obtained and released from the host system. Default is 1GB. The value can be a hex or decimal number.
`CNY_CPMALLOC_TOUCH_PAGES`	A zero value indicates that page touching of newly allocated malloc space will NOT be performed. A non-zero value indicates that ALL pages added to the malloc pool will be touched prior to returning space to the caller. Default value is non-zero.
`CNY_CPMALLOC_TRIM_THRESHOLD`	Specifies the maximum amount of unused coprocessor memory to keep before releasing back to the system. Default is 1GB. The value can be a hex or decimal number.
`CNY_DT_IGNORE_CALLS=`*`fn`*	Ignore all functions listed in *fn* when compiling a coprocessor region (coprocessor code generation will not be inhibited if one of the specified functions is called within a region)
`CNY_PTABLE`	Specifies an alternate pdf.dat (personality data file) to use for instruction validation. Default value is pdf.dat in the appropriate personality directory.
`CNY_PTABLE_OVERRIDE`	If set to "**`on`**", the compiler assumes all instructions are valid. Default value is **`off`**.
`CNY_TOUCH_MIGRATED_PAGES`	When set to a non-zero value, all pages in a migrated memory region (following a successful memory migration using `cny_migrate_data()`) will be touched. The default value is 1.

Appendix E Coprocessor Intrinsics

The Convey compilers support calls of the following intrinsic procedures (and uses of the equivalent operator) in coprocessor code. Calls to these intrinsic procedures or uses of these operators, in both scalar code and vectorized loops, will not prevent a coprocessor region from being created.

Some of these operations are inlined by the compiler, while the more complicated operations implemented with coprocessor library routines. Which intrinsic are inlined varies depending on the personality selected and compiler flags specified.

Note that the list is sorted alphabetically by the description column, not the C/C++ or Fortran intrinsic name. A "☑ " symbol indicates a language intrinsic for which the compiler issues one or more coprocessor instructions or calls a coprocessor intrinsic library routine. A "⃠" symbol indicates an intrinsic which isn't available in that particular language.

Description	**Intrinsic Name**		**Argument Datatypes Supported**			
	Fortran Name (generic)	**C/C++ Names**	**REAL** (4 byte) **float**	**REAL** (8 byte) **double**	**INTEGER** (4 byte) **int**	**INTEGER** (8 byte) **long, long long**
absolute value	abs	fabs, fabsf, fabsl, abs, labs, llabs	☑	☑	☑	☑
arc cosine	acos	acos, acosf	☑	☑		
arc sine	asin	asin, asinf	☑	☑		
arc tangent	atan	atan, atanf	☑	☑		
arc tangent (2 arg form)	atan2	atan2, atan2f	☑	☑		
ceiling	ceiling	ceil, ceilf	☑	☑		
exponential	(e**y)	exp, expf	☑	☑		
floor	floor	floor, floorf	☑	☑		
hyperbolic cosine	cosh	cosh, coshf	☑	☑		
hyperbolic sine	sinh	sinh, sinhf	☑	☑		
hyperbolic	tanh	tanh, tanhf	☑	☑		

tangent						
logarithm	log	log, logf	☑	☑		
logarithm base 10	log10	log10, log10f	☑	☑		
mod	mod	fmod, fmodf	☑	☑		
modulo	modulo		☑	☑		
positive difference	dim	fdim, fdimf	☑	☑		
power	(x**y)	pow, powf	☑	☑		
round to nearest	anint	⃠	☑	☑		
round to nearest (IEEE)	⃠	rint, rintf	☑	☑		
round to zero	aint	trunc, truncf	☑	☑		
sign	sign	copysign, copysignf	☑	☑		
sine	sin	sin, sinf	☑	☑		
square root	sqrt	sqrt, sqrtf	☑	☑		
tangent	tan	tan, tanf	☑	☑		

Appendix F User Provided Dual Target Profitability

By default, the host version of a routine compiled with `-mcny_dual_target` will always dispatch the coprocessor version of that routine if the coprocessor is attached.

