



NVIDIA[®]

HPC Compilers

USER'S GUIDE

TABLE OF CONTENTS

Preface.....	ix
Audience Description.....	ix
Compatibility and Conformance to Standards.....	ix
Organization.....	x
Hardware and Software Constraints.....	xi
Conventions.....	xi
Terms.....	xii
Related Publications.....	xiii
Chapter 1. Getting Started.....	1
1.1. Overview.....	1
1.2. Creating an Example.....	1
1.3. Invoking the Command-level NVIDIA HPC Compilers.....	2
1.3.1. Command-line Syntax.....	2
1.3.2. Command-line Options.....	3
1.4. Filename Conventions.....	3
1.4.1. Input Files.....	3
1.4.2. Output Files.....	5
1.5. Fortran, C++ and C Data Types.....	6
1.6. Platform-specific considerations.....	6
1.6.1. Using the NVIDIA HPC Compilers on Linux.....	7
1.7. Site-Specific Customization of the Compilers.....	7
1.7.1. Use siterc Files.....	7
1.7.2. Using User rc Files.....	7
1.8. Common Development Tasks.....	8
Chapter 2. Use Command-line Options.....	10
2.1. Command-line Option Overview.....	10
2.1.1. Command-line Options Syntax.....	10
2.1.2. Command-line Suboptions.....	11
2.1.3. Command-line Conflicting Options.....	11
2.2. Help with Command-line Options.....	11
2.3. Getting Started with Performance.....	12
2.3.1. Using -fast.....	12
2.3.2. Other Performance-Related Options.....	13
2.4. Frequently-used Options.....	13
Chapter 3. Multicore CPU Optimization.....	16
3.1. Overview of Optimization.....	16
3.1.1. Local Optimization.....	17
3.1.2. Global Optimization.....	17
3.1.3. Loop Optimization: Unrolling, Vectorization and Parallelization.....	17
3.1.4. Interprocedural Analysis (IPA) and Optimization.....	17

3.1.5. Function Inlining.....	18
3.2. Getting Started with Optimization.....	18
3.2.1. -help.....	19
3.2.2. -Minfo.....	19
3.2.3. -Mneginfo.....	20
3.2.4. -dryrun.....	20
3.2.5. -v.....	20
3.3. Local and Global Optimization.....	20
3.3.1. -Msafepr.....	20
3.3.2. -O.....	21
3.4. Loop Unrolling using -Munroll.....	23
3.5. Vectorization using -Mvect.....	24
3.5.1. Vectorization Sub-options.....	24
3.5.2. Vectorization Example Using SIMD Instructions.....	26
3.6. Interprocedural Analysis and Optimization using -Mipa.....	28
3.6.1. Building a Program Without IPA – Single Step.....	29
3.6.2. Building a Program Without IPA – Several Steps.....	29
3.6.3. Building a Program Without IPA Using Make.....	29
3.6.4. Building a Program with IPA.....	30
3.6.5. Building a Program with IPA – Single Step.....	30
3.6.6. Building a Program with IPA – Several Steps.....	31
3.6.7. Building a Program with IPA Using Make.....	32
3.6.8. Questions about IPA.....	32
Chapter 4. Using Function Inlining.....	34
4.1. Automatic function inlining in C++ and C.....	34
4.2. Invoking Procedure Inlining.....	35
4.3. Using an Inline Library.....	36
4.4. Creating an Inline Library.....	36
4.4.1. Working with Inline Libraries.....	37
4.4.2. Dependencies.....	38
4.4.3. Updating Inline Libraries – Makefiles.....	38
4.5. Error Detection during Inlining.....	38
4.6. Examples.....	39
4.7. Restrictions on Inlining.....	39
Chapter 5. Using OpenMP for Multicore CPUs.....	41
5.1. OpenMP Overview.....	41
5.1.1. OpenMP Shared-Memory Parallel Programming Model.....	42
5.1.2. Terminology.....	43
5.1.3. OpenMP Example.....	44
5.2. Task Overview.....	44
5.3. Fortran Parallelization Directives.....	45
5.4. C++ and C Parallelization Pragmas.....	46
5.5. Directive and Pragma Recognition.....	46

5.6. Runtime Library Routines.....	46
5.7. OpenMP Environment Variables.....	52
Chapter 6. Using an NVIDIA GPU.....	54
6.1. Overview.....	54
6.2. Terminology.....	55
6.3. Execution Model.....	57
6.3.1. Host Functions.....	57
6.3.2. Levels of Parallelism.....	57
6.4. Memory Model.....	58
6.4.1. Separate Host and Accelerator Memory Considerations.....	58
6.4.2. Accelerator Memory.....	58
6.4.3. Cache Management.....	58
6.4.4. CUDA Unified Memory.....	59
6.5. OpenACC Programming Model.....	61
6.5.1. Enable OpenACC Directives.....	61
6.5.2. OpenACC Support.....	61
6.5.3. OpenACC Extensions.....	62
6.6. Supported Processors and GPUs.....	62
6.7. CUDA Versions.....	62
6.8. Compute Capability.....	64
6.9. Compiling an OpenACC Program.....	65
6.9.1. -acc.....	65
6.9.2. -gpu.....	66
6.10. OpenACC for Multicore CPUs.....	67
6.11. Running an OpenACC Program.....	68
6.12. OpenACC Error Handling.....	68
6.13. Environment Variables.....	72
6.14. Profiling Accelerator Kernels.....	73
6.15. OpenACC Runtime Libraries.....	74
6.15.1. Runtime Library Definitions.....	74
6.15.2. Runtime Library Routines.....	75
6.16. Supported Intrinsics.....	76
6.16.1. Supported Fortran Intrinsics Summary Table.....	76
6.16.2. Supported C Intrinsics Summary Table.....	77
Chapter 7. PCAST.....	79
7.1. Overview.....	79
7.2. PCAST with a "Golden" File.....	80
7.3. PCAST with OpenACC.....	83
7.4. Limitations.....	88
7.5. Environment Variables.....	88
Chapter 8. Using MPI.....	90
8.1. Using Open MPI on Linux.....	90
8.2. Using MPI Compiler Wrappers.....	91

8.3. Testing and Benchmarking.....	91
Chapter 9. Creating and Using Libraries.....	92
9.1. Using builtin Math Functions in C++ and C.....	92
9.2. Using System Library Routines.....	93
9.3. Creating and Using Shared Object Files on Linux.....	93
9.3.1. Procedure to create a use a shared object file.....	93
9.3.2. ldd Command.....	94
9.4. Using LIB3F.....	95
9.5. LAPACK, BLAS and FFTs.....	95
9.6. Linking with ScaLAPACK.....	95
9.7. The C++ Standard Template Library.....	95
Chapter 10. Environment Variables.....	96
10.1. Setting Environment Variables.....	96
10.1.1. Setting Environment Variables on Linux.....	96
10.2. HPC Compiler Related Environment Variables.....	97
10.3. HPC Compilers Environment Variables.....	97
10.3.1. FORTRANOPT.....	98
10.3.2. GMON_OUT_PREFIX.....	98
10.3.3. LD_LIBRARY_PATH.....	98
10.3.4. MANPATH.....	99
10.3.5. NCPUS.....	99
10.3.6. NCPUS_MAX.....	99
10.3.7. NO_STOP_MESSAGE.....	99
10.3.8. PATH.....	99
10.3.9. NVCOMPILER_CONTINUE.....	100
10.3.10. NVCOMPILER_TERM.....	100
10.3.11. NVCOMPILER_TERM_DEBUG.....	101
10.3.12. PWD.....	102
10.3.13. STATIC_RANDOM_SEED.....	102
10.3.14. TMP.....	102
10.3.15. TMPDIR.....	102
10.4. Using Environment Modules on Linux.....	102
10.5. Stack Traceback and JIT Debugging.....	103
Chapter 11. Distributing Files – Deployment.....	104
11.1. Deploying Applications on Linux.....	104
11.1.1. Runtime Library Considerations.....	104
11.1.2. 64-bit Linux Considerations.....	105
11.1.3. Linux Redistributable Files.....	105
11.1.4. Restrictions on Linux Portability.....	105
11.1.5. Licensing for Redistributable (REDIST) Files.....	106
Chapter 12. Inter-language Calling.....	107
12.1. Overview of Calling Conventions.....	107
12.2. Inter-language Calling Considerations.....	107

12.3. Functions and Subroutines.....	108
12.4. Upper and Lower Case Conventions, Underscores.....	109
12.5. Compatible Data Types.....	109
12.5.1. Fortran Named Common Blocks.....	110
12.6. Argument Passing and Return Values.....	111
12.6.1. Passing by Value (%VAL).....	111
12.6.2. Character Return Values.....	111
12.6.3. Complex Return Values.....	112
12.7. Array Indices.....	112
12.8. Examples.....	113
12.8.1. Example – Fortran Calling C.....	113
12.8.2. Example – C Calling Fortran.....	114
12.8.3. Example – C++ Calling C.....	115
12.8.4. Example – C Calling C ++.....	115
12.8.5. Example – Fortran Calling C++.....	116
12.8.6. Example – C++ Calling Fortran.....	117
Chapter 13. Programming Considerations for 64-Bit Environments.....	119
13.1. Data Types in the 64-Bit Environment.....	119
13.1.1. C++ and C Data Types.....	119
13.1.2. Fortran Data Types.....	119
13.2. Large Static Data in Linux.....	120
13.3. Large Dynamically Allocated Data.....	120
13.4. 64-Bit Array Indexing.....	120
13.5. Compiler Options for 64-bit Programming.....	121
13.6. Practical Limitations of Large Array Programming.....	122
13.7. Medium Memory Model and Large Array in C.....	122
13.8. Medium Memory Model and Large Array in Fortran.....	123
13.9. Large Array and Small Memory Model in Fortran.....	124
Chapter 14. C++ and C Inline Assembly and Intrinsics.....	126
14.1. Inline Assembly.....	126
14.2. Extended Inline Assembly.....	127
14.2.1. Output Operands.....	128
14.2.2. Input Operands.....	129
14.2.3. Clobber List.....	131
14.2.4. Additional Constraints.....	132
14.2.5. Simple Constraints.....	132
14.2.6. Machine Constraints.....	133
14.2.7. Multiple Alternative Constraints.....	135
14.2.8. Constraint Modifiers.....	136
14.3. Operand Aliases.....	137
14.4. Assembly String Modifiers.....	137
14.5. Extended Asm Macros.....	139
14.6. Intrinsics.....	139

LIST OF TABLES

Table 1	NVIDIA HPC Compilers and Commands	xii
Table 2	Option Descriptions	5
Table 3	Examples of Using siterc and User rc Files	8
Table 4	Typical -fast Options	12
Table 5	Additional -fast Options	13
Table 6	Commonly Used Command-Line Options	14
Table 7	Typical -fast Options	18
Table 8	Additional -fast Options	19
Table 9	Example of Effect of Code Unrolling	23
Table 10	-Mvect Suboptions	25
Table 11	Frequently-used OpenMP Runtime Library Routines Summary	47
Table 12	52
Table 13	Pool Allocator Environment Variables	60
Table 14	Supported Environment Variables	72
Table 15	Accelerator Runtime Library Routines	75
Table 16	Supported Fortran Intrinsics	76
Table 17	Supported C Intrinsic Double Functions	77
Table 18	Supported C Intrinsic Float Functions	78
Table 19	Supported Types for Tolerance Measurements	80
Table 20	PCAST_COMPARE Options	88
Table 21	NVIDIA HPC Compilers Environment Variable Summary	97
Table 22	Supported NVCOMPILER_TERM Values	100
Table 23	Fortran and C/C++ Data Type Compatibility	109
Table 24	Fortran and C/C++ Representation of the COMPLEX Type	110

Table 25 64-bit Compiler Options	121
Table 26 Effects of Options on Memory and Array Sizes	121
Table 27 64-Bit Limitations	122
Table 28 Simple Constraints	132
Table 29 x86_64 Machine Constraints	134
Table 30 Multiple Alternative Constraints	135
Table 31 Constraint Modifier Characters	136
Table 32 Assembly String Modifier Characters	138
Table 33 Intrinsic Header File Organization	140

PREFACE

This guide is part of a set of manuals that describe how to use the NVIDIA HPC Fortran, C++ and C compilers. These compilers include the *NVFORTTRAN*, *NVC++* and *NVC* compilers. They work in conjunction with an assembler, linker, libraries and header files on your target system, and include a CUDA toolchain, libraries and header files for GPU computing. You can use the NVIDIA HPC compilers to develop, optimize and parallelize applications for NVIDIA GPUs and x86-64, OpenPOWER and Arm Server multicore CPUs.

The *NVIDIA HPC Compilers User's Guide* provides operating instructions for the NVIDIA HPC compilers command-level development environment. The *NVIDIA HPC Compilers Reference Manual* contains details concerning the NVIDIA compilers' interpretation of the Fortran, C++ and C language standards, implementation of language extensions, and command-level compilation. Users are expected to have previous experience with or knowledge of the Fortran, C++ and C programming languages. These guides do not teach the Fortran, C++ or C programming languages.

Audience Description

This manual is intended for scientists and engineers using the NVIDIA HPC compilers. To use these compilers, you should be aware of the role of high-level languages, such as Fortran, C++ and C as well as parallel programming models such as CUDA, OpenACC and OpenMP in the software development process, and you should have some level of understanding of programming. The NVIDIA HPC compilers are available on a variety of NVIDIA GPUs and x86-64, OpenPOWER and Arm CPU-based platforms and operating systems. You need to be familiar with the basic commands available on your system.

Compatibility and Conformance to Standards

Your system needs to be running a properly installed and configured version of the NVIDIA HPC compilers. For information on installing NVIDIA HPC compilers, refer to the Release Notes and Installation Guide included with your software.

For further information, refer to the following:

- ▶ *American National Standard Programming Language FORTRAN*, ANSI X3. -1978 (1978).

- ▶ *ISO/IEC 1539-1 : 1991, Information technology – Programming Languages – Fortran*, Geneva, 1991 (Fortran 90).
- ▶ *ISO/IEC 1539-1 : 1997, Information technology – Programming Languages – Fortran*, Geneva, 1997 (Fortran 95).
- ▶ *ISO/IEC 1539-1 : 2004, Information technology – Programming Languages – Fortran*, Geneva, 2004 (Fortran 2003).
- ▶ *ISO/IEC 1539-1 : 2010, Information technology – Programming Languages – Fortran*, Geneva, 2010 (Fortran 2008).
- ▶ *ISO/IEC 1539-1 : 2018, Information technology – Programming Languages – Fortran*, Geneva, 2018 (Fortran 2018).
- ▶ *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- ▶ *The Fortran 2003 Handbook*, Adams et al, Springer, 2009.
- ▶ *OpenACC Application Program Interface*, Version 2.7, November 2018, <http://www.openacc.org>.
- ▶ *OpenMP Application Program Interface*, Version 4.5, November 2015, <http://www.openmp.org>.
- ▶ *Programming in VAX Fortran*, Version 4.0, Digital Equipment Corporation (September, 1984).
- ▶ *IBM VS Fortran*, IBM Corporation, Rev. GC26-4119.
- ▶ *Military Standard, Fortran*, DOD Supplement to American National Standard Programming Language Fortran, ANSI x.3-1978, MIL-STD-1753 (November 9, 1978).
- ▶ *American National Standard Programming Language C*, ANSI X3.159-1989.
- ▶ *ISO/IEC 9899:1990, Information technology – Programming Languages – C*, Geneva, 1990 (C90).
- ▶ *ISO/IEC 9899:1999, Information technology – Programming Languages – C*, Geneva, 1999 (C99).
- ▶ *ISO/IEC 9899:2011, Information Technology – Programming Languages – C*, Geneva, 2011 (C11).
- ▶ *ISO/IEC 14882:2011, Information Technology – Programming Languages – C++*, Geneva, 2011 (C++11).
- ▶ *ISO/IEC 14882:2014, Information Technology – Programming Languages – C++*, Geneva, 2014 (C++14).
- ▶ *ISO/IEC 14882:2017, Information Technology – Programming Languages – C++*, Geneva, 2017 (C++17).

Organization

This guide contains the essential information on how to use the NVIDIA HPC compilers and is divided into these sections:

Getting Started provides an introduction to the NVIDIA HPC compilers and describes their use and overall features.

Use Command-line Options provides an overview of the command-line options as well as task-related lists of options.

Multicore CPU Optimization describes multicore CPU optimizations and related compiler options.

Using Function Inlining describes how to use function inlining and shows how to create an inline library.

Using OpenMP for Multicore CPUs describes how to use OpenMP for multicore CPU programming.

Using an NVIDIA GPU describes how to use an NVIDIA GPU and gives an introduction to using OpenACC.

PCAST describes how to use the Parallel Compiler Assisted Testing features of the HPC Compilers.

Using MPI describes how to use MPI with the NVIDIA HPC compilers.

Creating and Using Libraries discusses NVIDIA HPC compiler support libraries, shared object files, and environment variables that affect the behavior of the compilers.

Environment Variables describes the environment variables that affect the behavior of the NVIDIA HPC compilers.

Distributing Files – Deployment describes the deployment of your files once you have built, debugged and compiled them successfully.

Inter-language Calling provides examples showing how to place C language calls in a Fortran program and Fortran language calls in a C program.

Programming Considerations for 64-Bit Environments discusses issues of which programmers should be aware when targeting 64-bit processors.

C++ and C Inline Assembly and Intrinsics describes how to use inline assembly code in C++ and C programs, as well as how to use intrinsic functions that map directly to assembly machine instructions.

Hardware and Software Constraints

This guide describes versions of the NVIDIA HPC compilers that target NVIDIA GPUs and x86-64, OpenPOWER and Arm CPUs. Details concerning environment-specific values and defaults and system-specific features or limitations are presented in the release notes delivered with the NVIDIA HPC compilers.

Conventions

This guide uses the following conventions:

italic

is used for emphasis.

Constant Width

is used for filenames, directories, arguments, options, examples, and for language statements in the text, including assembly language statements.

Bold

is used for commands.

[item1]

in general, square brackets indicate optional items. In this case item1 is optional. In the context of p/t-sets, square brackets are required to specify a p/t-set.

{ item2 | item 3 }

braces indicate that a selection is required. In this case, you must select either item2 or item3.

filename ...

ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.

FORTRAN

Fortran language statements are shown in the text of this guide using a reduced fixed point size.

C++ and C

C++ and C language statements are shown in the text of this guide using a reduced fixed point size.

Terms

A number of terms related to systems, processors, compilers and tools are used throughout this guide. For example:

accelerator	FMA	-mcmmodel=medium	shared library
AVX	host	-mcmmodel=small	SIMD
CUDA	hyperthreading (HT)	MPI	SSE
device	large arrays	MPICH	static linking
driver	linux86-64	NUMA	x86-64
DWARF	LLVM	OpenPOWER	Arm
dynamic library	multicore	ppc64le	Aarch64

The following table lists the NVIDIA HPC compilers and their corresponding commands:

Table 1 NVIDIA HPC Compilers and Commands

Compiler or Tool	Language or Function	Command
NVFORTRAN	ISO/ANSI Fortran 2003	nvfortran
NVC++	ISO/ANSI C++17 with GNU compatibility	nvc++
NVC	ISO/ANSI C11	nvc

In general, the designation *NVFORTRAN* is used to refer to the NVIDIA Fortran compiler, and *nvfortran* is used to refer to the command that invokes the compiler. A similar convention is used for each of the NVIDIA HPC compilers.

For simplicity, examples of command-line invocation of the compilers generally reference the `nvfortran` command, and most source code examples are written in Fortran. Use of `NVC++` and `NVC` is consistent with `NVFORTRAN`, though there are command-line options and features of these compilers that do not apply to `NVFORTRAN`, and vice versa.

There are a wide variety of x86-64 CPUs in use. Most of these CPUs are forward-compatible, but not backward-compatible, meaning that code compiled to target a given processor will not necessarily execute correctly on a previous-generation processor.

A table listing the processor options that NVIDIA HPC compilers support is available in the Release Notes. The table also includes the features utilized by the compilers that distinguish them from a compatibility standpoint.

In this manual, the convention is to use "x86-64" to specify the group of CPUs that are x86-compatible, 64-bit enabled, and run a 64-bit operating system. x86-64 processors can differ in terms of their support for various prefetch, SSE and AVX instructions. Where such distinctions are important with respect to a given compiler option or feature, it is explicitly noted in this manual.

Related Publications

The following documents contain additional information related to the NVIDIA HPC compilers.

- ▶ *System V Application Binary Interface Processor Supplement* by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).
- ▶ *System V Application Binary Interface X86-64 Architecture Processor Supplement*.
- ▶ *OpenPOWER ABI for Linux Supplement, Power Architecture 64-Bit ELF V2 ABI Specification*, http://openpowerfoundation.org/wp-content/uploads/2016/03/ABI64BitOpenPOWERv1.1_16July2015_pub4.pdf.
- ▶ *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- ▶ *Programming in VAX Fortran, Version 4.0*, Digital Equipment Corporation (September, 1984).
- ▶ *IBM VS Fortran*, IBM Corporation, Rev. GC26-4119.
- ▶ *The C Programming Language* by Kernighan and Ritchie (Prentice Hall).
- ▶ *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele Jr. (Prentice Hall, 1987).
- ▶ *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup, AT&T Bell Laboratories, Inc. (Addison-Wesley Publishing Co., 1990).

Chapter 1.

GETTING STARTED

This section describes how to use the NVIDIA HPC compilers.

1.1. Overview

The command used to invoke a compiler, such as the `nvfortran` command, is called a *compiler driver*. The compiler driver controls the following phases of compilation: preprocessing, compiling, assembling, and linking. Once a file is compiled and an executable file is produced, you can execute, debug, or profile the program on your system.

In general, using an NVIDIA HPC compiler involves three steps:

1. Produce program source code in a file containing a `.f` extension or another appropriate extension, as described in [Input Files](#). This program may be one that you have written or one that you are modifying.
2. Compile the program using the appropriate compiler command.
3. Execute, debug, or profile the executable file on your system.

You might also want to deploy your application, though this is not a required step.

The NVIDIA HPC compilers allow many variations on these general program development steps. These variations include the following:

- ▶ Stop the compilation after preprocessing, compiling or assembling to save and examine intermediate results.
- ▶ Provide options to the driver that control compiler optimization or that specify various features or limitations.
- ▶ Include as input intermediate files such as preprocessor output, compiler output, or assembler output.

1.2. Creating an Example

Let's look at a simple example of using the NVIDIA Fortran compiler to create, compile, and execute a program that prints:

hello

1. Create your program.

For this example, suppose you enter the following simple Fortran program in the file `hello.f`:

```
print *, "hello"
end
```

2. Compile the program.

When you created your program, you called it `hello.f`. In this example, we compile it from a shell command prompt using the default `nvfortran` driver option. Use the following syntax:

```
$ nvfortran hello.f
```

By default, the executable output is placed in the file `a.out`. However, you can specify an output file name by using the `-o` option.

To place the executable output in the file `hello`, use this command:

```
$ nvfortran -o hello hello.f
```

3. Execute the program.

To execute the resulting hello program, simply type the filename at the command prompt and press the **Return** or **Enter** key on your keyboard:

```
$ hello
```

Below is the expected output:

```
hello
```

1.3. Invoking the Command-level NVIDIA HPC Compilers

To translate and link a Fortran, C, or C++ program, the `nvfortran`, `nvc` and `nvc++` commands do the following:

1. Preprocess the source text file.
2. Check the syntax of the source text.
3. Generate an assembly language file.
4. Pass control to the subsequent assembly and linking steps.

1.3.1. Command-line Syntax

The compiler command-line syntax, using `nvfortran` as an example, is:

```
nvfortran [options] [path]filename [...]
```

Where:

options

is one or more command-line options, all of which are described in detail in [Use Command-line Options](#).

path

is the pathname to the directory containing the file named by filename. If you do not specify the path for a filename, the compiler uses the current directory. You must specify the path separately for each filename not in the current directory.

filename

is the name of a source file, preprocessed source file, assembly-language file, object file, or library to be processed by the compilation system. You can specify more than one [path]filename.

1.3.2. Command-line Options

The command-line options control various aspects of the compilation process. For a complete alphabetical listing and a description of all the command-line options, refer to [Use Command-Line Options](#).

The following list provides important information about proper use of command-line options.

- ▶ Command-line options and their arguments are case sensitive.
- ▶ The compiler drivers recognize characters preceded by a hyphen (-) as command-line options. For example, the `-Mlist` option specifies that the compiler creates a listing file.



The convention for the text of this manual is to show command-line options using a dash instead of a hyphen; for example, you see `-Mlist`.

- ▶ The order of options and the filename is flexible. That is, you can place options before and after the filename argument on the command line. However, the placement of some options is significant, such as the `-l` option, in which the order of the filenames determines the search order.



If two or more options contradict each other, the last one in the command line takes precedence.

1.4. Filename Conventions

The NVIDIA HPC compilers use the filenames that you specify on the command line to find and to create input and output files. This section describes the input and output filename conventions for the phases of the compilation process.

1.4.1. Input Files

You can specify assembly-language files, preprocessed source files, Fortran/C/C++ source files, object files, and libraries as inputs on the command line. The compiler driver determines the type of each input file by examining the filename extensions.

The drivers use the following conventions:

filename.f
indicates a Fortran source file.

filename.F
indicates a Fortran source file that can contain macros and preprocessor directives (to be preprocessed).

filename.FOR
indicates a Fortran source file that can contain macros and preprocessor directives (to be preprocessed).

filename.F90
indicates a Fortran 90/95 source file that can contain macros and preprocessor directives (to be preprocessed).

filename.F95
indicates a Fortran 90/95 source file that can contain macros and preprocessor directives (to be preprocessed).

filename.f90
indicates a Fortran 90/95 source file that is in freeform format.

filename.f95
indicates a Fortran 90/95 source file that is in freeform format.

filename.cuf
indicates a Fortran 90/95 source file in free format with CUDA Fortran extensions.

filename.CUF
indicates a Fortran 90/95 source file in free format with CUDA Fortran extensions and that can contain macros and preprocessor directives (to be preprocessed).

filename.c
indicates a C source file that can contain macros and preprocessor directives (to be preprocessed).

filename.C
indicates a C++ source file that can contain macros and preprocessor directives (to be preprocessed).

filename.i
indicates a preprocessed C or C++ source file.

filename.cc
indicates a C++ source file that can contain macros and preprocessor directives (to be preprocessed).

filename.cpp
indicates a C++ source file that can contain macros and preprocessor directives (to be preprocessed).

filename.s
indicates an assembly-language file.

filename.o
(Linux) indicates an object file.

filename.a
(Linux) indicates a library of object files.

filename.so
(Linux only) indicates a library of shared object files.

The driver passes files with `.s` extensions to the assembler and files with `.o`, `.so` and `.a` extensions to the linker. Input files with unrecognized extensions, or no extension, are also passed to the linker.

Files with a `.F` (Capital F) or `.FOR` suffix are first preprocessed by the Fortran compilers and the output is passed to the compilation phase. The Fortran preprocessor functions like `cpp` for C programs, but is built in to the Fortran compilers rather than implemented through an invocation of `cpp`. This design ensures consistency in the preprocessing step regardless of the type or revision of operating system under which you are compiling.

Any input files not needed for a particular phase of processing are not processed. For example, if on the command line you specify an assembly-language file (`filename.s`) and the `-S` option to stop before the assembly phase, the compiler takes no action on the assembly language file. Processing stops after compilation and the assembler does not run. In this scenario, the compilation must have been completed in a previous pass which created the `.s` file. For a complete description of the `-S` option, refer to [Output Files](#).

In addition to specifying primary input files on the command line, code within other files can be compiled as part of include files using the `INCLUDE` statement in a Fortran source file or the preprocessor `#include` directive in Fortran source files that use a `.F` extension or C++ and C source files.

When linking a program with a library, the linker extracts only those library components that the program needs. The compiler drivers link in several libraries by default. For more information about libraries, refer to [Create and Use Libraries](#).

1.4.2. Output Files

By default, an executable output file produced by one of the NVIDIA HPC compilers is placed in the file `a.out`. As the [Hello example](#) shows, you can use the `-o` option to specify the output file name.

If you use option `-F` (Fortran only), `-P` (C/C++ only), `-S` or `-c`, the compiler produces a file containing the output of the last completed phase for each input file, as specified by the option supplied.

The output file is a preprocessed source file, an assembly-language file, or an unlinked object file respectively. Similarly, the `-E` option does not produce a file, but displays the preprocessed source file on the standard output. Using any of these options, the `-o` option is valid only if you specify a single input file. If no errors occur during processing, you can use the files created by these options as input to a future invocation of any of the NVIDIA compiler drivers.

The following table lists the stop-after options and the output files that the compilers create when you use these options. It also indicates the accepted input files.

Table 2 Option Descriptions

Option	Stop After	Input	Output
<code>-E</code>	preprocessing	Source files	preprocessed file to standard out
<code>-F</code>	preprocessing	Source files. This option is not valid for <code>nvc</code> or <code>nvc++</code> .	preprocessed file (<code>.f</code>)

Option	Stop After	Input	Output
-P	preprocessing	Source files. This option is not valid for <code>nvfortran</code> .	preprocessed file (<code>.i</code>)
-S	compilation	Source files or preprocessed files	assembly-language file (<code>.s</code>)
-c	assembly	Source files, or preprocessed files, or assembly-language files	unlinked object file (<code>.o</code> or <code>.obj</code>)
none	linking	Source files, or preprocessed files, assembly-language files, object files, or libraries	executable file (<code>a.out</code>)

If you specify multiple input files or do not specify an object filename, the compiler uses the input filenames to derive corresponding default output filenames of the following form, where *filename* is the input filename without its extension:

filename.f

indicates a preprocessed file, if you compiled a Fortran file using the `-F` option.

filename.i

indicates a preprocessed file, if you compiled using the `-P` option.

filename.lst

indicates a listing file from the `-Mlist` option.

filename.o or filename.obj

indicates a object file from the `-c` option.

filename.s

indicates an assembly-language file from the `-S` option.



Unless you specify otherwise, the destination directory for any output file is the current working directory. If the file exists in the destination directory, the compiler overwrites it.

The following example demonstrates the use of output filename extensions.

```
$ nvfortran -c proto.f proto1.F
```

This produces the output files `proto.o` and `proto1.o`, which are binary object files. Prior to compilation, the file `proto1.F` is preprocessed because it has a `.F` filename extension.

1.5. Fortran, C++ and C Data Types

The NVIDIA Fortran, C++ and C compilers recognize scalar and aggregate data types. A scalar data type holds a single value, such as the integer value 42 or the real value 112.6. An aggregate data type consists of one or more scalar data type objects, such as an array of integer values.

1.6. Platform-specific considerations

The NVIDIA HPC Compilers are supported on x86-64, OpenPOWER and 64-bit Arm multicore CPUs running Linux.

1.6.1. Using the NVIDIA HPC Compilers on Linux

Linux Header Files

The Linux system header files contain many GNU gcc extensions. The NVIDIA HPC C++ and C compilers support many of these extensions and can compile most programs that the GNU compilers can compile. A few header files not interoperable with the NVIDIA compilers have been rewritten.

If you are using the NVIDIA HPC C++ or C compilers, please make sure that the supplied versions of these include files are found before the system versions. This hierarchy happens by default unless you explicitly add a `-I` option that references one of the system `include` directories.

1.7. Site-Specific Customization of the Compilers

If you are using the NVIDIA HPC Compilers and want all your users to have access to specific libraries or other files, there are special files that allow you to customize the compilers for your site.

1.7.1. Use `siterc` Files

The NVIDIA HPC Compiler command-level drivers utilize a file named `siterc` to enable site-specific customization of the behavior of the NVIDIA compilers. The `siterc` file is located in the `bin` subdirectory of the NVIDIA HPC Compilers installation directory. Using `siterc`, you can control how the compiler drivers invoke the various components in the compilation tool chain.

1.7.2. Using User `rc` Files

In addition to the `siterc` file, user `rc` files can reside in a given user's home directory, as specified by the user's `HOME` environment variable. You can use these files to control the respective NVIDIA HPC Compilers. All of these files are optional.

On Linux, these files are named `.mynvfortranrc`, `.mynvcrc`, and `.mynvc++rc`.

The following examples show how you can use these `rc` files to tailor a given installation for a particular purpose on `Linux_x86_64` targets. The process is similar with obvious substitutions for `ppc64le` and `aarch64` targets.

Table 3 Examples of Using siterc and User rc Files

To do this...	Add the line shown to the indicated file(s)
Make available to all linux compilations the libraries found in /opt/newlibs/64	set SITELIB=/opt/newlibs/64; to /opt/nv/Linux_x86_64/20.9/compilers/bin/siterc
Add to all linux compilations a new library path: /opt/local/fast	append SITELIB=/opt/local/fast; to /opt/nv/Linux_x86_64/20.9/compilers/bin/siterc
With linux compilations, change -Mmpi to link in /opt/mympi/64/libmpix.a	set MPILIBDIR=/opt/mympi/64; set MPILIBNAME=mpix; to /opt/nv/Linux_x86_64/20.9/compilers/bin/siterc
Build a Fortran executable for linux that resolves shared objects in the relative directory ./REDIST	set set RPATH=./REDIST; to ~/.mynvfortranrc

1.8. Common Development Tasks

Now that you have a brief introduction to the compiler, let's look at some common development tasks that you might wish to perform.

- ▶ When you compile code you can specify a number of options on the command line that define specific characteristics related to how the program is compiled and linked, typically enhancing or overriding the default behavior of the compiler. For a list of the most common command line options and information on all the command line options, refer to [Use Command-line Options](#).
- ▶ Code optimization for multicore CPUs allows the compiler to organize your code for efficient execution. While possibly increasing compilation time and making the code more difficult to debug, these techniques typically produce code that runs significantly faster than code that does not use them. For more information on optimization refer to [Multicore CPU Optimization](#).
- ▶ Function inlining, a special type of optimization, replaces a call to a function or a subroutine with the body of the function or subroutine. This process can speed up execution by eliminating parameter passing and the function or subroutine call and return overhead. In addition, function inlining allows the compiler to optimize the function with the rest of the code. However, function inlining may also result in much larger code size with no increase in execution speed. For more information on function inlining, refer to [Using Function Inlining](#).
- ▶ A library is a collection of functions or subprograms used to develop software. Libraries contain "helper" code and data, which provide services to independent programs, allowing code and data to be shared and changed in a modular fashion. The functions and programs in a library are grouped for ease of use and linking.

When creating your programs, it is often useful to incorporate standard libraries or proprietary ones. For more information on this topic, refer to [Creating and Using Libraries](#).

- ▶ Environment variables define a set of dynamic values that can affect the way running processes behave on a computer. It is often useful to use these variables to set and pass information that alters the default behavior of the NVIDIA HPC Compilers and the executables which they generate. For more information on these variables, refer to [Environment Variables](#).
- ▶ Deployment, though possibly an infrequent task, can present some unique issues related to concerns of porting the code to other systems. Deployment, in this context, involves distribution of a specific file or set of files that are already compiled and configured. The distribution must occur in such a way that the application executes accurately on another system which may not be configured exactly the same as the system on which the code was created. For more information on what you might need to know to successfully deploy your code, refer to [Distributing Files – Deployment](#).
- ▶ An intrinsic is a function available in a given language whose implementation is handled specially by the compiler. Intrinsics make using processor-specific enhancements easier because they provide a C++ and C language interface to assembly instructions. In doing so, the compiler manages details that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.

Chapter 2.

USE COMMAND-LINE OPTIONS

A command line option allows you to control specific behavior when a program is compiled and linked. This section describes the syntax for properly using command-line options and provides a brief overview of a few of the more common options.

2.1. Command-line Option Overview

Before looking at all the command-line options, first become familiar with the syntax for these options. There are a large number of options available to you, yet most users only use a few of them. So, start simple and progress into using the more advanced options.

By default, the NVIDIA HPC Compilers generate code that is optimized for the type of processor on which compilation is performed, the compilation host. Before adding options to your command-line, review [Help with Command-line Options](#) and [Frequently-used Options](#).

2.1.1. Command-line Options Syntax

On a command-line, options need to be preceded by a hyphen (-). If the compiler does not recognize an option, you get an unknown switch error. The error can be downgraded to a warning by adding the `-noswitcherror` option.

This document uses the following notation when describing options:

[item]

Square brackets indicate that the enclosed item is optional.

{item | item}

Braces indicate that you must select one and only one of the enclosed items. A vertical bar (|) separates the choices.

...

Horizontal ellipses indicate that zero or more instances of the preceding item are valid.

2.1.2. Command-line Suboptions

Some options accept several suboptions. You can specify these suboptions either by using the full option statement multiple times or by using a comma-separated list for the suboptions.

The following two command lines are equivalent:

```
nvfortran -Mvect=simd -Mvect=noaltcode
```

```
nvfortran -Mvect=simd,noaltcode
```

2.1.3. Command-line Conflicting Options

Some options have an opposite or negated counterpart. For example, both `-Mvect` and `-Mnovect` are available. `-Mvect` enables vectorization and `-Mnovect` disables it. If you used both of these commands on a command line, they would conflict.



When you use conflicting options on a command line, the last encountered option takes precedence over any previous one.

The conflicting options rule is important for a number of reasons.

- ▶ Some options, such as `-fast`, include other options. Therefore, it is possible for you to be unaware that you have conflicting options.
- ▶ You can use this rule to create makefiles that apply specific flags to a set of files, as shown in the following example.

Example: Makefiles with Options

In this makefile fragment, `CCFLAGS` uses vectorization. `CCNOVECTFLAGS` uses the flags defined for `CCFLAGS` but disables vectorization.

```
CCFLAGS=c -Mvect=simd
CCNOVECTFLAGS=$(CCFLAGS) -Mnovect
```

2.2. Help with Command-line Options

If you are just getting started with the NVIDIA HPC Compilers, it is helpful to know which options are available, when to use them, and which options most users find effective.

Using `-help`

The `-help` option is useful because it provides information about all options supported by a given compiler.

You can use `-help` in one of three ways:

- ▶ Use `-help` with no parameters to obtain a list of all the available options with a brief one-line description of each.

- Add a parameter to `-help` to restrict the output to information about a specific option. The syntax for this usage is:

```
-help <command line option>
```

Suppose you use the following command to restrict the output to information about the `-fast` option:

```
$ nvfortran -help -fast
```

The output you see is similar to:

```
-fast Common optimizations; includes -O2 -Munroll=c:1 -Mnoframe -Mlre
```

In the following example, we add the `-help` parameter to restrict the output to information about the help command. The usage information for `-help` shows how groups of options can be listed or examined according to function.

```
$ nvfortran -help -help
      -help[=groups|asm|debug|language|linker|opt|other|overall|phase|
preprol
      suffix|switch|target|variable]
```

- Add a parameter to `-help` to restrict the output to a specific set of options or to a building process. The syntax for this usage is this:

```
-help=<subgroup>
```

2.3. Getting Started with Performance

This section provides a quick overview of a few of the command-line options that are useful in improving multicore CPU performance.

2.3.1. Using `-fast`

The NVIDIA HPC Compilers implement a wide range of options that allow users a fine degree of control on each optimization phase. When it comes to optimization of code, the quickest way to start is to use the option `-fast`. These options create a generally optimal set of flags. They incorporate optimization options to enable use of vector streaming SIMD instructions for 64-bit targets. They enable vectorization with SIMD instructions, cache alignment, and flush to zero mode.



The contents of the `-fast` option are host-dependent. Further, you should use these options on both compile and link command lines.

The following table shows the typical `-fast` options.

Table 4 Typical `-fast` Options

Use this option...	To do this...
<code>-O2</code>	Specifies a code optimization level of 2.
<code>-Munroll=c:1</code>	Unrolls loops, executing multiple instances of the original loop during each iteration.

Use this option...	To do this...
-Mnoframe	Do not generate code to set up a stack frame. Note: With this option, a stack trace does not work.
-Mlre	Enable loop-carried redundancy elimination.
-Mpre	Enable partial redundancy elimination

On most modern CPUs the `-fast` also includes the options shown in this table:

Table 5 Additional `-fast` Options

Use this option...	To do this...
-Mvect=simd	Generates packed SIMD instructions.
-Mcache_align	Aligns long objects on cache-line boundaries.
-Mflushz	Sets flush-to-zero mode.
-M[no]vect	Controls automatic vector pipelining.

To see the specific behavior of `-fast` for your target, use the following command:

```
$ nvfortran -help -fast
```

2.3.2. Other Performance-Related Options

While `-fast` is designed to be the quickest route to best performance, it is limited to routine boundaries. Depending on the nature and writing style of the source code, the compiler often can perform further optimization by knowing the global context of usage of a given routine. For instance, determining the possible value range of actual parameters of a routine could enable a loop to be vectorized; similarly, determining static occurrence of calls helps to decide which routine is beneficial to inline.

These types of global optimizations are under control of Interprocedural Analysis (IPA) in NVIDIA HPC Compilers. Option `-Mipa` enables Interprocedural Analysis. `-Mipa=fast` is the recommended option to get best performances for global optimization. You can also add the suboption `inline` to enable automatic global inlining across files. You might consider using `-Mipa=fast,inline`. This option for interprocedural analysis and global optimization can improve performance.