If some calls of the routine would perform better executing on the host, while others would perform better executing on the coprocessor, the routine may be compiled with the `-mcny_lib_select` flag, and the host code will execute a user supplied profitability test routine, and only dispatch the coprocessor version of the routine if the profitability test routines returns non-zero.

The profitability test routine must be named `xxx$selector`, where `xxx` was the routine name of interest. It must be a function that returns an `INTEGER*4` / `int` value, and it will be called with the same arguments/types as the routine of interest was called with. The profitability test routine does not need to be in the same source file as the routine of interest. The profitability test routine may be coded in C or Fortran.

If the profitability test routine isn't present in the executable program, the coprocessor version of the routine will be dispatched (if the coprocessor is attached to the current process). If the profitability test routine is present, and returns a non-zero value, the coprocessor version of the routine will be dispatched; otherwise, the host version of the routine wil be executed.

Example: the coprocessor version of `libtest` is only dispatched if `N` is greater than 60.

Compiled with:

```
$ cnyf90 -mcny_dual_target -mcny_lib_select sourcefile.f90

integer function libtest$selector(N,arr1,arr2,arr3)
  integer N,arr1,arr2,arr3
  if (N.gt.60) then
    libtest$selector = 1
  else
    libtest$selector = 0
  endif
end

integer*8    function  libtest(N, arr1,arr2,arr3)
  integer i,arr1(n), arr2(n), arr3(n)
  do i =1,n
    arr1(i) = N-i
    arr2(i) = i
  enddo

  do i=1,n
    arr3(i) = arr1(i) + arr2(i)
  enddo
  libtest = 0
  do i=1,n
    libtest = libtest + arr3(i)
  enddo
end
```

```
    program mainprog
      parameter (n=50)
      integer  arr1(N), arr2(N), arr3(N)
! next directive prevents compiler from trying to generate coprocessor
! code for this main program (due to -mcny_dual_target flag) which would
! give warnings due to I/O statements
!$cny dual_target(0)
      integer*8 libtest,sum
      sum = libtest(n,arr1,arr2,arr3)
      print*,  arr3
      print *, "sum:", sum
      print *, "exp:", (n-1)*n/2 + n*(n+1)/2

  end
```

Appendix G Low Level Interfaces to Coprocessor Routines

The `copcall` interface routines provide a low level mechanism for dispatching coprocessor routines. These routines provide explicit control over what registers are used to pass arguments, and may be particularily useful for PDK users. In general, Convey recommends using the `begin_coproc/end_coproc` directives and pragmas described earlier.

The interfaces described in this chapter are callable from C, C++, or Fortran.

C language interface routines

The first three arguments to every interface routine callable from C are the same:

`sig`	The signature for the coprocessor routine being invoked. This value should be a value returned from the `cny$get_signature_fptr` routine in the **`sig1`** argument. **`sig1`** should be declared as **a** `cny_image_t` type.
`funcptr`	The address of the coprocessor routines entry point. It should have the type of the value returned by the coprocessor routine (pointer to function returning that type).
`argdesc`	A character string that describes the type of each argument to be passed into the coprocessor routine. All arguments after `argdesc` are passed into the interface routine. `argdesc` should contain one character, in argument order, for each argument to be passed into the coprocessor routine:

`'a'`	pass a 32 bit quantity in an A register
`'A'`	pass a 64 bit quantity in an A register
`'s'`	pass a 32 bit float quantity in an S register
`'S'`	pass a 64 bit double quantity in an S register
`'l'`	pass a 32 bit quantity in an S register (lower-case L)
`'L'`	pass a 64 bit quantity in an S register

The first value loaded into an A register is loaded into A8, the second into A9, …

Similarly, the first value loaded into an S register is loaded into S1, the second into S2, …

The interface routines are declared in a Convey provided include file. Add this include line to your C source code:

```
#include <convey/usr/cny_comp.h>
```

The routines are declared as:

```
extern float  f_copcall_fmt(cny_image_t,float(*func)(void),char*, ...);
extern double d_copcall_fmt(cny_image_t,double(*func)(void),char*,...);
extern int    i_copcall_fmt(cny_image_t,int   (*func)(void),char*, ...);
extern long   l_copcall_fmt(cny_image_t,long  (*func)(void),char*, ...);
extern void     copcall_fmt(cny_image_t,void  (*func)(void),char*, ...);
extern void*  v_copcall_fmt(cny_image_t,void*(*func)(void),char*, ...);
```

For example:

```
i_copcall_fmt(mysig, cpTest1, "AL", 0x12LL, 0x34LL);
```

passes two long longs into the `cpTest1` coprocessor routine: 0x12LL goes into A register **`a8`** and **`0x34LL`** goes into S register **`s1`**, both as 64 bit quantities. `mysig` is a personality signature.