For more information on optimization, refer to [Multicore CPU Optimization](#). For specific information about these options, refer to the 'Optimization Controls' section of the [HPC Compilers Reference User Guide, docs.nvidia.com/hpc-sdk/compilers/pdf/hpc209ref.pdf](https://docs.nvidia.com/hpc-sdk/compilers/pdf/hpc209ref.pdf).

2.4. Frequently-used Options

In addition to overall performance, there are a number of other options that many users find useful when getting started. The following table provides a brief summary of these options.

Table 6 Commonly Used Command-Line Options

Use this option...	To do this...
<code>-acc</code>	Enable parallelization using OpenACC directives. By default the compilers will parallelize and offload OpenACC regions to an NVIDIA GPU. Use <code>-acc=multicore</code> to parallelize OpenACC regions for execution on all the cores of a multicore CPU.
<code>-gpu</code>	Control the type of GPU for which code is generated, the version of CUDA to be targeted, and several other aspects of GPU code generation.
<code>-stdpar</code>	(C++ only) Enable parallelization and offloading of C++17 Parallel Algorithms invocations to NVIDIA GPUs.
<code>-fast</code>	This option creates a generally optimal set of flags for targets that support SIMD capability. It incorporates optimization options to enable use of vector streaming SIMD instructions, cache alignment and flushz.
<code>-g</code>	Instructs the compiler to include symbolic debugging information in the object module; sets the optimization level to zero unless a <code>-O</code> option is present on the command line.
<code>-gopt</code>	Instructs the compiler to include symbolic debugging information in the object file, and to generate optimized code identical to that generated when <code>-g</code> is not specified.
<code>-help</code>	Provides information about available options.
<code>-mcmodel=medium</code>	Enables medium-model core generation for 64-bit targets, which is useful when the data space of the program exceeds 4GB.
<code>-Mconcur</code>	Instructs the compiler to enable auto-concurrentization of loops. If specified, the compiler uses multiple CPU cores to execute loops that it determines to be parallelizable; thus, loop iterations are split to execute optimally in a multithreaded execution context.
<code>-Minfo</code>	Instructs the compiler to produce information on standard error.
<code>-Minline</code>	Enables function inlining.
<code>-Mipa=fast,inline</code>	Enables interprocedural analysis and optimization. Also enables automatic procedure inlining.
<code>-Mkeepasm</code>	Keeps the generated assembly files.
<code>-Munroll</code>	Invokes the loop unroller to unroll loops, executing multiple instances of the loop during each iteration. This also sets the optimization level to 2 if the level is set to less than 2, or if no <code>-O</code> or <code>-g</code> options are supplied.
<code>-M[no]vect</code>	Enables/Disables the code vectorizer.
<code>--[no_]exceptions</code>	Removes exception handling from user code. For C++, declares that the functions in this file generate no C++ exceptions, allowing more optimal code generation.
<code>-o</code>	Names the output file.
<code>-O <level></code>	Specifies code optimization level where <level> is 0, 1, 2, 3, or 4.
<code>-tp <target></code>	Specify a CPU target other than the compilation host CPU.

Use this option...	To do this...
<code>-Wl, <option></code>	Compiler driver passes the specified options to the linker.

Chapter 3.

MULTICORE CPU OPTIMIZATION

Source code that is readable, maintainable, and produces correct results is not always organized for efficient execution. Normally, the first step in the program development process involves producing code that executes and produces the correct results. This first step usually involves compiling without much worry about optimization. After code is compiled and debugged, code optimization and parallelization become an issue.

Invoking one of the NVIDIA HPC Compiler commands with certain options instructs the compiler to generate optimized code. Optimization is not always performed since it increases compilation time and may make debugging difficult. However, optimization produces more efficient code that usually runs significantly faster than code that is not optimized.

The compilers optimize code according to the specified optimization level. You can use a number of options to specify the optimization levels, including `-O`, `-Mvect`, `-Mipa` and `-Mconcur`. In addition, you can use several of the `-M<nvflag>` switches to control specific types of optimization.

This chapter describes the overall effect of the optimization options supported by the NVIDIA HPC Compilers, and basic usage of several options.

3.1. Overview of Optimization

In general, optimization involves using transformations and replacements that generate more efficient code. This is done by the compiler and involves replacements that are independent of the particular target processor's architecture as well as replacements that take advantage of the x86-64/OpenPOWER architecture, instruction set and registers.

For discussion purposes, we categorize optimization:

- Local Optimization

- Global Optimization

- Loop Optimization

- Interprocedural Analysis (IPA) and Optimization

- Optimization Through Function Inlining

3.1.1. Local Optimization

A basic block is a sequence of statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching, except at the end. Local optimization is performed on a block-by-block basis within a program's basic blocks.

The NVIDIA HPC Compilers perform many types of local optimization including: algebraic identity removal, constant folding, common sub-expression elimination, redundant load and store elimination, scheduling, strength reduction, and peephole optimizations.

3.1.2. Global Optimization

This optimization is performed on a subprogram/function over all its basic blocks. The optimizer performs control-flow and data-flow analysis for an entire program unit. All loops, including those formed by ad hoc branches such as IFs or GOTOs, are detected and optimized.

Global optimization includes: constant propagation, copy propagation, dead store elimination, global register allocation, invariant code motion, and induction variable elimination.

3.1.3. Loop Optimization: Unrolling, Vectorization and Parallelization

The performance of certain classes of loops may be improved through vectorization or unrolling options. Vectorization transforms loops to improve memory access performance and make use of packed SSEvector instructions which perform the same operation on multiple data items concurrently. Unrolling replicates the body of loops to reduce loop branching overhead and provide better opportunities for local optimization, vectorization and scheduling of instructions. Performance for loops on systems with multiple processors may also improve using the parallelization features of the NVIDIA HPC Compilers.

3.1.4. Interprocedural Analysis (IPA) and Optimization

Interprocedural analysis (IPA) allows use of information across function call boundaries to perform optimizations that would otherwise be unavailable. For example, if the actual argument to a function is in fact a constant in the caller, it may be possible to propagate that constant into the callee and perform optimizations that are not valid if the dummy argument is treated as a variable. A wide range of optimizations are enabled or improved by using IPA, including but not limited to data alignment optimizations, argument removal, constant propagation, pointer disambiguation, pure function detection, F90/F95 array shape propagation, data placement, empty function removal, automatic function inlining, inlining of functions from pre-compiled libraries, and interprocedural optimization of functions from pre-compiled libraries.

3.1.5. Function Inlining

This optimization allows a call to a function to be replaced by a copy of the body of that function. This optimization will sometimes speed up execution by eliminating the function call and return overhead. Function inlining may also create opportunities for other types of optimization. Function inlining is not always beneficial. When used improperly it may increase code size and generate less efficient code.

3.2. Getting Started with Optimization

The first concern should be getting the program to execute and produce correct results. To get the program running, start by compiling and linking without optimization. Add `-O0` to the compile line to select no optimization; or add `-g` to debug the program easily and isolate any coding errors exposed during porting.

To get started quickly with optimization, a good set of options to use with any of the NVIDIA HPC compilers is `-fast`. For example:

```
$ nvfortran -fast -Mipa=fast,inline prog.f
```

For all of the NVIDIA HPC Fortran, C++ and C compilers, the `-fast -Mipa=fast,inline` options generally produce code that is well-optimized without the possibility of significant slowdowns due to pathological cases.

- ▶ The `-fast` option is an aggregate option that includes a number of individual NVIDIA compiler options; which compiler options are included depends on the target for which compilation is performed.
- ▶ The `-Mipa=fast,inline` option invokes interprocedural analysis (IPA), including several IPA suboptions. The `inline` suboption enables automatic inlining with IPA. If you do not wish to use automatic inlining, you can compile with `-Mipa=fast` and use several IPA suboptions without inlining.

These aggregate options incorporate a generally optimal set of flags for targets that support SIMD capability, including vectorization with SIMD instructions, cache alignment, and flushz.

The following table shows the typical `-fast` options.

Table 7 Typical `-fast` Options

Use this option...	To do this...
<code>-O2</code>	Specifies a code optimization level of 2 and <code>-Mvect=SIMD</code> .
<code>-Munroll=c:1</code>	Unrolls loops, executing multiple instances of the original loop during each iteration.
<code>-Mnoframe</code>	Indicates to not generate code to set up a stack frame. Note With this option, a stack trace does not work.
<code>-Mlre</code>	Indicates loop-carried redundancy elimination.
<code>-Mautoinline</code>	Enables automatic function inlining in C & C++.

Use this option...	To do this...
-Mpre	Indicates partial redundancy elimination

On modern multicore CPUs the `-fast` also typically includes the options shown in the following table:

Table 8 Additional `-fast` Options

Use this option...	To do this...
-Mvect=simd	Generates packed SSE and AVX instructions.
-Mcache_align	Aligns long objects on cache-line boundaries.
-Mflushz	Sets flush-to-zero mode.

By experimenting with individual compiler options on a file-by-file basis, further significant performance gains can sometimes be realized. However, depending on the coding style, individual optimizations can sometimes cause slowdowns, and must be used carefully to ensure performance improvements.

There are other useful command line options related to optimization and parallelization, such as `-help`, `-Minfo`, `-Mneginfo`, `-dryrun`, and `-v`.

3.2.1. `-help`

As described in [Help with Command-Line Options](#), you can see a specification of any command-line option by invoking any of the NVIDIA HPC Compilers with `-help` in combination with the option in question, without specifying any input files.

For example, you might want information on `-O`:

```
$ nvfortran -help -O
```

The resulting output is similar to this:

```
-O Set opt level. All -O1 optimizations plus traditional scheduling and
  global scalar optimizations performed
```

Or you can see the full functionality of `-help` itself, which can return information on either an individual option or groups of options:

```
$ nvfortran -help -help
```

The resulting output is similar to this:

```
-help[=groups|asm|debug|language|linker|opt|other|overall|
phase|prepro|suffix|switch|target|variable]
Show compiler switches
```

3.2.2. `-Minfo`

You can use the `-Minfo` option to display compile-time optimization listings. When this option is used, the NVIDIA HPC Compilers issue informational messages to standard error (stderr) as compilation proceeds. From these messages, you can determine which loops are optimized using unrolling, SIMD vectorization, parallelization, GPU

offloading, interprocedural optimizations and various miscellaneous optimizations. You can also see where and whether functions are inlined.

3.2.3. -Mneginfo

You can use the `-Mneginfo` option to display informational messages to standard error (stderr) that explain why certain optimizations are inhibited.

3.2.4. -dryrun

The `-dryrun` option can be useful as a diagnostic tool if you need to see the steps used by the compiler driver to preprocess, compile, assemble and link in the presence of a given set of command line inputs. When you specify the `-dryrun` option, these steps are printed to standard error (stderr) but are not actually performed. For example, you can use this option to inspect the default and user-specified libraries that are searched during the link phase, and the order in which they are searched by the linker.

3.2.5. -v

The `-v` option is similar to `-dryrun`, except each compilation step is performed and not simply printed.

3.3. Local and Global Optimization

This section describes local and global optimization.

3.3.1. -Msafepttr

The `-Msafepttr` option can significantly improve performance of C++ and C programs in which there is known to be no pointer aliasing. For obvious reasons, this command-line option must be used carefully. There are a number of suboptions for `-Msafepttr`:

- ▶ `-Msafepttr=all` – All pointers are safe. Equivalent to the default setting: `-Msafepttr`.
- ▶ `-Msafepttr=arg` – Function formal argument pointers are safe. Equivalent to `-Msafepttr=dummy`.
- ▶ `-Msafepttr=global` – Global pointers are safe.
- ▶ `-Msafepttr=local` – Local pointers are safe. Equivalent to `-Msafepttr=auto`.
- ▶ `-Msafepttr=static` – Static local pointers are safe.

If your C++ or C program has pointer aliasing and you also want automating inlining, then compiling with `-Mipa=fast` or `-Mipa=fast,inline` includes pointer aliasing optimizations. IPA may be able to optimize some of the alias references in your program and leave intact those that cannot be safely optimized.

3.3.2. -O

Using the NVIDIA HPC Compiler commands with the `-O<level>` option (the capital O is for Optimize), you can specify any integer level from 0 to 4.

-O0

Level zero specifies no optimization. A basic block is generated for each language statement. At this level, the compiler generates a basic block for each statement.

Performance will almost always be slowest using this optimization level. This level is useful for the initial execution of a program. It is also useful for debugging, since there is a direct correlation between the program text and the code generated. To enable debugging, include `-g` on your compile line.

-O1

Level one specifies local optimization. Scheduling of basic blocks is performed. Register allocation is performed.

Local optimization is a good choice when the code is very irregular, such as code that contains many short statements containing IF statements and does not contain loops (DO or DO WHILE statements). Although this case rarely occurs, for certain types of code, this optimization level may perform better than level-two (`-O2`).

-O

When no level is specified, level two global optimizations are performed, including traditional scalar optimizations, induction recognition, and loop invariant motion. No SIMD vectorization is enabled.

-O2

Level two specifies global optimization. This level performs all level-one local optimization as well as level two global optimization described in `-O`. In addition, more advanced optimizations such as SIMD code generation, cache alignment, and partial redundancy elimination are enabled.

-O3

Level three specifies aggressive global optimization. This level performs all level-one and level-two optimizations and enables more aggressive hoisting and scalar replacement optimizations that may or may not be profitable.

-O4

Level four performs all level-one, level-two, and level-three optimizations and enables hoisting of guarded invariant floating point expressions.

Types of Optimizations

The NVIDIA HPC Compilers perform many different types of local optimizations, including but not limited to:

- Algebraic identity removal
- Constant folding
- Common subexpression elimination
- Local register optimization
- Peephole optimizations
- Redundant load and store elimination
- Strength reductions

Level-two optimization (`-O2` or `-O`) specifies global optimization. The `-fast` option generally specifies global optimization; however, the `-fast` switch varies from release to release, depending on a reasonable selection of switches for any one particular release. The `-O` or `-O2` level performs all level-one local optimizations as well as global optimizations. Control flow analysis is applied and global registers are allocated for all functions and subroutines. Loop regions are given special consideration. This optimization level is a good choice when the program contains loops, the loops are short, and the structure of the code is regular.

The NVIDIA HPC Compilers perform many different types of global optimizations, including but not limited to:

- Branch to branch elimination
- Constant propagation
- Copy propagation
- Dead store elimination
- Global register allocation
- Induction variable elimination
- Invariant code motion

You can explicitly select the optimization level on the command line. For example, the following command line specifies level-two optimization which results in global optimization:

```
$ nvfortran -O2 prog.f
```

The default optimization level changes depending on which options you select on the command line. For example, when you select the `-g` debugging option, the default optimization level is set to level-zero (`-O0`). However, if you need to debug optimized code, you can use the `-gopt` option to generate debug information without perturbing

optimization. For a description of the default levels, refer to [Default Optimization Levels](#).

The `-fast` option includes `-O2` on all targets. If you want to override the default for `-fast` with `-O3` while maintaining all other elements of `-fast`, simply compile as follows:

```
$ nvfortran -fast -O3 prog.f
```

3.4. Loop Unrolling using `-Munroll`

This optimization unrolls loops, which reduces branch overhead, and can improve execution speed by creating better opportunities for instruction scheduling. A loop with a constant count may be completely unrolled or partially unrolled. A loop with a non-constant count may also be unrolled. A candidate loop must be an innermost loop containing one to four blocks of code.

The following example shows the use of the `-Munroll` option:

```
$ nvfortran -Munroll prog.f
```

The `-Munroll` option is included as part of `-fast` on all targets. The loop unroller expands the contents of a loop and reduces the number of times a loop is executed. Branching overhead is reduced when a loop is unrolled two or more times, since each iteration of the unrolled loop corresponds to two or more iterations of the original loop; the number of branch instructions executed is proportionately reduced. When a loop is unrolled completely, the loop's branch overhead is eliminated altogether.

Loop unrolling may be beneficial for the instruction scheduler. When a loop is completely unrolled or unrolled two or more times, opportunities for improved scheduling may be presented. The code generator can take advantage of more possibilities for instruction grouping or filling instruction delays found within the loop.

Examples Showing Effect of Unrolling

The following side-by-side examples show the effect of code unrolling on a segment that computes a dot product.



This example is only meant to represent how the compiler can transform the loop; it is not meant to imply that the programmer needs to manually change code. In fact, manually unrolling your code can sometimes inhibit the compiler's analysis and optimization.

Table 9 Example of Effect of Code Unrolling

Dot Product Code	Unrolled Dot Product Code
<pre>REAL*4 A(100), B(100), Z INTEGER I DO I=1, 100 Z = Z + A(i) * B(i)</pre>	<pre>REAL*4 A(100), B(100), Z INTEGER I DO I=1, 100, 2 Z = Z + A(i) * B(i)</pre>

Dot Product Code	Unrolled Dot Product Code
<pre>END DO END</pre>	<pre> Z = Z + A(i+1) * B(i+1) END DO END</pre>

Using the `-Minfo` option, the compiler informs you when a loop is being unrolled. For example, a message similar to the following, indicating the line number, and the number of times the code is unrolled, displays when a loop is unrolled:

```
dot:
  5, Loop unrolled 5 times
```

Using the `c:<m>` and `n:<m>` sub-options to `-Munroll`, or using `-Mnounroll`, you can control whether and how loops are unrolled on a file-by-file basis. For more information on `-Munroll`, refer to [Use Command-line Options](#).

3.5. Vectorization using `-Mvect`

The `-Mvect` option is included as part of `-fast` on all multicore CPU targets. If your program contains computationally-intensive loops, the `-Mvect` option may be helpful. If in addition you specify `-Minfo`, and your code contains loops that can be vectorized, the compiler reports relevant information on the optimizations applied.

When an NVIDIA HPC Compiler command is invoked with the `-Mvect` option, the vectorizer scans code searching for loops that are candidates for high-level transformations such as loop distribution, loop interchange, cache tiling, and idiom recognition (replacement of a recognizable code sequence, such as a reduction loop, with optimized code sequences or function calls). When the vectorizer finds vectorization opportunities, it internally rearranges or replaces sections of loops (the vectorizer changes the code generated; your source code's loops are not altered). In addition to performing these loop transformations, the vectorizer produces extensive data dependence information for use by other phases of compilation and detects opportunities to use vector or packed SIMD instructions on processors where these are supported.

The `-Mvect` option can speed up code which contains well-behaved countable loops which operate on large floating point arrays in Fortran and their C++ and C counterparts. However, it is possible that some codes will show a decrease in performance when compiled with the `-Mvect` option due to the generation of conditionally executed code segments, inability to determine data alignment, and other code generation factors. For this reason, it is recommended that you check carefully whether particular program units or loops show improved performance when compiled with this option enabled.

3.5.1. Vectorization Sub-options

The vectorizer performs high-level loop transformations on countable loops. A loop is countable if the number of iterations is set only before loop execution and cannot be modified during loop execution. Some of the vectorizer transformations can be

controlled by arguments to the `-Mvect` command line option. The following sections describe the arguments that affect the operation of the vectorizer. In addition, some of these vectorizer operations can be controlled from within code using directives and pragmas.

The vectorizer performs the following operations:

- ▶ Loop interchange
- ▶ Loop splitting
- ▶ Loop fusion
- ▶ Generation of SIMD instructions on CPUs where these are supported
- ▶ Generation of prefetch instructions on processors where these are supported
- ▶ Loop iteration peeling to maximize vector alignment
- ▶ Alternate code generation

The following table lists and briefly describes some of the `-Mvect` suboptions.

Table 10 -Mvect Suboptions

Use this option ...	To instruct the vectorizer to do this ...
<code>-Mvect=altcode</code>	Generate appropriate code for vectorized loops.
<code>-Mvect=[no]assoc</code>	Perform[disable] associativity conversions that can change the results of a computation due to a round-off error. For example, a typical optimization is to change one arithmetic operation to another arithmetic operation that is mathematically correct, but can be computationally different and generate faster code. This option is provided to enable or disable this transformation, since a round-off error for such associativity conversions may produce unacceptable results.
<code>-Mvect=fuse</code>	Enable loop fusion.
<code>-Mvect=gather</code>	Enable vectorization of indirect array references.
<code>-Mvect=idiom</code>	Enable idiom recognition.
<code>-Mvect=levels:<n></code>	Set the maximum next level of loops to optimize.
<code>-Mvect=nocond</code>	Disable vectorization of loops with conditions.
<code>-Mvect=partial</code>	Enable partial loop vectorization via inner loop distribution.
<code>-Mvect=prefetch</code>	Automatically generate prefetch instructions when vectorizable loops are encountered, even in cases where SSE/SIMD instructions are not generated.
<code>-Mvect=short</code>	Enable short vector operations.
<code>-Mvect=simd</code>	Automatically generate packed SSE (Streaming SIMD Extensions)/SIMD, and prefetch instructions when vectorizable loops are encountered. SIMD instructions, first introduced on Pentium III and AthlonXP processors, operate on single-precision floating-point data.

Use this option ...	To instruct the vectorizer to do this ...
-Mvect=sizelimit:n	Limit the size of vectorized loops.
-Mvect=sse	Equivalent to -Mvect=simd.
-Mvect=uniform	Perform consistent optimizations in both vectorized and residual loops. Be aware that this may affect the performance of the residual loop.



Inserting **no** in front of an option disables the option. For example, to disable the generation of SIMD instructions, compile with -Mvect=nosimd.

3.5.2. Vectorization Example Using SIMD Instructions

One of the most important vectorization options is -Mvect=simd. When you use this option, the compiler automatically generates SIMD vector instructions, where possible, when targeting processors on which these instructions are supported. This process can improve performance by several factors compared with the equivalent scalar code. All of the NVIDIA HPC Fortran, C++ and C compilers support this capability.

In the program in [Vector operation using SIMD instructions](#), the vectorizer recognizes the vector operation in subroutine 'loop' when either compiler switch -Mvect=simd or -fast is used. This example shows the compilation, informational messages, and runtime results using SIMD instructions on an Intel Core i7 7800X Skylake system, along with issues that affect SIMD performance.

Loops vectorized using SIMD instructions operate much more efficiently when processing vectors that are aligned to a cache-line boundary. You can cause unconstrained data objects of size 16 bytes or greater to be cache-aligned by compiling with the -Mcache_align switch. An unconstrained data object is a data object that is not a common block member and not a member of an aggregate data structure.



For stack-based local variables to be properly aligned, the main program or function must be compiled with -Mcache_align.

The -Mcache_align switch has no effect on the alignment of Fortran allocatable or automatic arrays. If you have arrays that are constrained, such as vectors that are members of Fortran common blocks, you must specifically pad your data structures to ensure proper cache alignment. You can use -Mcache_align for only the beginning address of each common block to be cache-aligned.

The following examples show the results of compiling the sample code in [Vector operation using SIMD instructions](#) both with and without the option -Mvect=simd.

Vector operation using SIMD instructions

```

program vector_op
  parameter (N = 9999)
  real*4 x(N), y(N), z(N), w(N)
  do i = 1, n
    y(i) = i
    z(i) = 2*i
    w(i) = 4*i
  
```



```

enddo
do j = 1, 200000
  call loop(x,y,z,w,1.0e0,N)
enddo
print *, x(1),x(771),x(3618),x(6498),x(9999)
end

```

```

subroutine loop(a,b,c,d,s,n)
  integer i, n
  real*4 a(n), b(n), c(n), d(n),s
  do i = 1, n
    a(i) = b(i) + c(i) - s * d(i)
  enddo
end

```

Assume the preceding program is compiled as follows, where `-Mvect=nosimd` disables SIMD vectorization:

```

% nvfortran -fast -Mvect=nosimd -Minfo vadd.f -Mfree -o vadd
vector_op:
  4, Loop unrolled 16 times
    Generated 1 prefetches in scalar loop
  9, Loop not vectorized/parallelized: contains call
loop:
  18, Loop unrolled 8 times
    FMA (fused multiply-add) instruction(s) generated

```

The following output shows a sample result if the generated executable is run and timed on an Intel Core i7 7800X Skylake system:

```

$ /bin/time vadd
-1.000000      -771.0000      -3618.000      -6498.000
-9999.000
0.99user 0.01system 0:01.18elapsed 84%CPU (0avgtext+0avgdata 3120maxresident)k
7736inputs+0outputs (4major+834minor)pagefaults 0swaps

$ /bin/time vadd
-1.000000      -771.0000      -3618.000      -6498.000
-9999.000
2.31user 0.00system 0:02.57elapsed 89%CPU (0avgtext+0avgdata 6976maxresident)k
8192inputs+0outputs (4major+149minor)pagefaults 0swaps

```

Now, recompile with vectorization enabled, and you see results similar to these:

```

% nvfortran -fast -Minfo vadd.f -Mfree -o vadd
vector_op:
  4, Loop not vectorized: may not be beneficial
    Unrolled inner loop 8 times
    Residual loop unrolled 7 times (completely unrolled)
    Generated 1 prefetches in scalar loop
  9, Loop not vectorized/parallelized: contains call
loop:
  18, Generated 2 alternate versions of the loop
    Generated vector simd code for the loop
    Generated 3 prefetch instructions for the loop
    Generated vector simd code for the loop
    Generated 3 prefetch instructions for the loop
    Generated vector simd code for the loop
    Generated 3 prefetch instructions for the loop
    FMA (fused multiply-add) instruction(s) generated

```

Notice the informational messages for the loop at line 18. The first line of the message indicates that two alternate versions of the loop were generated. The loop count and alignments of the arrays determine which of these versions is executed. The next several

lines indicate the loop was vectorized and that prefetch instructions have been generated for three loads to minimize latency of data transfers from main memory.

Executing again, you should see results similar to the following:

```
$ /bin/time vadd-simd
-1.000000      -771.0000      -3618.000      -6498.000
-9999.000
0.27user 0.00system 0:00.29elapsed 93%CPU (0avgtext+0avgdata 3124maxresident)k
0inputs+0outputs (0major+838minor)pagefaults 0swaps

$ /bin/time vadd-simd
-1.000000      -771.0000      -3618.000      -6498.000
-9999.000
0.62user 0.00system 0:00.65elapsed 95%CPU (0avgtext+0avgdata 6976maxresident)k
0inputs+0outputs (0major+151minor)pagefaults 0swaps
```

The SIMD result is 3.7 times faster than the equivalent non-SIMD version of the program.

Speed-up realized by a given loop or program can vary widely based on a number of factors:

- ▶ When the vectors of data are resident in the data cache, performance improvement using SIMD instructions is most effective.
- ▶ If data is aligned properly, performance will be better in general than when using SIMD operations on unaligned data.
- ▶ If the compiler can guarantee that data is aligned properly, even more efficient sequences of SIMD instructions can be generated.
- ▶ The efficiency of loops that operate on single-precision data can be higher. SIMD instructions can operate on four single-precision elements concurrently, but only two double-precision elements.



Compiling with `-Mvect=simd` can result in numerical differences from the executables generated with less optimization. Certain vectorizable operations, for example dot products, are sensitive to order of operations and the associative transformations necessary to enable vectorization (or parallelization).

3.6. Interprocedural Analysis and Optimization using `-Mipa`

The NVIDIA HPC Fortran, C++ and C compilers use interprocedural analysis (IPA) that results in minimal changes to makefiles and the standard edit-build-run application development cycle. Other than adding `-Mipa` to the command line, no other changes are required. For reference and background, the process of building a program without IPA is described later in this section, followed by the minor modifications required to use IPA with the NVIDIA compilers. While the NVC compiler is used here to show how IPA works, similar capabilities apply to each of the NVIDIA HPC Fortran, C++ and C compilers.

3.6.1. Building a Program Without IPA – Single Step

Using the `nvc` command-level compiler driver, multiple source files can be compiled and linked into a single executable with one command. The following example compiles and links three source files:

```
% nvc -o a.out file1.c file2.c file3.c
```

In actuality, the `nvc` driver executes several steps to produce the assembly code and object files corresponding to each source file, and subsequently to link the object files together into a single executable file. This command is roughly equivalent to the following commands performed individually:

```
% nvc -S -o file1.s file1.c
% as -o file1.o file1.s
% nvc -S -o file2.s file2.c
% as -o file2.o file2.s
% nvc -S -o file3.s file3.c
% as -o file3.o file3.s
% nvc -o a.out file1.o file2.o file3.o
```

If any of the three source files is edited, the executable can be rebuilt with the same command line:

```
% nvc -o a.out file1.c file2.c file3.c
```



This always works as intended, but has the side-effect of recompiling all of the source files, even if only one has changed. For applications with a large number of source files, this can be time-consuming and inefficient.

3.6.2. Building a Program Without IPA – Several Steps

It is also possible to use individual `nvc` commands to compile each source file into a corresponding object file, and one to link the resulting object files into an executable:

```
% nvc -c file1.c
% nvc -c file2.c
% nvc -c file3.c
% nvc -o a.out file1.o file2.o file3.o
```

The `nvc` driver invokes the compiler and assembler as required to process each source file, and invokes the linker for the final link command. If you modify one of the source files, the executable can be rebuilt by compiling just that file and then relinking:

```
% nvc -c file1.c
% nvc -o a.out file1.o file2.o file3.o
```

3.6.3. Building a Program Without IPA Using Make

The program compilation and linking process can be simplified greatly using the `make` utility on systems where it is supported. Suppose you create a `makefile` containing the following lines:

```
a.out: file1.o file2.o file3.o
    nvc $(OPT) -o a.out file1.o file2.o file3.o
file1.o: file1.c
    nvc $(OPT) -c file1.c
file2.o: file2.c
    nvc $(OPT) -c file2.c
```

```
file3.o: file3.c
nvc $(OPT) -c file3.c
```

It is then possible to type a single make command:

```
% make
```

The `make` utility determines which object files are out of date with respect to their corresponding source files, and invokes the compiler to recompile only those source files and to relink the executable. If you subsequently edit one or more source files, the executable can be rebuilt with the minimum number of recompilations using the same single `make` command.

3.6.4. Building a Program with IPA

Interprocedural analysis and optimization (IPA) by the NVIDIA HPC Compilers alters the standard and `make` utility command-level interfaces as little as possible. IPA occurs in three phases:

- ▶ **Collection:** Create a summary of each function or procedure, collecting the useful information for interprocedural optimizations. This is done during the compile step if the `-Mipa` switch is present on the command line; summary information is collected and stored in the object file.
- ▶ **Propagation:** Process all the object files to propagate the interprocedural summary information across function and file boundaries. This is done during the link step, when all the object files are combined, if the `-Mipa` switch is present on the link command line.
- ▶ **Recompile/Optimization:** Recompile each of the object files with the propagated interprocedural information, producing a specialized object file. This process is also performed during the link step when the `-Mipa` switch is present on the link command line.

When linking with `-Mipa`, the NVIDIA HPC Compilers automatically regenerate IPA-optimized versions of each object file, essentially recompiling each file. If there are IPA-optimized objects from a previous build, the compilers will minimize the recompile time by reusing those objects if they are still valid. They will still be valid if the IPA-optimized object is newer than the original object file, and the propagated IPA information for that file has not changed since it was optimized.

After each object file has been recompiled, the regular linker is invoked to build the application with the IPA-optimized object files. The IPA-optimized object files are saved in the same directory as the original object files, for use in subsequent program builds.

3.6.5. Building a Program with IPA – Single Step

By adding the `-Mipa` command line switch, several source files can be compiled and linked with interprocedural optimizations with one command:

```
% nvc -Mipa=fast -o a.out file1.c file2.c file3.c
```

Just like compiling without `-Mipa`, the driver executes several steps to produce the assembly and object files to create the executable:

```
% nvc -Mipa=fast -S -o file1.s file1.c
% as -o file1.o file1.s
```

```
% nvc -Mipa=fast -S -o file2.s file2.c
% as -o file2.o file2.s
% nvc -Mipa=fast -S -o file3.s file3.c
% as -o file3.o file3.s
% nvc -Mipa=fast -o a.out file1.o file2.o file3.o
```

In the last step, an IPA linker is invoked to read all the IPA summary information and perform the interprocedural propagation. The IPA linker reinvokes the compiler on each of the object files to recompile them with interprocedural information. This creates three new objects with mangled names:

```
file1_ipa5_a.out.oo.o, file2_ipa5_a.out.oo.o, file3_ipa5_a.out.oo.o
```

The system linker is then invoked to link these IPA-optimized objects into the final executable. Later, if one of the three source files is edited, the executable can be rebuilt with the same command line:

```
% nvc -Mipa=fast -o a.out file1.c file2.c file3.c
```

This works, but again has the side-effect of compiling each source file, and recompiling each object file at link time.

3.6.6. Building a Program with IPA – Several Steps

Just by adding the `-Mipa` command-line switch, it is possible to use individual `nvc` commands to compile each source file, followed by a command to link the resulting object files into an executable:

```
% nvc -Mipa=fast -c file1.c
% nvc -Mipa=fast -c file2.c
% nvc -Mipa=fast -c file3.c
% nvc -Mipa=fast -o a.out file1.o file2.o file3.o
```

The `nvc` driver invokes the compiler and assembler as required to process each source file, and invokes the IPA linker for the final link command. If you modify one of the source files, the executable can be rebuilt by compiling just that file and then relinking:

```
% nvc -Mipa=fast -c file1.c
% nvc -Mipa=fast -o a.out file1.o file2.o file3.o
```

When the IPA linker is invoked, it will determine that the IPA-optimized object for `file1.o` (`file1_ipa5_a.out.oo.o`) is stale, since it is older than the object `file1.o`; and hence it needs to be rebuilt, and reinvokes the compiler to generate it. In addition, depending on the nature of the changes to the source file `file1.c`, the interprocedural optimizations previously performed for `file2` and `file3` may now be inaccurate. For instance, IPA may have propagated a constant argument value in a call from a function in `file1.c` to a function in `file2.c`; if the value of the argument has changed, any optimizations based on that constant value are invalid. The IPA linker determines which, if any, of the previously created IPA-optimized objects need to be regenerated; and, as appropriate, reinvokes the compiler to regenerate them. Only those objects that are stale or which have new or different IPA information are regenerated. This approach saves compile time.

3.6.7. Building a Program with IPA Using Make

As shown earlier, programs can be built with IPA using the make utility. Just add the command-line switch `-Mipa`, as shown here:

```
OPT=-Mipa=fast
a.out: file1.o file2.o file3.o
    nvc $(OPT) -o a.out file1.o file2.o file3.o
file1.o: file1.c
    nvc $(OPT) -c file1.c
file2.o: file2.c
    nvc $(OPT) -c file2.c
file3.o: file3.c
    nvc $(OPT) -c file3.c
```

Using the single `make` command invokes the compiler to generate any of the object files that are out-of-date, then invokes `nvc` to link the objects into the executable. At link time, `nvc` calls the IPA linker to regenerate any stale or invalid IPA-optimized objects.

```
% make
```

3.6.8. Questions about IPA

Question: Why is the object file so large?

Answer: An object file created with `-Mipa` contains several additional sections. One is the summary information used to drive the interprocedural analysis. In addition, the object file contains the compiler internal representation of the source file, so the file can be recompiled at link time with interprocedural optimizations. There may be additional information when inlining is enabled. The total size of the object file may be 5-10 times its original size. The extra sections are not added to the final executable.

Question: What if I compile with `-Mipa` and link without `-Mipa`?

Answer: The NVIDIA HPC Compilers generate a legal object file, even when the source file is compiled with `-Mipa`. If you compile with `-Mipa` and link without `-Mipa`, the linker is invoked on the original object files. A legal executable is generated. While this executable does not have the benefit of interprocedural optimizations, any other optimizations do apply.

Question: What if I compile without `-Mipa` and link with `-Mipa`?

Answer: At link time, the IPA linker must have summary information about all the functions or routines used in the program. This information is created only when a file is compiled with `-Mipa`. If you compile a file without `-Mipa` and then try to get interprocedural optimizations by linking with `-Mipa`, the IPA linker will issue a message that some routines have no IPA summary information, and will proceed to run the system linker using the original object files. If some files were compiled with `-Mipa` and others were not, it will determine the safest approximation of the IPA summary information for those files not compiled with `-Mipa`, and use that to recompile the other files using interprocedural optimizations.

Question: Can I build multiple applications in the same directory with `-Mipa`?

Answer: Yes. Suppose you have three source files: `main1.c`, `main2.c`, and `sub.c`, where `sub.c` is shared between the two applications. Suppose you build the first application with `-Mipa`, using this command:

```
% nvc -Mipa=fast -o app1 main1.c sub.c
```

The IPA linker creates two IPA-optimized object files and uses them to build the first application.

```
main1_ipa4_app1.o sub_ipa4_app1.o
```

Now suppose you build the second application using this command:

```
% nvc -Mipa=fast -o app2 main2.c sub.c
```

The IPA linker creates two more IPA-optimized object files:

```
main2_ipa4_app2.o sub_ipa4_app2.o
```



There are now three object files for `sub.c`: the original `sub.o`, and two IPA-optimized objects, one for each application in which it appears.

Question: How is the mangled name for the IPA-optimized object files generated?

Answer: The mangled name has `'_ipa'` appended, followed by the decimal number of the length of the executable file name, followed by an underscore and the executable file name itself. The suffix is changed to `.oo` so that linking `*.o` does not pull in the IPA-optimized objects. If the IPA linker determines that the file would not benefit from any interprocedural optimizations, it does not have to recompile the file at link time, and uses the original object.

Question: Can I use parallel make environments (e.g., `pmake`) with IPA?

Answer: No. IPA is not compatible with parallel make environments.

Chapter 4.

USING FUNCTION INLINING

Function inlining replaces a call to a function or a subroutine with the body of the function or subroutine. This can speed up execution by eliminating parameter passing and function/subroutine call and return overhead. It also allows the compiler to optimize the function with the rest of the code. Note that using function inlining indiscriminately can result in much larger code size and no increase in execution speed.

The NVIDIA HPC compilers provide two categories of inlining:

- ▶ **Automatic function inlining** – In C++ and C, you can inline static functions with the `inline` keyword by using the `-Mautoinline` option, which is included with `-fast`.
- ▶ **Function inlining** – You can inline functions which were extracted to the inline libraries in Fortran, C++ and C. There are two ways of enabling function inlining: with and without the `lib` suboption. For the latter, you create inline libraries, for example using the `nvfortran` compiler driver and the `-o` and `-Mextract` options.

There are important restrictions on inlining. Inlining only applies to certain types of functions. Refer to [Restrictions on Inlining](#) for more details on function inlining limitations.

This section describes how to use the following options related to function inlining:

```
-Mautoinline  
-Mextract  
-Minline  
-Mnoinline  
-Mrecursive
```

4.1. Automatic function inlining in C++ and C

To enable automatic function inlining in C++ and C for static functions with the `inline` keyword, use the `-Mautoinline` option (included in `-fast`). Use `-Mnoautoinline` to disable it.

These `-Mautoinline` suboptions let you determine the selection criteria, where `n` loosely corresponds to the number of lines in the procedure:

maxsize:n

Automatically inline functions size `n` and less

totalsize:n

Limit automatic inlining to total size of `n`

4.2. Invoking Procedure Inlining

To invoke the procedure inliner, use the `-Minline` option. If you do not specify an inline library, the compiler performs a special prepass on all source files named on the compiler command line before it compiles any of them. This pass extracts procedures that meet the requirements for inlining and puts them in a temporary inline library for use by the compilation pass.

Several `-Minline` suboptions let you determine the selection criteria for procedures to be inlined. These suboptions include:

except:func

Inlines all eligible procedures except `func`, a procedure in the source text. You can use a comma-separated list to specify multiple procedure.

[name:]func

Inlines all procedures in the source text whose name matches `func`. You can use a comma-separated list to specify multiple procedures.

[maxsize:]n

A numeric option is assumed to be a size. Procedures of size `n` or less are inlined, where `n` loosely corresponds to the number of lines in the procedure. If both `n` and `func` are specified, then procedures matching the given name(s) or meeting the size requirements are inlined.

reshape

Fortran subprograms with array arguments are not inlined by default if the array shape does not match the shape in the caller. Use this option to override the default.

smallsize:n

Always inline procedures of size smaller than `n` regardless of other size limits.

totalsize:n

Stop inlining in a procedure when the procedure's total size with inlining reaches the `n` specified.

[lib:]file.ext

Instructs the inliner to inline the procedures within the library file `file.ext`. If no inline library is specified, procedures are extracted from a temporary library created during an extract prepass.



Tip Create the library file using the `-Mextract` option.

If you specify both a procedure name and a `maxsize n`, the compiler inlines procedures that match the procedure name *or* have `n` or fewer statements.

If a name is used without a keyword, then a name with a period is assumed to be an inline library and a name without a period is assumed to be a procedure name. If a number is used without a keyword, the number is assumed to be a size.

Inlining can be disabled with `-Mnoinline`.

In the following example, the compiler inlines procedures with fewer than approximately 100 statements in the source file `myprog.f` and writes the executable code in the default output file `a.out`.

```
$ nvfortran -Minline=maxsize:100 myprog.f
```

4.3. Using an Inline Library

If you specify one or more inline libraries on the command line with the `-Minline` option, the compiler does not perform an initial extract pass. The compiler selects functions to inline from the specified inline library. If you also specify a size or function name, all functions in the inline library meeting the selection criteria are selected for inline expansion at points in the source text where they are called.

If you do not specify a function name or a size limitation for the `-Minline` option, the compiler inlines every function in the inline library that matches a function in the source text.