Fortran language interface routines

The interface routines callable from Fortran all have a trailing underscore as part of their name. By default, the Fortran compiler appends an underscore to every external name, so the trailing underscore should not be included in the routine name unless one of the compiler flags that suppress trailing underscores is used. The same interface names with two trailing underscores are aliased to the corresponding names with one trailing underscore, in case `-fsecond-underscore` is used.

The first argument to each of these interface routines should be an `INTEGER*8` signature value. Typically, the `cny_f_get_signature` routine is used to convert a personality name or other partial signature into a signature value.

The second argument should be the address of the coprocessor routine to invoke.

The third argument should be the number of arguments to pass to the coprocessor routine.

Any remaining arguments in the call to the interface routine will be passed via an A register, starting with A8.

Each of the interface routines that return a value should be explicitly declared with that type, and every interface routine should be declared EXTERNAL with an **`EXTERNAL`** statement or declaration attribute.

Return Type	**Function Name**	
`real*4`	`f_copcall_f_`	
`real*8`	`d_copcall_f_`	
`integer*4`	`i_copcall_f_`	
`integer*8`	`l_copcall_f_`	
`none`	`copcall_f_`	
`Address (64 bits)`	`v_copcall_f_`	(declare as returning an **`integer*8`**)

These routines may also be called from C/C++, as long as Fortran's calling conventions are followed (everything is pass by address).

Simple Coprocessor Interface Routine Example

The simplest form of the varargs interface is shown below. It does not pass in any arguments, nor return any values, but does let the user specify the name of the coprocessor routine to call. The interface routine name is `copcall_fmt`, and is called from C as follows:

```
#include <stdio.h>
#include <convey/usr/cny_comp.h>

extern void myasm();
cny_image_t sig1, sig2;

…
  if (cny$get_signature_fptr)
    /* get a signature for the personality nickname "sp" */
    (*cny$get_signature_fptr) ("sp", &sig1, &sig2);
  else
    fprintf(stderr,"where is get_signature_fptr?\n"), exit(1);

if (cny$coprocessor_ok)
  copcall_fmt(sig1, &myasm, "");
else
    host_version_of_myasm();
…
```

The name `myasm` should be replaced with the name of the coprocessor routine (in the `.globl` directive).

The indirect call through `cny$get_signature` returns a signature (in `sig1`) that the simulator and runtime libraries can use. This example gets a single precision signature. The simulator has built-in support for the single and double precision personalities. All other personality numbers will cause the simulator to assume a customer defined personality is being used, and appropriate simulator extensions and remote debugging are required (see the **ConveyPDK Reference Manual** for more information about custom defined personalities).

Although no arguments are passed into `myasm`, and no value is returned, the `myasm` coprocessor routine can load from and store to external globally visible memory, such as `COMMON` blocks and `extern` variables. The personality name or number supplied to `cny$get_signature_fptr` should be

- the personality number or a nickname of the single precision or double precision vector personality, or
- a user-defined personality number (or nickname), preferably in the range reserved for site local personalities (32000-32999), or in the range reserved and assigned by Convey for ISV provided personalities (33000+).

For a user-defined personality, if any of the reserved user-defined opcodes are present in the assembly language routine, the necessary simulator support, compiled with that personality number, must be provided by the user. See the **Convey PDK Reference Manual** for more information.