In the following example, the compiler inlines the function `proc` from the inline library `lib.il` and writes the executable code in the default output file `a.out`.

```
$ nvfortran -Minline=name:proc,lib:lib.il myprog.f
```

The following command line is equivalent to the preceding line, with the exception that in the following example does not use the keywords `name:` and `lib:`. You typically use keywords to avoid name conflicts when you use an inline library name that does not contain a period. Otherwise, without the keywords, a period informs the compiler that the file on the command line is an inline library.

```
$ nvfortran -Minline=proc,lib.il myprog.f
```

4.4. Creating an Inline Library

You can create or update an inline library using the `-Mextract` command-line option. If you do not specify selection criteria with the `-Mextract` option, the compiler attempts to extract all procedures.

Several `-Mextract` options let you determine the selection criteria for creating or updating an inline library. These selection criteria include:

func

Extracts the procedure `func`. you can use a comma-separated list to specify multiple procedures.

[name:]func

Extracts the procedure whose name matches `func`, a procedure in the source text.

[size:]n

Limits the size of the extracted procedures to those with a statement count less than or equal to `n`, the specified size.



The size `n` may not exactly equal the number of statements in a selected procedure; the size parameter is merely a rough gauge.

[lib:]ext.lib

Stores the extracted information in the library directory `ext.lib`.

If no inline library is specified, procedures are extracted to a temporary library created during an extract prepass for use during the compilation stage.

When you use the `-Mextract` option, only the extract phase is performed; the compile and link phases are not performed. The output of an extract pass is a library of procedures available for inlining. This output is placed in the inline library file specified on the command line with the `-o` filename specification. If the library file exists, new information is appended to it. If the file does not exist, it is created. You can use a command similar to the following:

```
$ nvfortran -Mextract=lib:lib.il myfunc.f
```

You can use the `-Minline` option with the `-Mextract` option. In this case, the extracted library of procedures can have other procedures inlined into the library. Using both options enables you to obtain more than one level of inlining. In this situation, if you do not specify a library with the `-Minline` option, the inline process consists of two extract passes. The first pass is a hidden pass implied by the `-Minline` option, during which the compiler extracts procedures and places them into a temporary library. The second pass uses the results of the first pass but puts its results into the library that you specify with the `-o` option.

4.4.1. Working with Inline Libraries

An inline library is implemented as a directory with each inline function in the library stored as a file using an encoded form of the inlinable function.

A special file named `TOC` in the inline library directory serves as a table of contents for the inline library. This is a printable, ASCII file which you can examine to locate information about the library contents, such as names and sizes of functions, the source file from which they were extracted, the version number of the extractor which created the entry, and so on.

Libraries and their elements can be manipulated using ordinary system commands.

- ▶ Inline libraries can be copied or renamed.
- ▶ Elements of libraries can be deleted or copied from one library to another.
- ▶ The `ls` or `dir` command can be used to determine the last-change date of a library entry.

4.4.2. Dependencies

When a library is created or updated using one of the NVIDIA HPC compilers, the last-change date of the library directory is updated. This allows a library to be listed as a dependence in a makefile and ensures that the necessary compilations are performed when a library is changed.

4.4.3. Updating Inline Libraries – Makefiles

If you use inline libraries you must be certain that they remain up-to-date with the source files into which they are inlined. One way to assure inline libraries are updated is to include them in a makefile.

The makefile fragment in the following example assumes the file `utils.f` contains a number of small functions used in the files `parser.f` and `alloc.f`.

This portion of the makefile:

- ▶ Maintains the inline library `utils.il`.
- ▶ Updates the library whenever you change `utils.f` or one of the include files it uses.
- ▶ Compiles `parser.f` and `alloc.f` whenever you update the library.

Sample Makefile

```
SRC = mydir
FC = nvfortran
FFLAGS = -O2
main.o: $(SRC)/main.f $(SRC)/global.h
    $(FC) $(FFLAGS) -c $(SRC)/main.f
utils.o: $(SRC)/utils.f $(SRC)/global.h $(SRC)/utils.h
    $(FC) $(FFLAGS) -c $(SRC)/utils.f
utils.il: $(SRC)/utils.f $(SRC)/global.h $(SRC)/utils.h
    $(FC) $(FFLAGS) -Mextract=15 -o utils.il $(SRC)/utils.f
parser.o: $(SRC)/parser.f $(SRC)/global.h utils.il
    $(FC) $(FFLAGS) -Minline=utils.il -c $(SRC)/parser.f
alloc.o: $(SRC)/alloc.f $(SRC)/global.h utils.il
    $(FC) $(FFLAGS) -Minline=utils.il -c $(SRC)/alloc.f
myprog: main.o utils.o parser.o alloc.o
    $(FC) -o myprog main.o utils.o parser.o alloc.o
```

4.5. Error Detection during Inlining

You can specify the `-Minfo=inline` option to request inlining information from the compiler when you invoke the inliner. For example:

```
$ nvfortran -Minline=mylib.il -Minfo=inline myext.f
```

4.6. Examples

Assume the program `dhry` consists of a single source file `dhry.f`. The following command line builds an executable file for `dhry` in which `proc7` is inlined wherever it is called:

```
$ nvfortran dhry.f -Minline=proc7
```

The following command lines build an executable file for `dhry` in which `proc7` plus any functions of approximately 10 or fewer statements are inlined (one level only).



The specified functions are inlined only if they are previously placed in the inline library, `temp.il`, during the extract phase.

```
$ nvfortran dhry.f -Mextract=lib:temp.il
$ nvfortran dhry.f -Minline=10,proc7,temp.il
```

Using the same source file `dhry.f`, the following example builds an executable for `dhry` in which all functions of roughly ten or fewer statements are inlined. Two levels of inlining are performed. This means that if function A calls function B, and B calls C, and both B and C are inlinable, then the version of B which is inlined into A will have had C inlined into it.

```
$ nvfortran dhry.f -Minline=maxsize:10
```

4.7. Restrictions on Inlining

The following Fortran subprograms cannot be extracted:

- ▶ Main or BLOCK DATA programs.
- ▶ Subprograms containing alternate return, assigned GO TO, DATA, SAVE, or EQUIVALENCE statements.
- ▶ Subprograms containing FORMAT statements.
- ▶ Subprograms containing multiple entries.

A Fortran subprogram is not inlined if any of the following applies:

- ▶ It is referenced in a statement function.
- ▶ A common block mismatch exists; in other words, the caller must contain all common blocks specified in the callee, and elements of the common blocks must agree in name, order, and type (except that the caller's common block can have additional members appended to the end of the common block).
- ▶ An argument mismatch exists; in other words, the number and type (size) of actual and formal parameters must be equal.
- ▶ A name clash exists, such as a call to subroutine `xyz` in the extracted subprogram and a variable named `xyz` in the caller.

The following types of C and C++ functions cannot be inlined:

- ▶ Functions which accept a variable number of arguments

Certain C/C++ functions can only be inlined into the file that contains their definition:

- ▶ Static functions
- ▶ Functions which call a static function
- ▶ Functions which reference a static variable

Chapter 5.

USING OPENMP FOR MULTICORE CPUS

The NVFORTRAN compiler supports the OpenMP Fortran Application Program Interface for multicore CPUs. The NVC++ and NVC compilers support the OpenMP C++ and C Application Program Interface for multicore CPUs.

OpenMP is a specification for a set of compiler directives, an applications programming interface (API), and a set of environment variables that can be used to specify parallel execution in Fortran, C++ and C programs.

This section provides information on OpenMP as currently supported on multicore CPUs by the NVIDIA HPC compilers, which includes support for most features in the OpenMP 4.5 specification.

Use the `-mp` compiler switch to enable processing of the OpenMP directives and pragmas listed in this section.



OpenMP is not currently supported on NVIDIA GPUs. All **target** regions in an OpenMP 4.5 program are executed using the multicore CPU host as the target.



When using `nvc++` on Linux, the GNU STL is thread-safe to the extent listed in the GNU documentation as required by the C++11 standard. If an STL thread-safe issue is suspected, the suspect code can be run sequentially inside of an OpenMP region using **`#pragma omp critical`** sections.

5.1. OpenMP Overview

OpenMP 4.5

The NVIDIA HPC Fortran, C++ and C compilers can compile most OpenMP 4.5 programs for parallel execution across all the cores of a multicore CPU or server. **target** regions are implemented with default support for the multicore host as the target, and **parallel** and **distribute** loops are parallelized across all OpenMP threads.

Current known limitations include:

- ▶ The **simd** construct can be used to provide tuning hints only; the **simd** construct's **private**, **lastprivate**, **reduction**, and **collapse** clauses are processed and supported, but the **simd** construct is only used as a hint to the auto-vectorizer.
- ▶ The **declare simd** construct is ignored.
- ▶ The **ordered** construct's **simd** clause is ignored.
- ▶ The **task** construct's **depend** and **priority** clauses are not supported.
- ▶ The loop construct's **linear**, **schedule**, and **ordered(n)** clauses are not supported.
- ▶ The **declare reduction** directive is not supported.
- ▶ The **reduction** clause does not accept pointer or reference types.

5.1.1. OpenMP Shared-Memory Parallel Programming Model

The OpenMP shared-memory programming model for multicore CPUs is a collection of compiler directives or pragmas, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran, C++ and C programs.

Fortran directives, C++ and C pragmas

Allow users to mark sections of code that can be executed in parallel when the code is compiled using the `-mp` command-line option.

When this option is not present, the compiler ignores these directives and pragmas.

Fixed-form Fortran OpenMP directives begin with `!$OMP`, `C$OMP`, or `*$OMP`, beginning in column 1. Free-form Fortran OpenMP directives begin with `!$OMP`. OpenMP pragmas for C++ and C begin with `#pragma omp`. This format allows the user to have a single source code file for use with or without the `-mp` option, as these lines are then merely viewed as comments when `-mp` is not present.

These directives and pragmas allow the user to create task, loop, and parallel section work-sharing constructs and synchronization constructs. They also allow the user to define how data is shared or copied between parallel threads of execution.



The data environment is controlled either by using clauses on the directives or pragmas, or with additional directives or pragmas.

Runtime library routines

Are available to query the parallel runtime environment, for example to determine how many threads are participating in execution of a parallel region.

Environment variables

Are available to control the execution behavior of parallel programs. For more information, see the [OpenMP website, http://www.openmp.org](http://www.openmp.org).

Macro substitution

C++ and C pragmas are subject to macro replacement after `#pragma omp`.

5.1.2. Terminology

There are a number of OpenMP terms for which it is useful to have common definitions.

Thread

An execution entity with a stack and associated static memory, called *threadprivate memory*.

- ▶ An OpenMP thread is a thread that is managed by the OpenMP runtime system.
- ▶ A thread-safe routine is a routine that performs the intended function even when executed concurrently, that is, by more than one thread.

Region

All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation.

Regions are *nested* if one region is (dynamically) enclosed by another region, that is, a region is encountered during the execution of another region. NVIDIA supports both lexically and non-lexically nested parallel regions.

Parallel region

There is a distinction between a parallel region and an active parallel region. A parallel region can be either inactive or active.

- ▶ An inactive parallel region is executed by a single thread.
- ▶ An active parallel region is a parallel region that is executed by a team consisting of more than one thread.



The definition of an active parallel region changed between OpenMP 2.5 and OpenMP 3.1. In OpenMP 2.5, the definition was a parallel region whose IF clause evaluates to true. To examine the significance of this change, look at the following example:

```
program test
  logical omp_in_parallel

!$omp parallel
  print *, omp_in_parallel()
!$omp end parallel

  stop
end
```

Suppose we run this program with OMP_NUM_THREADS set to one. In OpenMP 2.5, this program yields T while in OpenMP 3.1, the program yields F. In OpenMP 3.1, execution is not occurring by more than one thread. Therefore, change in this definition may mean previous programs require modification.

Task

A specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct.

Target region

A Region that operates within a distinct data environment and is specified in OpenMP 4.5 so that it can be executed either locally on the host or on a separate accelerator device. **Note:** The NVIDIA HPC Compilers map all Target regions to the multicore host

5.1.3. OpenMP Example

Look at the following simple OpenMP example involving loops.

OpenMP Loop Example

```

PROGRAM MAIN
  INTEGER I, N, OMP_GET_THREAD_NUM
  REAL*8 V(1000), GSUM, LSUM
  GSUM = 0.0D0
  N = 1000
  DO I = 1, N
    V(I) = DBLE(I)
  ENDDO

!$OMP PARALLEL PRIVATE(I, LSUM) SHARED(V, GSUM, N)
  LSUM = 0.0D0
!$OMP DO
  DO I = 1, N
    LSUM = LSUM + V(I)
  ENDDO
!$OMP END DO
!$OMP CRITICAL
  print *, "Thread ", OMP_GET_THREAD_NUM(), " local sum: ", LSUM
  GSUM = GSUM + LSUM
!$OMP END CRITICAL
!$OMP END PARALLEL

  PRINT *, "Global Sum: ", GSUM
  STOP
END

```

If you execute this example with the environment variable OMP_NUM_THREADS set to 4, then the output looks similar to this:

```

Thread      0 local sum:    31375.000000000000
Thread      1 local sum:    93875.000000000000
Thread      2 local sum:    156375.000000000000
Thread      3 local sum:    218875.000000000000
Global Sum:  500500.000000000000
FORTRAN STOP

```

5.2. Task Overview

Every part of an OpenMP program is part of a task. A task, whose execution can be performed immediately or delayed, has these characteristics:

- ▶ Code to execute
- ▶ A data environment – that is, it owns its data
- ▶ An assigned thread that executes the code and uses the data.

There are two activities associated with tasks: packaging and execution.

- ▶ **Packaging:** Each encountering thread packages a new instance of a task – code and data.
- ▶ **Execution:** Some thread in the team executes the task at some later time.

In the following sections, we use this terminology:

Task

The package of code and instructions for allocating data created when a thread encounters a task construct. A task can be implicit or explicit.

- ▶ An explicit task is a task generated when a task construct is encountered during execution.
- ▶ An implicit task is a task generated by the implicit parallel region or generated when a parallel construct is encountered during execution.

Task construct

A task directive or pragma plus a structured block.

Task region

The dynamic sequence of instructions produced by the execution of a task by a thread.

5.3. Fortran Parallelization Directives

Parallelization directives are comments in a program that are interpreted by the NVIDIA Fortran compiler when the option `-mp` is specified on the command line. The form of a parallelization directive is:

```
sentinel directive_name [clauses]
```

The *sentinel* must comply with these rules:

- ▶ Be one of these: `!$OMP`, `C$OMP`, or `*$OMP`.
- ▶ Must start in column 1 (one) for free-form code.
- ▶ Must appear as a single word without embedded white space.

The *directive_name* can be any OpenMP 4.5 directive other than those listed above as limitations. The valid clauses depend on the directive and are also subject to the limitations listed above.

In addition to the sentinel rules, the directive must also comply with these rules:

- ▶ Standard Fortran syntax restrictions, such as line length, case insensitivity, and so on, apply to the directive line.
- ▶ Initial directive lines must have a space or zero in column six.
- ▶ Continuation directive lines must have a character other than a space or a zero in column six.
- ▶ Directives which are presented in pairs must be used in pairs.

Clauses associated with directives have these characteristics:

- ▶ The order in which clauses appear in the parallelization directives is not significant.

- ▶ Commas separate clauses within the directives, but commas are not allowed between the directive name and the first clause.
- ▶ Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause.

5.4. C++ and C Parallelization Pragma

Parallelization pragmas are `#pragma` statements in a C++ or C program that are interpreted by the NVIDIA HPC C++ and C compilers when the option `-mp` is specified on the command line. The form of a parallelization pragma is:

```
#pragma omp    pragma_name    [clauses]
```

The format for pragmas include these rules:

- ▶ The pragmas follow the conventions of the C++ and C rules.
- ▶ Whitespace can appear before and after the `#`.
- ▶ Preprocessing tokens following the `#pragma omp` are subject to macro replacement.
- ▶ The order in which clauses appear in the parallelization pragmas is not significant.
- ▶ Spaces separate clauses within the pragmas.
- ▶ Clauses on pragmas may be repeated as needed subject to the restrictions listed in the description of each clause.

For the purposes of the OpenMP pragmas, a C++ or C structured block is defined to be a statement or compound statement (a sequence of statements beginning with `{` and ending with `}`) that has a single entry and a single exit. No statement or compound statement is a C++ or C structured block if there is a jump into or out of that statement.

5.5. Directive and Pragma Recognition

The compiler option `-mp` enables recognition of the parallelization directives and pragmas.

The use of this option also implies:

-Miomutex

For directives, critical sections are generated around Fortran I/O statements.

For pragmas, calls to C++ or C I/O library functions are system-dependent and are not necessarily guaranteed to be thread-safe. I/O library calls within C++ or C parallel regions should be protected by critical regions to ensure they function correctly on all systems.

5.6. Runtime Library Routines

User-callable functions are available to the programmer to query and alter the parallel execution environment.

Any C++ or C program unit that invokes these functions should include the statement `#include <omp.h>`. The `omp.h` header file contains definitions for each of the C++ and C library routines and the required type definitions. For example, to use the `omp_get_num_threads` function, use this syntax:

```
#include <omp.h>
int omp_get_num_threads(void);
```

The following table summarizes a few of the commonly-used runtime library calls.



The Fortran call is shown first followed by the equivalent C++ or C call.

Table 11 Frequently-used OpenMP Runtime Library Routines Summary

Frequently-used Runtime Library Routines with Examples	
omp_get_num_threads Returns the number of threads in the team executing the parallel region from which it is called. When called from a serial region, this function returns 1. A nested parallel region is the same as a single parallel region. By default, the value returned by this function is equal to the value of the environment variable <code>OMP_NUM_THREADS</code> or to the value set by the last previous call to omp_set_num_threads() .	
Fortran	<code>integer function omp_get_num_threads()</code>
C++ or C	<code>int omp_get_num_threads(void);</code>
omp_set_num_threads Sets the number of threads to use for the next parallel region. This subroutine or function can only be called from a serial region of code. If it is called from within a parallel region, or from within a subroutine or function that is called from within a parallel region, the results are undefined. Further, this subroutine or function has precedence over the <code>OMP_NUM_THREADS</code> environment variable.	
Fortran	<code>subroutine omp_set_num_threads(scalar_integer_exp)</code>
C++ or C	<code>void omp_set_num_threads(int num_threads);</code>
omp_get_thread_num Returns the thread number within the team. The thread number lies between 0 and omp_get_num_threads()-1 . When called from a serial region, this function returns 0. A nested parallel region is the same as a single parallel region.	
Fortran	<code>integer function omp_get_thread_num()</code>
C++ or C	<code>int omp_get_thread_num(void);</code>
omp_get_ancestor_thread_num Returns, for a given nested level of the current thread, the thread number of the ancestor.	
Fortran	<code>integer function omp_get_ancestor_thread_num(level) integer level</code>
C++ or C	<code>int omp_get_ancestor_thread_num(int level);</code>
omp_get_active_level	

Frequently-used Runtime Library Routines with Examples	
Returns the number of enclosing active parallel regions enclosing the task that contains the call.	
Fortran	<code>integer function omp_get_active_level()</code>
C++ or C	<code>int omp_get_active_level(void);</code>
omp_get_level	
Returns the number of parallel regions enclosing the task that contains the call.	
Fortran	<code>integer function omp_get_level()</code>
C++ or C	<code>int omp_get_level(void);</code>
omp_get_max_threads	
Returns the maximum value that can be returned by calls to omp_get_num_threads() . If omp_set_num_threads() is used to change the number of processors, subsequent calls to omp_get_max_threads() return the new value. Further, this function returns the maximum value whether executing from a parallel or serial region of code.	
Fortran	<code>integer function omp_get_max_threads()</code>
C++ or C	<code>int omp_get_max_threads(void);</code>
omp_get_num_procs	
Returns the number of processors that are available to the program	
Fortran	<code>integer function omp_get_num_procs()</code>
C++ or C	<code>int omp_get_num_procs(void);</code>
omp_get_stack_size	
Returns the value of the OpenMP internal control variable that specifies the size that is used to create a stack for a newly created thread. This value may not be the size of the stack of the current thread.	
Fortran	<pre>!omp_get_stack_size interface function omp_get_stack_size () use omp_lib_kinds integer (kind=OMP_STACK_SIZE_KIND) :: omp_get_stack_size end function omp_get_stack_size end interface</pre>
C++ or C	<code>size_t omp_get_stack_size(void);</code>
omp_set_stack_size	
Changes the value of the OpenMP internal control variable that specifies the size to be used to create a stack for a newly created thread. The integer argument specifies the stack size in kilobytes. The size of the stack of the current thread cannot be changed. In the NVIDIA implementation, all OpenMP or auto-parallelization threads are created	