Sample coprocessor routine call in C

This sample C routine calls the assembly language routine `fib_` that appeared earlier in this document. It can be compiled and linked (assuming `fib.o` was created previously) with the command

```
  $ cnycc  -g  mainfib.c  fib.o
```

```
/* mainfib.c */
#include <stdio.h>
#include <convey/usr/cny_comp.h>

long long result_[32];

int main() {
  extern void fib_();
  int i;
  cny_image_t sig1, sig2;

  if (cny$get_signature_fptr)
    (*cny$get_signature_fptr) ("sp", &sig1, &sig2);
  else
    fprintf(stderr,"where is get_signature_fptr?\n"), exit(1);


  if (cny$coprocessor_ok)
    copcall_fmt(sig1, &fib_, "");
  else
      host_version_of_fib();


  for (i=0; i<32; ++i)
    printf(" %d  %lld\n", i, result_[i]);
}
```

Sample coprocessor routine call in Fortran

Compile with:

```
    $ cnyf95 -fno-second-underscore -g mainfib.f90 fib.o
```

where fib.o was produced by compiling the coprocessor assembly routine in **`fib.s`**.

```
program fibonacci
  integer*8 results(32)
  external fib  ! fortran adds an underscore to external names
  common /result/ results
  integer*8 sig1, sig2
  integer stat
  integer cny_f_coprocessor_ok

  call cny_f_get_signature("sp", sig1, sig2, stat)
  write(*,'(z " " z  " " z)'), sig1, sig2, stat

  write(6,*) "sig=", mydef_sig
  if (cny_f_coprocessor_ok() .eq.1 ) then
     call copcall_f_(sig1, fib, 0)
   else
     call fib()
  endif
  do i=1,32
    write(6,*)results(i)
  end do
end
```

Appendix H Process Group Tutorial

Thie following tutorial is adapted from a paper by Andries Brouwer (http://www.win.tue.nl/~aeb/linux/lk/lk-10.html#ss10.2).

Every process is member of a *process group*, uniquely identified by its *process group ID*. When a process is created, it becomes a member of the process group of its parent. The process group ID of a process group is the process ID of the first member of the process group. This first member of a process group is called the *process group leader*. A process may find its process group ID using the system call **getpgrp()**, or, equivalently, **getpgid(0)**.

On a Convey Server, the shell command `ps j` shows the PPID (parent process ID), PID (process ID), PGID (process group ID) and SID (session ID) of processes. Typically, the processes of one pipeline, such as

```
% cat paper | ideal | pic | tbl | eqn | ditroff > out
```

form a single process group.

Creation

A process `pid` is put into the process group `pgid` by

```
setpgid(pid, pgid);
```

If **pgid == pid** or **pgid == 0** then this creates a new process group with process group leader **pid**. Otherwise, **pid** is put into the existing process group **pgid**. A zero value for **pid** refers to the current process. **setpgrp()** is equivalent to **setpgid(0,0)**.

Restrictions on setpgid()

The calling process' process id must be **pid**, or **pid**'s parent, and the parent can only do this before **pid** has done an **exec()**, and only when both belong to the same session. It is an error if process **pid** is a session leader (since **setpgid** would change its **pgid**).

Typical sequence

```
p = fork();
if (p == (pid_t) -1) {
        /* ERROR */
} else if (p == 0) {     /* CHILD */
        setpgid(0, pgid);
        ...
} else {                 /* PARENT */
        setpgid(p, pgid);
        ...
}
```

This ensures that regardless of whether the parent or child is scheduled first, the process group setting is as expected by both.

Signalling and waiting

One can signal all members of a process group:

```
killpg(pgrp, sig);
```

One can wait for children in ones own process group:

```
waitpid(0, &status, ...);
```

or in a specified process group:

```
waitpid(-pgrp, &status, ...);
```

Foreground process group

Among the process groups in a session at most one can be the *foreground process group* of that session. The tty input and tty signals (signals generated by ^C, ^Z, etc.) go to processes in this foreground process group.

A process can determine the foreground process group in its session using `tcgetpgrp(fd)`, where `fd` refers to its controlling `tty`. If there is none, this returns a random value larger than 1 that is not a process group ID.

A process can set the foreground process group in its session using `tcsetpgrp(fd,pgrp)`, where `fd` refers to its controlling `tty`, and `pgrp` is a process group in the its session, and this session still is associated to the controlling `tty` of the calling process.

How does one get `fd`? By definition, `/dev/tty` refers to the controlling `tty`, entirely independent of redirects of standard input and output. (There is also the function `ctermid()` to get the name of the controlling terminal. On a POSIX standard system it will return `/dev/tty`). Opening the name of the controlling `tty` gives a file descriptor `fd`.

Background process groups

All process groups in a session that are not foreground process group are *background process groups*. Since the user at the keyboard is interacting with foreground processes, background processes should stay away from it. When a background process reads from the terminal it gets a `SIGTTIN` signal. Normally, that will stop it, the job control shell notices and tells the user, who can use `fg` to continue this background process as a foreground process, and then this process can read from the terminal. But if the background process ignores or blocks the `SIGTTIN` signal, or if its process group is orphaned (see below), then the `read()` returns an `EIO` error, and no signal is sent. (Indeed, the idea is to tell the process that reading from the terminal is not allowed right now. If it wouldn't see the signal, then it will see the error return.)

When a background process writes to the terminal, it may get a `SIGTTOU` signal. May: namely, when the flag that this must happen is set (it is off by default). One can set the flag by

```
% stty tostop
```

clear it again by

```
% stty -tostop
```

and inspect it by

```
% stty -a
```

Again, if `TOSTOP` is set but the background process ignores or blocks the `SIGTTOU` signal, or if its process group is orphaned (see below), then the `write()` returns an `EIO` error, and no signal is sent.

Orphaned process groups

The process group leader is the first member of the process group. It may terminate before the others, and then the process group is without leader.

A process group is called *orphaned* when *the parent of every member is either in the process group or outside the session*. In particular, the process group of the session leader is always orphaned.

If termination of a process causes a process group to become orphaned, and some member is stopped, then all are sent first `SIGHUP` and then `SIGCONT`.

The idea is that perhaps the parent of the process group leader is a job control shell (in the same session but a different process group). As long as this parent is alive, it can handle the stopping and starting of members in the process group. When it dies, there may be nobody to continue stopped processes. Therefore, these stopped processes are sent **SIGHUP**, so that they die unless they catch or ignore it, and then `SIGCONT` to continue them.

Note that the process group of the session leader is already orphaned, so no signals are sent when the session leader dies.

Note that a process group may become orphaned by termination of a process that was:

- a parent and not itself in the process group, or
- the last element of the process group with a parent outside but in the same session.

A process group can also become orphaned when some member is moved to a different process group.

Sessions

Every process group is in a unique *session*. (When the process is created, it becomes a member of the session of its parent.) By convention, the session ID of a session equals the process ID of the first member of the session, called the *session leader*. A process finds the ID of its session using the system call `getsid()`.

Every session may have a *controlling tty*, which is the controlling tty of each of its member processes. A file descriptor for the controlling tty is obtained by opening `/dev/tty`. If the open of `/dev/tty` fails, there was no controlling `tty`. Given a file descriptor (`fd`) for the controlling `tty`, one may obtain the `SID` using `tcgetsid(fd)`.

A session is often set up by a login process. The terminal on which one is logged in then becomes the controlling tty of the session. All processes that are descendants of the login process will in general be members of the session.

Creation

A new session is created by

```
pid = setsid();
```

This is allowed only when the current process is not a process group leader. In order to be sure of that, you can fork first:

```
p = fork();
if (p) exit(0);
pid = setsid();
```

The result is that the current process (with process ID `pid`) becomes session leader of a new session with session ID `pid`. Moreover, it becomes process group leader of a new process group. Both session and process group contain only the single process pid. Furthermore, this process has no controlling tty.

The restriction that the current process must not be a process group leader is needed: otherwise its pid serves as pgid of some existing process group and cannot be used as the pgid of a new process group.

Getting a controlling tty

The TIOCSCTTY ioctl will give us a controlling tty, provided that (i) the current process is a session leader, and (ii) it does not yet have a controlling tty, and (iii) maybe the tty should not already control some other session; if it does it is an error if we aren't root, or we steal the tty if we are all-powerful.