Frequently-used Runtime Library Routines with Examples	
just prior to the first parallel region; therefore, only calls to <code>omp_set_stack_size()</code> that occur prior to the first region have an effect.	
Fortran	<code>subroutine omp_set_stack_size(integer(KIND=OMP_STACK_SIZE_KIND))</code>
C++ or C	<code>void omp_set_stack_size(size_t stack_size);</code>
<code>omp_get_team_size</code>	
Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs.	
Fortran	<code>integer function omp_get_team_size (level) integer level</code>
C++ or C	<code>int omp_get_team_size(int level);</code>
<code>omp_in_final</code>	
Returns whether or not the call is within a final task. Returns <code>.TRUE.</code> for directives and non-zero for pragmas if called from within a final task region.	
Fortran	<code>integer function omp_in_final()</code>
C++ or C	<code>int omp_in_final(void);</code>
<code>omp_in_parallel</code>	
Returns whether or not the call is within a parallel region. Returns <code>.TRUE.</code> for directives and non-zero for pragmas if called from within a parallel region and <code>.FALSE.</code> for directives and zero for pragmas if called outside of a parallel region. When called from within a parallel region that is serialized, for example in the presence of an IF clause evaluating <code>.FALSE.</code> for directives and zero for pragmas, the function returns <code>.FALSE.</code> for directives and zero for pragmas.	
Fortran	<code>logical function omp_in_parallel()</code>
C++ or C	<code>int omp_in_parallel(void);</code>
<code>omp_set_dynamic</code>	
Allows automatic dynamic adjustment of the number of threads used for execution of parallel regions. This function is recognized, but currently has no effect.	
Fortran	<code>subroutine omp_set_dynamic(scalar_logical_exp)</code>
C++ or C	<code>void omp_set_dynamic(int dynamic_threads);</code>
<code>omp_get_dynamic</code>	
Allows the user to query whether automatic dynamic adjustment of the number of threads used for execution of parallel regions is enabled. This function is recognized, but currently always returns <code>.FALSE.</code> for directives and zero for pragmas.	
Fortran	<code>logical function omp_get_dynamic()</code>
C++ or C	<code>void omp_get_dynamic(void);</code>
<code>omp_set_nested</code>	

Frequently-used Runtime Library Routines with Examples	
Allows enabling/disabling of nested parallel regions.	
Fortran	<pre>subroutine omp_set_nested(nested) logical nested</pre>
C++ or C	<pre>void omp_set_nested(int nested);</pre>
omp_get_nested	
Allows the user to query whether dynamic adjustment of the number of threads available for execution of parallel regions is enabled.	
Fortran	<pre>logical function omp_get_nested()</pre>
C++ or C	<pre>int omp_get_nested(void);</pre>
omp_set_schedule	
Set the value of the run_sched_var.	
Fortran	<pre>subroutine omp_set_schedule(kind, modifier) include 'omp_lib_kinds.h' integer (kind=omp_sched_kind) kind integer modifier</pre>
C++ or C	<pre>void omp_set_schedule(omp_sched_t kind, int chunk_size);</pre>
omp_get_schedule	
Retrieve the value of the run_sched_var.	
Fortran	<pre>subroutine omp_get_schedule(kind, modifier) include 'omp_lib_kinds.h' integer (kind=omp_sched_kind) kind integer modifier</pre>
C++ or C	<pre>void omp_get_schedule(omp_sched_t *kind, int *chunk_size);</pre>
omp_get_wtime	
Returns the elapsed wall clock time, in seconds, as a DOUBLE PRECISION value for directives and as a floating-point double value for pragmas.	
Times returned are per-thread times, and are not necessarily globally consistent across all threads.	
Fortran	<pre>double precision function omp_get_wtime()</pre>
C++ or C	<pre>double omp_get_wtime(void);</pre>
omp_get_wtick	
Returns the resolution of omp_get_wtime(), in seconds, as a DOUBLE PRECISION value for Fortran directives and as a floating-point double value for C++ or C pragmas.	
Fortran	<pre>double precision function omp_get_wtick()</pre>
C++ or C	<pre>double omp_get_wtick();</pre>
omp_init_lock	
Initializes a lock associated with the variable lock for use in subsequent calls to lock routines.	

Frequently-used Runtime Library Routines with Examples	
The initial state of the lock is unlocked. If the variable is already associated with a lock, it is illegal to make a call to this routine.	
Fortran	<pre> subroutine omp_init_lock(lock) include 'omp_lib_kinds.h' integer(kind=omp_lock_kind) lock </pre>
C++ or C	<pre> void omp_init_lock(omp_lock_t *lock); void omp_init_nest_lock(omp_nest_lock_t *lock); </pre>
omp_destroy_lock	
Disassociates a lock associated with the variable.	
Fortran	<pre> subroutine omp_destroy_lock(lock) include 'omp_lib_kinds.h' integer(kind=omp_lock_kind) lock </pre>
C++ or C	<pre> void omp_destroy_lock(omp_lock_t *lock); void omp_destroy_nest_lock(omp_nest_lock_t *lock); </pre>
omp_set_lock	
Causes the calling thread to wait until the specified lock is available.	
The thread gains ownership of the lock when it is available. If the variable is not already associated with a lock, it is illegal to make a call to this routine.	
Fortran	<pre> subroutine omp_set_lock(lock) include 'omp_lib_kinds.h' integer(kind=omp_lock_kind) lock </pre>
C++ or C	<pre> void omp_set_lock(omp_lock_t *lock); void omp_set_nest_lock(omp_nest_lock_t *lock); </pre>
omp_unset_lock	
Causes the calling thread to release ownership of the lock associated with integer_var.	
If the variable is not already associated with a lock, it is illegal to make a call to this routine.	
Fortran	<pre> subroutine omp_unset_lock(lock) include 'omp_lib_kinds.h' integer(kind=omp_lock_kind) lock </pre>
C++ or C	<pre> void omp_unset_lock(omp_lock_t *lock); void omp_unset_nest_lock(omp_nest_lock_t *lock); </pre>
omp_test_lock	
Causes the calling thread to try to gain ownership of the lock associated with the variable.	
The function returns <code>.TRUE.</code> for directives and non-zero for pragmas if the thread gains ownership of the lock; otherwise, it returns <code>.FALSE.</code> for directives and zero for pragmas.	
If the variable is not already associated with a lock, it is illegal to make a call to this routine.	
Fortran	<pre> logical function omp_test_lock(lock) include 'omp_lib_kinds.h' integer(kind=omp_lock_kind) lock </pre>
C++ or C	<pre> int omp_test_lock(omp_lock_t *lock); int omp_test_nest_lock(omp_nest_lock_t *lock); </pre>

5.7. OpenMP Environment Variables

You can use OpenMP environment variables to control the behavior of OpenMP programs. These environment variables allow you to set and pass information that can alter the behavior of directives and pragmas.

The following summary table is a quick reference for the OpenMP environment variables supported by the NVIDIA HPC Compilers. Note that the text values are case insensitive, e.g. "TRUE" is the same as "true". Possible values are listed in all upper case for readability

Table 12

Environment Variable	Default	Description
OMP_DYNAMIC	TRUE	Enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.
OMP_MAX_ACTIVE_LEVELS	1	Specifies the maximum number of nested parallel regions. Nested parallelism is not currently supported by the NVIDIA HPC compilers so this environment variable is effectively ignored, and calls to <code>omp_set_max_active_level()</code> have no effect.
OMP_NUM_THREADS	# of physical CPU cores	Specifies the number of threads to use during execution of parallel regions. For example, <code>OMP_NUM_THREADS=4</code> will limit OpenMP executables to use 4 threads. Note that a setting such as <code>OMP_NUM_THREADS=4,2</code> is valid, but because nested parallelism is not currently supported any value beyond the first position is ignored and in this case the program will execute as if the setting is <code>OMP_NUM_THREADS=4</code> .
OMP_SCHEDULE	STATIC with chunk size of 0	Specifies the type of iteration scheduling and optionally the chunk size to use for <code>omp do</code> , <code>omp for</code> and <code>omp parallel</code> loops that include the runtime schedule clause. The supported schedule types are STATIC, DYNAMIC, GUIDED and AUTO.
OMP_PROC_BIND	FALSE	Specifies whether executing threads should be bound to a core during execution. Allowable values are TRUE, FALSE, CLOSE, SPREAD and MASTER. Multiple comma-separated values CLOSE, SPREAD and MASTER are allowed, specifying thread affinity policy on the corresponding level of parallelism. The current implementation ignores all values at levels greater than 1.
OMP_STACKSIZE	System default	Overrides the default stack size for a newly created thread.
OMP_THREAD_LIMIT	2^{31}	Specifies the absolute maximum number of threads that can be used in a program.
OMP_WAIT_POLICY	PASSIVE	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are ACTIVE and PASSIVE.

Environment Variable	Default	Description
OMP_PLACES		A list of places to which threads will bind. Multiple formats are supported. In the simplest cases the value can be an abstract name that describes a set of places (e.g. THREADS, CORES or SOCKETS) or an explicit list of places described by non-negative numbers (e.g. OMP_PLACES={0,1,2,3}). A detailed format permits specifying individual places as intervals in the form "lower-bound : length : stride". A full description of the detailed format is given in section 4.5, OMP_PLACES, of the OpenMP 4.5 specification.

Chapter 6.

USING AN NVIDIA GPU

An NVIDIA GPU can be used as an accelerator to which a CPU can offload data and executable kernels to perform compute-intensive calculations. This section gives an overview of options for programming NVIDIA GPUs, and describes in detail how OpenACC compiler directives can be used to specify regions of code in Fortran, C and C++ programs that can be offloaded from a *host* CPU to an NVIDIA GPU.

6.1. Overview

With the NVIDIA HPC Compilers you can program NVIDIA GPUs using certain standard language constructs, OpenACC directives, or CUDA Fortran language extensions. GPU programming with standard language constructs or directives allows you to create high-level GPU-accelerated programs without the need to explicitly initialize the GPU, manage data or program transfers between the host and GPU, or initiate GPU startup and shutdown. Rather, all of these details are implicit in the programming model and are managed by the NVIDIA HPC SDK Fortran, C++ and C compilers. GPU programming with CUDA extensions gives you access to all NVIDIA GPU features and full control over data management and offloading of compute-intensive loops and kernels.

The NVCC++ compiler supports automatic offload of C++17 Parallel Algorithms invocations to NVIDIA GPUs under control of the `-stdpar` compiler option. See the Blog post *Accelerating Standard C++ with GPUs* for details on using this feature. The NVFORTRAN compiler supports automatic offload to NVIDIA GPUs of certain Fortran array intrinsics and patterns of array syntax, including use of Volta and Ampere architecture Tensor Cores for appropriate intrinsics. See the Blog post *Bringing Tensor Cores to Standard Fortran* for details on using this feature.

The NVFORTRAN compiler supports CUDA programming in Fortran. See the *NVIDIA CUDA Fortran Programming Guide* for complete details on how to use CUDA Fortran. The NVCC compiler supports CUDA programming in C and C++ in combination with a host C++ compiler on your system. See the *CUDA C++ Programming Guide* for an introduction and overview of how to use NVCC and CUDA C++.

The NVFORTRAN, NVCC++ and NVCC compilers all support directive-based programming of NVIDIA GPUs using OpenACC. OpenACC is an accelerator

programming model that is portable across operating systems and various host CPUs and types of accelerators, including both NVIDIA GPUs and multicore CPUs. OpenACC directives allow a programmer to migrate applications incrementally to accelerator targets using standards-compliant Fortran, C++ or C that remains completely portable to other compilers and systems. It allows the programmer to augment information available to the compilers, including specification of data local to an accelerator region, guidance on mapping of loops onto an accelerator, and similar performance-related details.

In user-directed accelerator programming the user specifies the regions of a host program to be targeted for offloading to an accelerator. The bulk of a user's program, as well as regions containing constructs that are not supported on the targeted accelerator, are executed on the host. The remainder of this chapter gives an overview of directive-based OpenACC programming. For complete details on using OpenACC with NVIDIA GPUs, see the *OpenACC Getting Started Guide*.

6.2. Terminology

Clear and consistent terminology is important in describing any programming model. This section provides definitions of the terms required for you to effectively use this section and the associated programming model.

Accelerator

a parallel processor, such as a GPU or a CPU running in multicore mode, to which a CPU can offload data and executable kernels to perform compute-intensive calculations.

Compute intensity

for a given loop, region, or program unit, the ratio of the number of arithmetic operations performed on computed data divided by the number of memory transfers required to move that data between two levels of a memory hierarchy.

Compute region

a structured block defined by an OpenACC compute construct. A *compute construct* is a structured block containing loops which are compiled for the accelerator. A compute region may require device memory to be allocated and data to be copied from host to device upon region entry, and data to be copied from device to host memory and device memory deallocated upon exit. The dynamic range of a compute construct, including any code in procedures called from within the construct, is the compute region. In this release, compute regions may not contain other compute regions or data regions.

Construct

a structured block identified by the programmer or implicitly defined by the language. Certain actions may occur when program execution reaches the start and end of a construct, such as device memory allocation or data movement between the host and device memory. Loops in a compute construct are targeted for execution on the accelerator. The dynamic range of a construct including any code in procedures called from within the construct, is called a *region*.

CUDA

stands for Compute Unified Device Architecture; CUDA C++ and Fortran language extensions and API calls can be used to explicitly control and program an NVIDIA GPU.

Data region

a region defined by an OpenACC data construct, or an implicit data region for a function or subroutine containing OpenACC directives. Data regions typically require device memory to be allocated and data to be copied from host to device memory upon entry, and data to be copied from device to host memory and device memory deallocated upon exit. Data regions may contain other data regions and compute regions.

Device

a general reference to any type of accelerator.

Device memory

memory attached to an accelerator which is physically separate from the host memory.

Directive

in C, a `#pragma`, or in Fortran, a specially formatted comment statement that is interpreted by a compiler to augment information about or specify the behavior of the program.

DMA

Direct Memory Access, a method to move data between physically separate memories; this is typically performed by a DMA engine, separate from the host CPU, that can access the host physical memory as well as an IO device or GPU physical memory.

GPU

a Graphics Processing Unit; one type of accelerator device.

Host

the main CPU that in this context has an attached accelerator device. The host CPU controls the program regions and data loaded into and executed on the device.

Loop trip count

the number of times a particular loop executes.

OpenACC

a parallel programming standard describing a set of compiler directives which can be applied to standard Fortran, C++ and C to specify regions of code for offloading from a host CPU to an attached accelerator.

Private data

with respect to an iterative loop, data which is used only during a particular loop iteration. With respect to a more general region of code, data which is used within the region but is not initialized prior to the region and is re-initialized prior to any use after the region.

Region

the dynamic range of a construct, including any procedures invoked from within the construct.

Structured block

in C++ or C, an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom.

Vector operation

a single operation or sequence of operations applied uniformly to each element of an array.

Visible device copy

a copy of a variable, array, or subarray allocated in device memory, that is visible to the program unit being compiled.

6.3. Execution Model

The execution model targeted by the NVIDIA HPC Compilers is host-directed execution with an attached accelerator device, such as a GPU. The bulk of a user application executes on the host. Compute intensive regions are offloaded to the accelerator device under control of the host. The accelerator device executes kernels, which may be as simple as a tightly-nested loop, or as complex as a subroutine, depending on the accelerator hardware.

6.3.1. Host Functions

Even in accelerator-targeted regions, the host must orchestrate the execution; it

- ▶ allocates memory on the accelerator device
- ▶ initiates data transfer
- ▶ sends the kernel code to the accelerator
- ▶ passes kernel arguments
- ▶ queues the kernel
- ▶ waits for completion
- ▶ transfers results back to the host
- ▶ deallocates memory



In most cases, the host can queue a sequence of kernels to be executed on the device, one after the other.

6.3.2. Levels of Parallelism

OpenACC supports three levels of parallelism:

- ▶ an outer *doall* (fully parallel) loop level
- ▶ a *workgroup* or *threadblock* (worker parallel) loop level
- ▶ an inner *synchronous* (SIMD or vector) loop level

Each level can be multidimensional with 2 or 3 dimensions, but the domain must be strictly rectangular. The *synchronous* level may not be fully implemented with SIMD or vector operations, so explicit synchronization is supported and required across this level. No synchronization is supported between parallel threads across the *doall* level.

The OpenACC execution model on the device side exposes these levels of parallelism and the programmer is required to understand the difference between, for example, a fully parallel loop and a loop that is vectorizable but requires synchronization

across iterations. All fully parallel loops can be scheduled for any of *doall*, *workgroup* or *synchronous* parallel execution, but by definition SIMD vector loops that require synchronization can only be scheduled for synchronous parallel execution.

6.4. Memory Model

The most significant difference between a *host-only* program and a *host+accelerator* program is that the memory on the accelerator can be completely separate from host memory, which is the case on most current GPUs. For example:

- ▶ The host cannot read or write accelerator memory by reference because it is not mapped into the virtual memory space of the host.
- ▶ All data movement between host memory and accelerator memory must be performed by the host through runtime library calls that explicitly move data between the separate memories.
- ▶ In general it is not valid to assume the accelerator can read or write host memory, though this is well-defined starting in the OpenACC 2.7 specification and supported for allocatable data in the NVIDIA HPC Compilers for GPUs and assumed when the accelerator target is a multicore CPU.

6.4.1. Separate Host and Accelerator Memory Considerations

The programmer must be aware of the potentially separate memories for many reasons, including but not limited to:

- ▶ Memory bandwidth between host memory and accelerator memory determines the compute intensity required to effectively accelerate a given region of code.
- ▶ Limited size of accelerator memory may prohibit offloading of regions of code that operate on very large amounts of data.

6.4.2. Accelerator Memory

On the accelerator side, current GPUs implement a weak memory model. In particular, they do not support memory coherence between threads unless those threads are parallel only at the synchronous level and the memory operations are separated by an explicit barrier. Otherwise, if one thread updates a memory location and another reads the same location, or two threads store a value to the same location, the hardware does not guarantee the results. While the results of running such a program might be inconsistent, it is not accurate to say that the results are incorrect. By definition, such programs are defined as being in error. While a compiler can detect some potential errors of this nature, it is nonetheless possible to write an accelerator region that produces inconsistent numerical results.

6.4.3. Cache Management

Some current GPUs have a software-managed cache, some have hardware-managed caches, and most have hardware caches that can be used only in certain situations and

are limited to read-only data. In low-level programming models such as CUDA, it is up to the programmer to manage these caches. However, in the OpenACC programming model, the compiler manages these caches using hints from the programmer in the form of directives.

6.4.4. CUDA Unified Memory

The NVIDIA OpenACC Compilers are interoperable with CUDA Unified Memory. This feature, described in detail in [OpenACC and CUDA Unified Memory, https://www.pgroup.com/blogs/posts/openacc-unified-memory.htm](https://www.pgroup.com/blogs/posts/openacc-unified-memory.htm) is available with the x86-64, OpenPOWER and Arm Server compilers. To enable this feature, add the options `-acc -gpu=managed` to the compiler and linker command lines.

In the presence of `-acc -gpu=managed`, all Fortran, C++ and C explicit allocation statements in a program unit are replaced by equivalent "managed" data allocation calls that place the data in CUDA Unified Memory. Managed data share a single address for CPU/GPU and data movement between CPU and GPU memories is implicitly handled by the CUDA driver. Therefore, OpenACC data clauses and directives are not needed for "managed" data. They are essentially ignored, and in fact can be omitted.

When a program allocates managed memory, it allocates host pinned memory as well as device memory thus making allocate and free operations somewhat more expensive and data transfers somewhat faster. A memory pool allocator is used to mitigate the overhead of the allocate and free operations. The pool allocator is enabled by default in the presence of the `-gpu=managed` or `-gpu=pinned` compiler options.

Data movement of managed data is controlled by the NVIDIA CUDA GPU driver; whenever data is accessed on the CPU or the GPU, it could trigger a data transfer if the last time it was accessed was not on the same device. In some cases, page thrashing may occur and impact performance. An introduction to CUDA Unified Memory is available on [Parallel Forall](#).

This feature has the following limitations:

- ▶ Use of managed memory applies only to dynamically-allocated data. Static data (C static and extern variables, Fortran module, common block and save variables) and function local data is still handled by the OpenACC runtime. Dynamically allocated Fortran local variables and Fortran allocatable arrays are implicitly managed but Fortran array pointers are not.
- ▶ Given an allocatable aggregate with a member that points to local, global or static data, compiling with `-gpu=managed` and attempting to access memory through that pointer from the compute kernel will cause a failure at runtime.
- ▶ C++ virtual functions are not supported.
- ▶ The `-gpu=managed` compiler option must be used to compile the files in which variables are allocated, even if there is no OpenACC code in the file.