Opening some terminal will give us a controlling tty, provided that (i) the current process is a session leader, and (ii) it does not yet have a controlling tty, and (iii) the tty does not already control some other session, and (iv) the open did not have the O_NOCTTY flag, and (v) the tty is not the foreground VT, and (vi) the tty is not the console, and (vii) maybe the tty should not be master or slave pty.

Getting rid of a controlling tty

If a process wants to continue as a daemon, it must detach itself from its controlling tty. Above we saw that setsid() will remove the controlling tty. Also the ioctl TIOCNOTTY does this. Moreover, in order not to get a controlling tty again as soon as it opens a tty, the process has to fork once more, to assure that it is not a session leader. Typical code fragment:

```
 if ((fork()) != 0)
         exit(0);
setsid();
if ((fork()) != 0)
         exit(0);
```

Also see the man page for daemon(3).

Disconnect

If the terminal is the slave side of a pseudotty, and the master side is closed (for the last time), then a SIGHUP is sent to the foreground process group of the slave side.

When the session leader dies, a SIGHUP is sent to all processes in the foreground process group. Moreover, the terminal stops being the controlling terminal of this session (so that it can become the controlling terminal of another session).

Thus, if the terminal goes away and the session leader is a job control shell, then it can handle things for its descendants, e.g. by sending them again a SIGHUP. If on the other hand the session leader is an innocent process that does not catch SIGHUP, it will die, and all foreground processes get a SIGHUP.

Appendix I Fence Checking with the Convey Simulator and Controlling Compiler Generated Fences

Introduction

The Convey SDK includes a simulator capable of simulating the execution of Convey coprocessor code on any 64-bit Intel/AMD host CPU running an appropriate 64-bit Linux distribution. The simulator provides a mechanism for checking the correct placement of memory fence instructions in Convey coprocessor code. The sections below describe memory fence instructions, and the use of the fence-checking functionality in the Convey simulator.

Overview of the FENCE Instruction

The Convey Coprocessor supports a weakly ordered memory model. All coprocessor memory reads and writes are allowed to proceed independently to memory through different interconnect paths to provide increased memory bandwidth. Requests that follow different interconnect paths may arrive at the destination memory controller in a different order than they were issued. The weak memory ordering applies to separate load/store instructions, as well as between different vector elements accessed by a single instruction.

In order to provide applications a mechanism to enforce proper memory ordering constraints, the Convey Coprocessor architecture includes a fence instruction. The fence instruction performs a memory fence operation. A fence operation ensures that all memory operations issued before the fence are completed before any memory operations that appear after the fence.

The following examples illustrate various scenarios where memory ordering on the Convey Coprocessor is an important consideration:

Example #1 (store through a vector of indices):

```
ST.UD        V5, V3(A4)    ; Scatter (unsigned doubleword)
```

In example #1, the elements of V5 are stored to memory using the element values of vector register V3 as the offset from the base address in A4 for each stored element. The order that the individual stores will be written to memory is not defined. If multiple elements of vector register V3 have identical values then multiple elements of vector register V5 will be written to the same location of memory. The final value of memory for locations where multiple elements are written is undefined. Note that a fence instruction cannot solve an ordering issue between elements of the same instruction. Other techniques can be used in this type of operation to ensure correct operation.

Example #2 (store followed by load with overlapping addresses):

```
ST     V5,0(A5)      ; Store vector to memory
LD     0(A4),V6      ; Load vector from memory
```

In example #2, a store instruction is followed by a load instruction. Even though the store occurs before the load, multiple memory interconnect paths may reorder the read and write prior to arriving at the memory controllers. If the vectors based at addresses A4 and A5 overlap, some of the load instruction elements may be read prior to the previous store instruction's write to memory.

Example #3 (example #2 with fence instruction):

```
ST     V5,0(A5)      ; Store vector to memory
FENCE                ; Fence instruction
LD     0(A4),V6      ; Load vector from memory
```

Example #3 adds a fence instruction to example #2 to ensure proper memory ordering if the vectors based at addresses A4 and A5 overlap. Note that the fence instruction prohibits the load instruction from beginning execution until the store instruction's write operations have been committed to the memory controller. The fence instruction ensures that all stores have passed through the memory interconnect prior to allowing the load instruction to access memory.