This feature has the following additional limitations when used with NVIDIA Kepler GPUs:

- ▶ Data motion on Kepler GPUs is achieved through fast pinned asynchronous data transfers; from the program's perspective, however, the transfers are synchronous.
- ▶ The NVIDIA HPC Compilers runtime enforces synchronous execution of kernels when `-gpu=managed` is used on a system with a Kepler GPU. This situation may result in slower performance because of the extra synchronizations and decreased overlap between CPU and GPU.
- ▶ The total amount of managed memory is limited to the amount of available device memory on Kepler GPUs.

CUDA Unified Memory Pool Allocator

Dynamic memory allocations are made using `cudaMallocManaged()`, a routine which has higher overhead than allocating non-unified memory using `cudaMalloc()`. The more calls to `cudaMallocManaged()`, the more significant the impact on performance.

To mitigate the overhead of `cudaMallocManaged()` calls, both `-gpu=managed` and `-gpu=pinned` use a CUDA Unified Memory pool allocator to minimize the number of calls to `cudaMallocManaged()`. The pool allocator is enabled by default. It can be disabled, or its behavior modified, using these environment variables:

Table 13 Pool Allocator Environment Variables

Environment Variable	Use
<code>NVCOMPILER_ACC_POOL_ALLOC</code>	Disable the pool allocator. The pool allocator is enabled by default; to disable it, set <code>NVCOMPILER_ACC_POOL_ALLOC</code> to 0.
<code>NVCOMPILER_ACC_POOL_SIZE</code>	Set the size of the pool. The default size is 1GB but other sizes (i.e., 2GB, 100MB, 500KB, etc.) can be used. The actual pool size is set such that the size is the nearest, smaller number in the Fibonacci series compared to the provided or default size. If necessary, the pool allocator will add more pools but only up to the <code>NVCOMPILER_ACC_POOL_THRESHOLD</code> value.
<code>NVCOMPILER_ACC_POOL_ALLOC_</code>	Set the maximum size for allocations. The default maximum size for allocations is 500MB but another size (i.e., 100KB, 10MB, 250MB, etc.) can be used as long as it is greater than or equal to 16B.
<code>NVCOMPILER_ACC_POOL_ALLOC_</code>	Set the minimum size for allocation blocks. The default size is 128B but other sizes can be used. The size must be greater than or equal to 16B.
<code>NVCOMPILER_ACC_POOL_THRESH</code>	Set the percentage of total device memory that the pool allocator can occupy. Values from 0 to 100 are accepted. The default value is 50, corresponding to 50% of device memory.

6.5. OpenACC Programming Model

With the emergence of GPU architectures in high performance computing, programmers want the ability to program using a familiar, high level programming model that provides both high performance and portability to a wide range of computing architectures. OpenACC emerged in 2011 as a programming model that uses high-level compiler directives to expose parallelism in the code and parallelizing compilers to build the code for a variety of parallel accelerators.

This chapter will not attempt to describe OpenACC itself. For that, please refer to the OpenACC specification on the OpenACC www.openacc.org website. Here, we will discuss differences between the OpenACC specification and its implementation by the NVIDIA HPC Compilers.

Other resources to help you with your parallel programming including video tutorials, course materials, code samples, a best practices guide and more are available on the OpenACC website.

6.5.1. Enable OpenACC Directives

NVIDIA HPC compilers enable OpenACC directives with the `-acc` and `-gpu` command line options. For more information on these options refer to [Compiling an OpenACC Program](#).

`_OPENACC` macro

The `_OPENACC` macro name is defined to have a value `yyyymm` where `yyyy` is the year and `mm` is the month designation of the version of the OpenACC directives supported by the implementation. For example, the version for November, 2017 is 201711. All OpenACC compilers define this macro when OpenACC directives are enabled.

6.5.2. OpenACC Support

The NVIDIA HPC Compilers implement most features of OpenACC 2.7 as defined in *The OpenACC Application Programming Interface*, Version 2.7, November 2018, <http://www.openacc.org>, with the exception that the following OpenACC 2.7 features are not supported:

- ▶ nested parallelism
- ▶ declare link
- ▶ enforcement of the **cache** clause restriction that all references to listed variables must lie within the region being cached
- ▶ Arrays, subarrays and composite variables in **reduction** clauses
- ▶ The **self** clause
- ▶ The **default** clause on data constructs

6.5.3. OpenACC Extensions

The NVIDIA Fortran compiler supports an extension to the `collapse` clause on the `loop` construct. The OpenACC specification defines `collapse`:

```
collapse(n)
```

NVIDIA Fortran supports the use of the identifier `force` within `collapse`:

```
collapse(force:n)
```

Using `collapse(force:n)` instructs the compiler to enforce collapsing parallel loops that are not perfectly nested.

6.6. Supported Processors and GPUs

This NVIDIA HPC Compilers release supports x86-64, OpenPOWER and Arm Server CPUs. Cross-compilation across the different families of CPUs is not supported, but you can use the `-tp=<target>` flag as documented in the man pages to specify a target processor within a family.

Use the `-acc` flag to enable OpenACC directives and the `-gpu` flag to target NVIDIA GPUs. You can then use the generated code on any supported system with CUDA installed that has a CUDA-enabled GeForce, Quadro, or Tesla card.

For more information on these flags as they relate to accelerator technology, refer to [Compiling an OpenACC Program](#).

For a complete list of supported CUDA GPUs, refer to the NVIDIA website at: http://www.nvidia.com/object/cuda_learn_products.html

6.7. CUDA Versions

The NVIDIA HPC compilers use components from NVIDIA's CUDA Toolkit to build programs for execution on an NVIDIA GPU. The NVIDIA HPC SDK installation package installs bundled CUDA Toolkit components into an HPC SDK installation sub-directory, and typically supports the last 3 released versions of the CUDA Toolkit.

An NVIDIA CUDA driver must be installed on a system with a GPU before you can run a program compiled for the GPU on that system. NVIDIA HPC SDK products do not contain CUDA device drivers. You must download and install the appropriate [CUDA Driver from NVIDIA](#). The CUDA Driver version must be at least as new as the CUDA version with which you compile your code.

The NVIDIA HPC SDK utility `nvaccelinfo` prints the driver version as its first line of output. You can use it to find out which version of the CUDA Driver is installed on your system.

The NVIDIA HPC SDK 20.9 includes components from the following versions of the CUDA Toolkit:

- CUDA 10.1

- ▶ CUDA 10.2
- ▶ CUDA 11.0

You can let the compiler choose which version of CUDA to use based on the configuration of your system, or you can force it to use a particular version using the `-gpu` command-line option.

If you do not specify a version of CUDA on the command line, the compiler uses the version of the CUDA Driver installed on the system on which you are compiling to determine which CUDA version to use. In the absence of any other information, the compiler will look for a CUDA toolchain version in the `/opt/nvidia/hpc_sdk/target/cuda` directory that matches the version of the CUDA Driver installed on the system. If a match is not found, the compiler searches for the newest CUDA toolchain version that is not newer than the CUDA Driver version. If there is no CUDA Driver installed, the compilers fall back to the default of CUDA 10.1.

You can change the compiler's default selection of CUDA version using one of the following methods:

- ▶ Use a compiler option. Add the `cudaX.Y` sub-option to `-gpu` where `X.Y` denotes the CUDA version. Using a compiler option changes the CUDA toolchain version for one invocation of the compiler. For example, to compile a C file with the CUDA 11.0 toolchain you would use:

```
nvc -acc -gpu=cuda11.0
```

- ▶ Use an rcfile variable. Add a line defining `DEFCUDAVERSION` to the `siterc` file in the installation `bin` directory or to a file named `.mynvrc` in your home directory. For example, to specify the CUDA 10.2 toolchain as the default, add the following line to one of these files:

```
set DEFCUDAVERSION=10.2;
```

Using an rcfile variable changes the CUDA toolchain version for all invocations of the compilers reading the rcfile.

When you specify a CUDA version, you can additionally instruct the compiler to use a CUDA Toolkit installation separate from the defaults bundled with the current installation of the HPC SDK compilers. While most users do not need to use any other CUDA toolchain version than those provided with the HPC SDK, situations do arise where this capability is needed. Developers working with pre-release CUDA software may occasionally need to test with a CUDA Toolkit version not formally supported in an HPC SDK release. Conversely, some developers might find a need to compile with a CUDA toolchain older than the oldest CUDA toolchain bundled with a given HPC SDK release. For these users, the NVIDIA HPC compilers can interoperate with components from a CUDA Toolkit installed outside of the HPC SDK installation directories.

NVIDIA tests extensively using the bundled versions of the CUDA components and fully supports their use. Use of CUDA Toolkit components not included with an NVIDIA install is done with your understanding that functionality differences may exist.

The ability to compile with CUDA toolchain components other than the versions installed with the HPC SDK compilers is supported on all platforms.

To use a CUDA toolkit that is not installed with an NVIDIA release, there are three options:

- ▶ Use the rcfile variable `DEFAULT_CUDA_HOME` to override the base default

```
set DEFAULT_CUDA_HOME = /opt/cuda-10.0;
```

- ▶ Set the environment variable `CUDA_HOME`

```
export CUDA_HOME=/opt/cuda-10.0
```

- ▶ Use the compiler command-line assignment `CUDA_HOME=`

```
nvfortran CUDA_HOME=/opt/cuda-10.0
```

The HPC SDK compilers use the following order of precedence when determining which version of the CUDA toolchain to use.

1. If you do not specify which CUDA version to use, the compiler picks the CUDA version from the HPC SDK installation directory that matches the version of the CUDA Driver installed on your system. If the HPC SDK installation directory does not contain a direct match, the newest version in that directory which is not newer than the CUDA driver version is used. If there is no CUDA driver installed on your system, the compiler falls back on an internal default.
2. The rcfile variable `DEFAULT_CUDA_HOME` will override the base default.
3. The environment variable `CUDA_HOME` will override all of the above defaults.
4. The environment variable `NVCOMPILER_CUDA_HOME` overrides all of the above; it is available for advanced users in case they need to override an already-defined `CUDA_HOME`.
5. A user-specified `cudaX.Y` sub-option to `-gpu` will override all of the above defaults and the corresponding version of the CUDA toolchain located in the HPC SDK installation directory will be used.
6. The compiler command-line assignment `CUDA_HOME=` will override all of the above defaults (including the `cudaX.Y` sub-option).

6.8. Compute Capability

The compilers can generate code for NVIDIA GPU compute capabilities 3.0 through 8.0. The compilers construct a default list of compute capabilities that matches the compute capabilities supported by the GPUs found on the system used in compilation. If there are no GPUs detected, the compilers select `cc35`, `cc50`, `cc60`, and `cc70`.

You can override the default by specifying one or more compute capabilities using either command-line options or an rcfile.

To change the default with a command-line option, provide a comma-separated list of compute capabilities to the `-gpu` option.

To change the default with an rcfile, set the **DEFCOMPUTECAP** value to a blank-separated list of compute capabilities in the `siterc` file located in your installation's `bin` directory:

```
set DEFCOMPUTECAP=60 70;
```

Alternatively, if you don't have permissions to change the `siterc` file, you can add the **DEFCOMPUTECAP** definition to a separate `.mynvrc` file in your home directory.

The generation of device code can be time consuming, so you may notice an increase in compile time as the number of compute capabilities increases.

6.9. Compiling an OpenACC Program

Several compiler options are applicable specifically when working with OpenACC. These options include `-acc`, `-gpu`, and `-Minfo`.

6.9.1. `-acc`

Enable OpenACC directives. The following suboptions may be used following an equals sign ("="), with multiple sub-options separated by commas:

gpu

(default) OpenACC directives are compiled for GPU execution only.

host

Compile for serial execution on the host CPU.

multicore

Compile for parallel execution on the host CPU.

legacy

Suppress warnings about deprecated NVIDIA accelerator directives.

[no]autopar

Enable [disable] loop autoparallelization within acc parallel. The default is to autoparallelize, that is, to enable loop autoparallelization.

[no]routineseq

Compile every routine for the device. The default behavior is to not treat every routine as a seq directive.

strict

Instructs the compiler to issue warnings for non-OpenACC accelerator directives.

sync

Ignore async clauses

verystRICT

Instructs the compiler to fail with an error for any non-OpenACC accelerator directive.

[no]wait

Wait for each device kernel to finish. Kernel launching is blocked by default unless the async clause is used.

Usage

The following command-line requests that OpenACC directives be enabled and that an error be issued for any non-OpenACC accelerator directive.

```
$ nvfortran -acc=verystRICT prog.f
```


6.9.2. -gpu

Used in combination with the -stdpar, -acc and -cuda flags to specify options for GPU code generation. The following suboptions may be used following an equals sign ("="), with multiple sub-options separated by commas:

autocompare

Automatically compare CPU vs GPU results at execution time: implies redundant

ccXY

Generate code for a device with compute capability X.Y. Multiple compute capabilities can be specified, and one version will be generated for each. By default, the compiler will detect the compute capability for each installed GPU. Use -help -gpu to see the valid compute capabilities for your installation.

ccall

Generate code for all compute capabilities supported by this platform and by the selected or default CUDA Toolkit.

cudaX.Y

Use CUDA X.Y Toolkit compatibility, where installed

[no]debug

Enable [disable] debug information generation in device code

deepcopy

Enable full deep copy of aggregate data structures in OpenACC; Fortran only

fastmath

Use routines from the fast math library

[no]flushz

Enable [disable] flush-to-zero mode for floating point computations on the GPU

[no]fma

Generate [do not generate] fused multiply-add instructions; default at -O3

keep

Keep the kernel files (.bin, .ptx, source)

[no]lineinfo

Enable [disable] GPU line information generation

loadcache:{L1|L2}

Choose what hardware level cache to use for global memory loads; options include the default, L1, or L2

managed

Use CUDA Managed Memory for compiler-visible allocatable data objects

maxregcount:n

Specify the maximum number of registers to use on the GPU; leaving this blank indicates no limit

pinned

Use CUDA Pinned Memory

[no]rdc

Generate [do not generate] relocatable device code.

redundant

Redundant CPU/GPU execution

safecache

Allow variable-sized array sections in cache directives; compiler assumes they fit into CUDA shared memory

[no]unroll

Enable [disable] automatic inner loop unrolling; default at `-O3`

zeroinit

Initialize allocated device memory with zero

Usage

In the following example, the compiler generates code for NVIDIA GPUs with compute capabilities 6.0 and 7.0.

```
$ nvfortran -acc -gpu=cc60,cc70 myprog.f
```

The compiler automatically invokes the necessary software tools to create the kernel code and embeds the kernels in the object file.

To link in the appropriate GPU libraries, you must link an OpenACC program with the `-acc` flag, and similarly for `-stdpar` or `-cuda`.

DWARF Debugging Formats

Use the `-g` option to enable generation of DWARF information on both the host and device; in the absence of other optimization flags, `-g` sets the optimization level to zero. If a `-O` option raises the optimization level to one or higher, only GPU line information is generated in device code even when `-g` is specified. To enforce full DWARF generation for device code at optimization levels above zero, use the `debug` sub-option to `-gpu`. Conversely, to prevent the generation of dwarf information for device code, use the `nodebug` sub-option to `-gpu`. Both `debug` and `nodebug` can be used independently of `-g`.

6.10. OpenACC for Multicore CPUs

The NVIDIA OpenACC compilers support the option `-acc=multicore`, to set the target accelerator for OpenACC programs to the host multicore CPU. This will compile OpenACC compute regions for parallel execution across the cores of the host processor or processors. The host multicore CPU will be treated as a shared-memory accelerator, so the data clauses (**copy**, **copyin**, **copyout**, **create**) will be ignored and no data copies will be executed.

By default, `-acc=multicore` will generate code that will use all the available cores of the processor. If the compute region specifies a value in the **num_gangs** clause, the minimum of the **num_gangs** value and the number of available cores will be used. At runtime, the number of cores can be limited by setting the environment variable

ACC_NUM_CORES to a constant integer value. The number of cores can also be set with the `void acc_set_num_cores(int numcores)` runtime call. If an OpenACC compute construct appears lexically within an OpenMP parallel construct, the OpenACC compute region will generate sequential code. If an OpenACC compute region appears dynamically within an OpenMP region or another OpenACC compute region, the program may generate many more threads than there are cores, and may produce poor performance.

The **ACC_BIND** environment variable is set by default with `-acc=multicore`; **ACC_BIND** has similar behavior to **MP_BIND** for OpenMP.

The `-acc=multicore` option differs from the `-acc=host` option in that `-acc=host` generates sequential host CPU code for the OpenACC compute regions.

6.11. Running an OpenACC Program

Running a GPU-accelerated OpenACC program is straightforward:

- ▶ When a program includes code for NVIDIA GPUs, the program looks for and dynamically loads the CUDA libraries. If the libraries are not available, or if they are in a different directory than they were when the program was compiled, you may need to append the appropriate library directory to your `LD_LIBRARY_PATH` environment variable on Linux.
- ▶ On Linux, when your program reaches its first OpenACC region, there may be a 0.5 to 1.5 second pause to warm up the GPU from a power-off state. You can avoid this delay by running the `nvcudainit` utility in the background, which keeps the GPU powered on.
- ▶ If you compile a program for a particular GPU compute capability, then run the program on a system without a GPU that supports that compute capability, or on a system where the target libraries are not in a directory where the runtime library can find them, the program may fail at runtime with an error message.
- ▶ If you set the environment variable `NVCOMPILER_ACC_NOTIFY` to a nonzero integer value, the runtime library prints a line to standard error every time it launches a kernel on the accelerator.

6.12. OpenACC Error Handling

The OpenACC specification provides a mechanism to allow you to intercept errors triggered during execution on a GPU and execute a specific routine in response before the program exits. For example, if an MPI process fails while allocating memory on the GPU, the application may want to call `MPI_Abort` to shut down all the other processes before the program exits. This section explains how to take advantage of this feature.

To intercept errors the application must give a callback routine to the OpenACC runtime. To provide the callback, the application calls `acc_set_error_routine` with a pointer to the callback routine.

The interface is the following, where `err_msg` contains a description of the error:

```
typedef void (*exitroutinetype) (char *err_msg);
extern void acc_set_error_routine(exitroutinetype callback_routine);
```

When the OpenACC runtime detects a runtime error, it will invoke the `callback_routine`.



This feature is not the same as error recovery. If the callback routine returns to the application, the behavior is decidedly undefined.

Let's look at this feature in more depth using an example.

Take the MPI program below and run it with two processes. Process 0 tries to allocate a large array on the GPU, then sends a message to the second process to acknowledge the success of the operation. Process 1 waits for the acknowledgment and terminates upon receiving it.

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

#define N 2147483648

int main(int argc, char **argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int ack;
    if(rank == 0) {
        float *a = (float*) malloc(sizeof(float) * N);

#pragma acc enter data create(a[0:N])
#pragma acc parallel loop independent
        for(int i = 0; i < N; i++) {
            a[i] = i * 0.5;
        }
#pragma acc exit data copyout(a[0:N])
        printf("I am process %d, I have initialized a vector of size %ld bytes on
the GPU. Sending acknowledgment to process 1.", rank, N);
        ack = 1;
        MPI_Send(&ack, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if(rank == 1) {
        MPI_Recv(&ack, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("I am process %d, I have received the acknowledgment from process 0
that data in the GPU has been initialized.\n", rank, N);
        fflush(stdout);
    }

    // do some more work

    MPI_Finalize();

    return 0;
}
```

We compile the program with:

```
$ mpicc -acc -o error_handling_mpi error_handling_mpi.c
```

If we run this program with two MPI processes, the output will look like the following:

```
$ mpirun -n 2 ./error_handling_mpi
Out of memory allocating -8589934592 bytes of device memory
total/free CUDA memory: 11995578368/11919294464
Present table dump for device[1]:
NVIDIA Tesla GPU 0, compute capability 3.7, threadid=1
...empty...
call to cuMemAlloc returned error 2: Out of memory

-----
Primary job terminated normally, but 1 process returned
a non-zero exit code.. Per user-direction, the job has been aborted.
-----

-----
mpirun detected that one or more processes exited with non-zero status,
thus causing the job to be terminated.
```

Process 0 failed while allocating memory on the GPU and terminated unexpectedly with an error. In this case `mpirun` was able to identify that one of the processes failed, so it shut down the remaining process and terminated the application. A simple two-process program like this is straightforward to debug. In a real world application though, with hundreds or thousands of processes, having a process exit prematurely may cause the application to hang indefinitely. Therefore it would be ideal to catch the failure of a process, control the termination of the other processes, and provide a useful error message.

We can use the OpenACC error handling feature to improve the previous program and correctly terminate the application in case of failure of an MPI process.

In the following sample code, we have added an error handling callback routine that will shut down the other processes if a process encounters an error while executing on the GPU. Process 0 tries to allocate a large array into the GPU and, if the operation is successful, process 0 will send an acknowledgment to process 1. Process 0 calls the OpenACC function `acc_set_error_routine` to set the function `handle_gpu_errors` as an error handling callback routine. This routine prints a message and calls `MPI_Abort` to shut down all the MPI processes. If process 0 successfully allocates the array on the GPU, process 1 will receive the acknowledgment. Otherwise, if process 0 fails, it will terminate itself and trigger the call to `handle_gpu_errors`. Process 1 is then terminated by the code executed in the callback routine.

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

#define N 2147483648

typedef void (*exitroutinetype)(char *err_msg);
extern void acc_set_error_routine(exitroutinetype callback_routine);

void handle_gpu_errors(char *err_msg) {
    printf("GPU Error: %s", err_msg);
    printf("Exiting...\n\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
    exit(-1);
}
```

```

int main(int argc, char **argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int ack;
    if(rank == 0) {
        float *a = (float*) malloc(sizeof(float) * N);

        acc_set_error_routine(&handle_gpu_errors);

#pragma acc enter data create(a[0:N])
#pragma acc parallel loop independent
        for(int i = 0; i < N; i++) {
            a[i] = i * 0.5;
        }
#pragma acc exit data copyout(a[0:N])
        printf("I am process %d, I have initialized a vector of size %ld bytes on
the GPU. Sending acknowledgment to process 1.", rank, N);
        fflush(stdout);
        ack = 1;
        MPI_Send(&ack, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if(rank == 1) {
        MPI_Recv(&ack, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("I am process %d, I have received the acknowledgment from process 0
that data in the GPU has been initialized.\n", rank, N);
        fflush(stdout);
    }

    // more work

    MPI_Finalize();

    return 0;
}

```

Again, we compile the program with:

```
$ mpicc -acc -o error_handling_mpi error_handling_mpi.c
```

We run the program with two MPI processes and obtain the output below:

```

$ mpirun -n 2 ./error_handling_mpi
Out of memory allocating -8589934592 bytes of device memory
total/free CUDA memory: 11995578368/11919294464
Present table dump for device[1]:
NVIDIA Tesla GPU 0, compute capability 3.7, threadid=1
...empty...
GPU Error: call to cuMemAlloc returned error 2: Out of memory
Exiting...