Example #4 (load followed by a store with overlapping addresses):

```
LD     0(A4),V6      ; Load vector from memory
FENCE                ; Fence instruction
ST     V5,0(A5)      ; Store vector to memory
```

Example #4 illustrates the possible ordering problem of a load followed by a store. If a fence instruction is not included then the load and store memory operations could be reordered in the memory interconnect prior to arriving at the memory controller. When the fence instruction is present the load and store instructions will operate as expected even when the two instructions reference the same memory locations.

Example #5: (example #4 with identical base addresses)

```
LD     0(A4),V6      ; Load vector from memory
ST     V5,0(A4)      ; Store vector to memory
```

Example #5 illustrates the case of a load followed by a store where a fence is not required. Since the base address for the load and store instructions are identical, it can safely be assumed that references to the same memory addresses follow the same interconnect path to memory. As a result, a fence is not required. Note that this is not true if the store were executed before the load since the Convey Coprocessor gives precedence to loads over stores.

The following table shows when a fence instruction is required to ensure the correct ordering of memory operations:

		2nd Request			
		Scalar Load	Scalar Store	AE Load	AE Store
	Scalar Load	-	-	-	Fence
1st Request	Scalar Store	-	-	Fence	Fence
	AE Load	-	Fence	-	Fence[1]
	AE Store	Fence	Fence	Fence	Fence[1]

Note 1: Two AE references to the same memory address that follow the same interconnect path to memory may omit the intervening fence instruction.

The Convey `no_fence` Directive

When generating code for the vector personalities of the Convey Coprocessor, the Convey compilers automatically generate fence instructions without any additional effort required on the part of the user. The compiler will only automatically insert fences where analysis shows they are needed. However, in certain situations, the compiler may insert fences in locations where they are not necessary. The user can override this behavior with the no_fence directive:

(C/C++)	**`#pragma cny no_fence`**
(Fortran)	**`!$CNY NO_FENCE`**

The `no_fence` directive disables the automatic generation of coprocessor fence operations within and around a vectorized loop. Use with extreme caution, since intermittent wrong answers may result if memory stores initiated by the coprocessor do not complete before the host processor loads that memory location. The compiler will only automatically insert fences where analysis shows they are needed.

The `no_fence` directive can take an optional parameter list, which provides finer granularity of control over fence suppression:

(C/C++)	**`#pragma cny no_fence(where)`**
(Fortran)	**`!$CNY NO_FENCE(where)`**

The where parameter may be one or more of the following keywords, separated by commas:

`all`	Suppresses all fences.
`around`	Suppresses fences before or after a loop
`around_loop`	
`before`	Suppresses fences before a loop
`before_loop`	

after **after_loop**	Suppresses fences after a loop
between **between_deps**	Suppresses fences between (potential) data dependencies

The `no_fence` directive disables the automatic generation of coprocessor fence operations within and around a vectorized loop. This should be used with extreme caution, since intermittent wrong answers may result if a fence was actually needed and memory stores initiated by the coprocessor do not complete before the host processor loads that memory location.

The Convey Simulator and Fence Checking

Running a given executable within the Convey simulator is straightforward; the user must simply set the CNY_SIM_THREAD environment variable prior to running the application:

(Bourne shell) `export CNY_SIM_THREAD=libcpSimLib2.so`

(C shell) `setenv CNY_SIM_THREAD libcpSimLib2.so`

Beware that executing an application in the Convey simulator is much slower than executing the same application on Convey coprocessor hardware, so smaller data sets should be used when working with the simulator. The simulator, however, provides a number of highly useful features that are not available when running applications on Convey hardware. One such feature is a mechanism for checking the correct placement of memory fence instructions in Convey coprocessor code. The sections below will describe memory fence instructions, and the use of the fence-checking functionality in the Convey simulator.

Setting the CNY_SIM_WEAK_ORDER_CHECK environment variable to a value of 1 prior to executing an application in the Convey simulator will enable the fence checking mechanism:

(Bourne shell) `export CNY_SIM_WEAK_ORDER_CHECK=1`

(C shell) `setenv CNY_SIM_WEAK_ORDER_CHECK 1`

Rather than permanently modifying the environment, it may be preferable to simply create a wrapper script to modify this variable during the execution of the application within the simulator. Here is an example of such a script:

```
#!/bin/bash
# simrun.sh – run applications in the Convey simulator

export CNY_SIM_THREAD=libcpSimLib2.so
export CNY_SIM_WEAK_ORDER_CHECK=1

$*
```

Save this script as simrun.sh, and be sure to set the executable bits using the **chmod** command:

```
chmod a+x simrun.sh
```

Now you can run any given application in the Convey simulator with fence checking enabled:

```
./simrun.sh  ./myApp.exe parm1 parm2 …
```

Example Using Fence Checking

Example: use of indirect indexing array with duplicate entries

Consider the following C program:

```
#include <stdio.h>
#include <math.h>

#define N 10000

static int idx[N];
static double x[N], y[N];
#pragma coproc_mem(idx, x, y)

int main(int argc, char *argv[])
{
  register int i;

  // intentionally create idx with some duplicate elements here this time
  for ( i = 0; i < N; i++ ) {
    if ( i%2 == 0 )
      idx[i] = i;
    else
      idx[i] = i + 1;
  }

  for ( i = 0; i < N; i++ ) {
    x[i] = i / N;
    y[i] = 0.0;
  }

#pragma cny begin_coproc
#pragma cny no_loop_dep(y)
  for ( i = 0; i < N; i++ ) {
    y[idx[i]] = sqrt(x[i]);
  }
#pragma cny end_coproc

  printf("y[1000]=%f\n", y[1000]);
  return 0;
}
```

This code is problematic as it uses an index array **idx** to indirectly reference elements of the vector **y**. Normally this would be fine, as long as the elements in **idx** do not repeat. The code has introduced a **no_loop_dep** directive to reassure the compiler that the use of the **idx** array to access elements of the vector is safe; when in reality; this is not the case, as the duplicate introduces a potential dependency inhibiting vectorization of the loop. The Convey simulator's weak order checking can detect and flag this incorrect code as follows:

```
Error (IP=0x0000000000800178): Missing Fence detected
    AE store with previous AE store (with different function
pipes)   Address: 0x0000000000c1de30


    List of accesses to same memory range since last fence:
      IP=0x0000000000800178 AE(0)  Store
Address=0x0000000000c1de30 Bytes=8
      IP=0x0000000000800178 AE(1)  Store
Address=0x0000000000c1de30 Bytes=8
```

(Many more instances of this output are omitted here for brevity.)

This output illustrates that the fence checker has detected the duplicate accesses through the invalid index array in the vectorized loop.

Index

aliasing models 29
assembler 88
associated runtime signature 108
attached coprocessor 10
begin_coproc 45
bundle 100
canonical instruction set version number 112
cbss 88
cdata 88
cdf 66
CNY_SIM_THREAD environment variable76
compile time signature resolution 107
coprocessor 9
coprocessor code 10
coprocessor code example 58
coprocessor dispatch 11
coprocessor region 11
coprocessor routine 11
cpm
  personality manager 112
ctext 88
Cumulative Normal Distribution Function 66
default hardware model number 108
default personality number 107
default signatures 108
dispatch 11
end_coproc 45
fap_mt_init_default_state 68
fap_mt_set_table_size 66
fap_mt_set_table_size_custom 66
fap_mt_use_default_state 68
fap_sobol_init 70
Financial Analytics Personality 65
gdb 99
hardware model number 106
host code 11
hybrid –core server 9
instruction packet 100
major version number 106
mersenne 66
Mersenne-Twister random number
  generator 66
minor version number 106
object code version #s 112
PATH environment variable 18
personality 9
personality database 108
personality number 106
pointer aliasing models 29
region 11
runtime signature resolution 109
signature 9, 105
signature resolution 10, 107
SOBOL Random Numbers 70
support 113
system default signature list 108
trademarks ii
user specified default signature list 110, 111,
  112
User-Defined Intrinsics 35
version number upgrade attribute 107