```

```

-----
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD
with errorcode 1.

```

This time the error on the GPU was intercepted by the application which managed it with the error handling callback routine. In this case the routine printed some information about the problem and called `MPI_Abort` to terminate the remaining processes and avoid any unexpected behavior from the application.

6.13. Environment Variables

This section summarizes the environment variables that NVIDIA OpenACC supports. These environment variables are user-setable environment variables that control behavior of accelerator-enabled programs at execution. These environment variables must comply with these rules:

- ▶ The names of the environment variables must be upper case.
- ▶ The values of environment variables are case insensitive and may have leading and trailing white space.
- ▶ The behavior is implementation-defined if the values of the environment variables change after the program has started, even if the program itself modifies the values.

The following table contains the environment variables that are currently supported and provides a brief description of each.

Table 14 Supported Environment Variables

Use this environment variable...	To do this...
NVCOMPILER_ACC_BUFFERSIZE	For NVIDIA CUDA devices, this defines the size of the pinned buffer used to transfer data between host and device.
NVCOMPILER_ACC_CUDA_PROFS	Set to 1 (or any positive value) to tell the runtime environment to insert an 'atexit(cuProfilerStop)' call upon exit. This behavior may be desired in the case where a profile is incomplete or where a message is issued to call cudaProfilerStop().
NVCOMPILER_ACC_DEBUG	Set to 1 to instruct the runtime to generate information about device memory allocation, data movement, kernel launches, and more. NVCOMPILER_ACC_DEBUG is designed mostly for use in debugging the runtime itself, but it may be helpful in understanding how the program interacts with the device. Expect copious amounts of output.
NVCOMPILER_ACC_DEVICE_NUM = ACC_DEVICE_NUM	Sets the default device number to use. NVCOMPILER_ACC_DEVICE_NUM overrides ACC_DEVICE_NUM. Controls the default device number to use when executing accelerator regions. The value of this environment variable must be a nonnegative integer between zero and the number of devices attached to the host.
NVCOMPILER_ACC_DEVICE_TYPE = ACC_DEVICE_TYPE	Sets the default device type to use. NVCOMPILER_ACC_DEVICE_TYPE overrides ACC_DEVICE_TYPE. Controls which accelerator device to use when executing accelerator regions, if the program has been compiled to use more than one different type of device. The value of this environment variable is implementation-defined, and in the NVIDIA implementation may be the string NVIDIA, MULTICORE or HOST
NVCOMPILER_ACC_GANGLIMIT	For NVIDIA CUDA devices, this defines the maximum number of gangs (CUDA thread blocks) that will be launched by a kernel.
NVCOMPILER_ACC_NOTIFY	Writes out a line for each kernel launch and/or data movement. When set to an integer value, the value, is used as a bit mask to print information about kernel launches (value 1), data transfers (value 2), region entry/exit (value 4), wait operations or synchronizations with the

Use this environment variable...	To do this...
	device (value 8), and device memory allocates and deallocates (value 16).
NVCOMPILER_ACC_PROFLIB	Enables 3rd party tools interface using the new profiler dynamic library interface.
NVCOMPILER_ACC_SYNCHRONOUS	Disables asynchronous launches and data movement.
NVCOMPILER_ACC_TIME	Enables a lightweight profiler to measure data movement and accelerator kernel execution time and print a summary at the end of program execution.

6.14. Profiling Accelerator Kernels

Support for Profiler/Trace Tool Interface

The NVIDIA HPC Compilers support the OpenACC Profiler/Trace Tools Interface. This is the interface used by the NVIDIA profilers to collect performance measurements of OpenACC programs.

Using NVCOMPILER_ACC_TIME

Setting the environment variable NVCOMPILER_ACC_TIME to a nonzero value enables collection and printing of simple timing information about the accelerator regions and generated kernels.



Turn off all CUDA Profilers (NVIDIA's Visual Profiler, NVPROF, CUDA_PROFILE, etc) when enabling NVCOMPILER_ACC_TIME, they use the same library to gather performance data and cannot be used concurrently.

Accelerator Kernel Timing Data

```
bb04.f90
  s1
    15: region entered 1 times
        time(us): total=1490738
                init=1489138 region=1600
                kernels=155 data=1445
        w/o init: total=1600 max=1600
                min=1600 avg=1600
    18: kernel launched 1 times
        time(us): total=155 max=155 min=155 avg=155
```

In this example, a number of things are occurring:

- For each accelerator region, the file name `bb04.f90` and subroutine or function name `s1` is printed, with the line number of the accelerator region, which in the example is 15.

- ▶ The library counts how many times the region is entered (1 in the example) and the microseconds spent in the region (in this example 1490738), which is split into initialization time (in this example 1489138) and execution time (in this example 1600).
- ▶ The execution time is then divided into kernel execution time and data transfer time between the host and GPU.
- ▶ For each kernel, the line number is given, (18 in the example), along with a count of kernel launches, and the total, maximum, minimum, and average time spent in the kernel, all of which are 155 in this example.

6.15. OpenACC Runtime Libraries

This section provides an overview of the user-callable functions and library routines that are available for use by programmers to query the accelerator features and to control behavior of accelerator-enabled programs at runtime.



In Fortran, none of the OpenACC runtime library routines may be called from a PURE or ELEMENTAL procedure.

6.15.1. Runtime Library Definitions

There are separate runtime library files for Fortran, and for C++ and C.

C++ and C Runtime Library Files

In C++ and C, prototypes for the runtime library routines are available in a header file named `accel.h`. All the library routines are `extern` functions with 'C' linkage. This file defines:

- ▶ The prototypes of all routines in this section.
- ▶ Any data types used in those prototypes, including an enumeration type to describe types of accelerators.

Fortran Runtime Library Files

In Fortran, interface declarations are provided in a Fortran include file named `accel_lib.h` and in a Fortran module named `accel_lib`. These files define:

- ▶ Interfaces for all routines in this section.
- ▶ Integer parameters to define integer kinds for arguments to those routines.
- ▶ Integer parameters to describe types of accelerators.

6.15.2. Runtime Library Routines

Table 15 lists and briefly describes the runtime library routines supported by the NVIDIA HPC Compilers in addition to the standard OpenACC runtime API routines.

Table 15 Accelerator Runtime Library Routines

This Runtime Library Routine...	Does this...
<code>acc_allocs</code>	Returns the number of arrays allocated in data or compute regions.
<code>acc_bytesalloc</code>	Returns the total bytes allocated by data or compute regions.
<code>acc_bytesin</code>	Returns the total bytes copied in to the accelerator by data or compute regions.
<code>acc_bytesout</code>	Returns the total bytes copied out from the accelerator by data or compute regions.
<code>acc_copyins</code>	Returns the number of arrays copied in to the accelerator by data or compute regions.
<code>acc_copyouts</code>	Returns the number of arrays copied out from the accelerator by data or compute regions.
<code>acc_disable_time</code>	Tells the runtime to stop profiling accelerator regions and kernels.
<code>acc_enable_time</code>	Tells the runtime to start profiling accelerator regions and kernels, if it is not already doing so.
<code>acc_exec_time</code>	Returns the number of microseconds spent on the accelerator executing kernels.
<code>acc_frees</code>	Returns the number of arrays freed or deallocated in data or compute regions.
<code>acc_get_device</code>	Returns the type of accelerator device used to run the next accelerator region, if one is selected.
<code>acc_get_device_num</code>	Returns the number of the device being used to execute an accelerator region.
<code>acc_get_free_memory</code>	Returns the total available free memory on the attached accelerator device.
<code>acc_get_memory</code>	Returns the total memory on the attached accelerator device.
<code>acc_get_num_devices</code>	Returns the number of accelerator devices of the given type attached to the host.
<code>acc_kernels</code>	Returns the number of accelerator kernels launched since the start of the program.
<code>acc_present_dump</code>	Summarizes all data present on the current device.
<code>acc_present_dump_all</code>	Summarizes all data present on all devices.
<code>acc_regions</code>	Returns the number of accelerator regions entered since the start of the program.
<code>acc_total_time</code>	Returns the number of microseconds spent in accelerator compute regions and in moving data for accelerator data regions.

6.16. Supported Intrinsics

An intrinsic is a function available in a given language whose implementation is handled specifically by the compiler. Typically, an intrinsic substitutes a sequence of automatically-generated instructions for the original function call. Since the compiler has an intimate knowledge of the intrinsic function, it can better integrate it and optimize it for the situation.

Intrinsics make the use of processor-specific enhancements easier because they provide a language interface to assembly instructions. In doing so, the compiler manages things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.

This section contains an overview of the Fortran and C intrinsics that the accelerator supports.

6.16.1. Supported Fortran Intrinsics Summary Table

Table 16 is an alphabetical summary of the supported Fortran intrinsics that the accelerator supports. These functions are specific to Fortran 90/95 unless otherwise specified.

In most cases support is provided for all the data types for which the intrinsic is valid. When support is available for only certain data types, the middle column of the table specifies which ones, using the following codes:

I for integer

S for single precision real

C for single precision complex

D for double precision real

Z for double precision complex

Table 16 Supported Fortran Intrinsics

This intrinsic		Returns this value ...
ABS	I,S,D	absolute value of the supplied argument.
ACOS		arccosine of the specified value.
AINT		truncation of the supplied value to a whole number.
ANINT		nearest whole number to the supplied argument.
ASIN		arcsine of the specified value.
ATAN		arctangent of the specified value.
ATAN2		arctangent of the specified value.
COS	S,D,C,Z	cosine of the specified value.
COSH		hyperbolic cosine of the specified value.
DBLE	S,D	conversion of the value to double precision real.
DPROD		double precision real product.
EXP	S,D,C,Z	exponential value of the argument.

This intrinsic		Returns this value ...
IAND		result of a bit-by-bit logical AND on the arguments.
IEOR		result of a bit-by-bit logical exclusive OR on the arguments.
INT	I,S,D	conversion of the value to integer type.
IOR		result of a bit-by-bit logical OR on the arguments.
LOG	S,D,C,Z	natural logarithm of the specified value.
LOG10		base-10 logarithm of the specified value.
MAX		maximum value of the supplied arguments.
MIN		minimum value of the supplied arguments.
MOD	I	remainder of the division.
NINT		nearest integer to the real argument.
NOT		result of a bit-by-bit logical complement on the argument.
POW		value of the first argument raised to the power of the second argument.
REAL	I,S,D	conversion of the argument to real.
SIGN		absolute value of A times the sign of B.
SIN	S,D,C,Z	value of the sine of the argument.
SINH		hyperbolic sine of the argument.
SQRT	S,D,C,Z	square root of the argument.
TAN		tangent of the specified value.
TANH		hyperbolic tangent of the specified value.

6.16.2. Supported C Intrinsics Summary Table

This section contains two alphabetical summaries – one for double functions and a second for float functions. These lists contain only those C intrinsics that the accelerator supports.

Table 17 Supported C Intrinsic Double Functions

This intrinsic	Returns this value ...
acos	arccosine of the specified value.
asin	arcsine of the specified value.
atan	arctangent of the specified value.
atan2	arctangent of y/x, where y is the first argument, x the second.
cos	cosine of the specified value.
cosh	hyperbolic cosine of the specified value.
exp	exponential value of the argument.
fabs	absolute value of the argument.

This intrinsic	Returns this value ...
fmax	maximum value of the two supplied arguments
fmin	minimum value of the two supplied arguments
log	natural logarithm of the specified value.
log10	base-10 logarithm of the specified value.
pow	value of the first argument raised to the power of the second argument.
sin	value of the sine of the argument.
sinh	hyperbolic sine of the argument.
sqrt	square root of the argument.
tan	tangent of the specified value.
tanh	hyperbolic tangent of the specified value.

Table 18 Supported C Intrinsic Float Functions

This intrinsic	Returns this value ...
acosf	arccosine of the specified value.
asinf	arcsine of the specified value.
atanf	arctangent of the specified value.
atan2f	arctangent of y/x, where y is the first argument, x the second.
cosf	cosine of the specified value.
coshf	hyperbolic cosine of the specified value.
expf	exponential value of the floating-point argument.
fabsf	absolute value of the floating-point argument.
logf	natural logarithm of the specified value.
log10f	base-10 logarithm of the specified value.
powf	value of the first argument raised to the power of the second argument.
sinf	value of the sine of the argument.
sinhf	hyperbolic sine of the argument.
sqrtf	square root of the argument.
tanf	tangent of the specified value.
tanhf	hyperbolic tangent of the specified value.

Chapter 7.

PCAST

Parallel Compiler Assisted Software Testing (PCAST) is a set of API calls and compiler directives useful in testing program correctness. Numerical results produced by a program can diverge when parts of the program are mapped onto a GPU, when new or additional compiler options are used, or when changes are made to the program itself. PCAST can help you determine where these divergences begin, and pinpoint the changes that cause them. It is useful in other situations as well, including when using new libraries, determining whether parallel execution is safe, or porting programs from one ISA or type of processor to another.

7.1. Overview

PCAST Comparisons can be performed in two ways. The first saves the initial run's data into a file through the **pcast_compare** call or directive. Add the calls or directives to your application where you want intermediate results to be compared. Then, execute the program to save the "golden" results where the values are known to be correct. During subsequent runs of the program, the same **pcast_compare** calls or directives will compare the computed intermediate results to the saved "golden" results and report the differences.

The second approach works in conjunction with the NVIDIA OpenACC implementation to compare GPU computation against the same program running on a CPU. In this case, all compute constructs are performed redundantly, both on the CPU and GPU. GPU results are compared against the CPU results, and differences reported. This is essentially like the first case where the CPU-calculated values are treated as the "golden" results. GPU to CPU comparisons can be done implicitly at the end of data regions with the **autocompare** flag or explicitly after kernels with the **acc_compare** call or directive.

With the **autocompare** flag, OpenACC regions will run redundantly on the CPU and GPU. On an OpenACC region exit where data is to be downloaded from device to host, PCAST will compare the values calculated on the CPU with those calculated in the GPU. Comparisons done with **autocompare** or **acc_compare** are handled in memory and do not write results to an intermediate file.

The following table outlines the supported data types that can be used with PCAST. Short, integer, long, and half precision data types are not supported with **ABS**, **REL**, **ULP**, or **IEEE** options; only a bit-for-bit comparison is supported.

For floating-point types, PCAST can calculate absolute, relative, and unit-last-place differences. Absolute differences measures only the absolute value of the difference (subtraction) between two values, i.e. $abs(A-B)$. Relative differences are calculated as a ratio between the difference of values, $A-B$, and the previous value A ; $abs((A-B)/A)$. Unit-least precision (Unit-last place) is a measure of the smallest distance between two values A and B . With the **ULP** option set, PCAST will report if the calculated ULP between two numbers is greater than some threshold.

Table 19 Supported Types for Tolerance Measurements

C/C++ Type	Fortran Type	ABS	REL	ULP	IEEE
float	real, real(4)	Yes	Yes	Yes	Yes
double	double precision, real(8)	Yes	Yes	Yes	Yes
float _Complex	complex, complex(4)	Yes	Yes	Yes	Yes
double _Complex	complex(8)	Yes	Yes	Yes	Yes
-	real(2)	No	No	No	No
(un)signed short	integer(2)	N/A	N/A	N/A	N/A
(un)signed int	integer, integer(4)	N/A	N/A	N/A	N/A
(un)signed long	integer(8)	N/A	N/A	N/A	N/A

7.2. PCAST with a "Golden" File

The run-time call **pcast_compare** highlights differences between successive program runs. It has two modes of operation, depending on the presence of a data file named *pcast_compare.dat* by default. If the file does not exist, **pcast_compare** assumes this is the first "golden" run. It will create the file and fill it with the computed data at each call to **pcast_compare**. If the file exists, **pcast_compare** assumes it is a test run. It will read the file and compare the computed data with the saved data from the file. The default behavior is to consider the first 50 differences to be a reportable error, no matter how small.

By default, the **pcast_compare.dat** file is in the same directory as the executable. The behavior of **pcast_compare**, and other comparison parameters, can be changed at runtime with the PCAST_COMPARE environment variable discussed in the [Environment Variables](#) section.

The signature of **pcast_compare** for C++ and C is:

```
void pcast_compare(void*, char*, size_t, char*, char*, char*, int);
```

The signature of **pcast_compare** for Fortran is:

```

subroutine pcast_compare(a, datatype, len, varname, filename, funcname, lineno)
  type(*), dimension(..) :: a
  character(*) :: datatype, varname, filename, funcname
  integer(8),value :: len
  integer(4),value :: lineno

```

The call takes seven arguments:

1. The address of the data to be saved or compared.
2. A string containing the data type.
3. The number of elements to compare.
4. A string treated as the variable name.
5. A string treated as the source file name.
6. A string treated as the function name.
7. An integer treated as a line number.

For example, the **pcast_compare** runtime call can be invoked like the following:

```
pcast_compare(a, "float", N, "a", "pcast_compare03.c", "main", 1);
```

```
call pcast_compare(a, 'real', n, 'a', 'pcast_compare1.f90', 'program', 9)
```

The caller should give meaningful names to the last four arguments. They can be anything, since they only serve to annotate the report. It is imperative that the identifiers are not modified between comparisons; comparisons must be called in the same order for each program run. If, for example, you are calling **pcast_compare** inside a loop, it is reasonable to set the last argument to be the loop index.

There also exists a directive form of the **pcast_compare**, which is functionally the same as the runtime call. It can be used at any point in the program to compare the current value of data to that recorded in the golden file, same as the runtime call. There are two benefits to using the directive over the API call:

1. The directive syntax is much simpler than the API syntax. Most of what the compare call needs to output data to the user can be gleaned by the compiler at compile-time (The type, variable name, file name, function name, and line number).

```
#pragma nvidia compare(a[0:n])
```

as opposed to:

```
pcast_compare(a, "float", N, "a", "pcast_compare03.c", "main", 1);
```

2. The directive is only enabled when the **-Mpcast** flag is set, so the source need not be changed when testing is complete. Consider the following usage examples:

```

#pragma nvidia compare(a[0:N]) // C++ and C
!$nvf compare(a(1:N)) ! Fortran

```

The directive interface is given below in C++ or C style, and in Fortran. Note that for Fortran, **var-list** is a variable name, a subarray specification, an array element, or a composite variable member.

```
#pragma nvidia compare (var-list) // C++ and C
!$nvf compare (var-list) ! Fortran
```

Let's look at an example of

```
int main() {
    int size = 1000;
    int i, t;
    float *a1;
    float *a2;

    a1 = (float*)malloc(sizeof(float)*size);
    a2 = (float*)malloc(sizeof(float)*size);

    for (i = 0; i < size; i++) {
        a1[i] = 1.0f;
        a2[i] = 2.0f;
    }

    for (t = 0; t < 5; t++) {
        for(i = 0; i < size; i++) {
            a2[i] += a1[i];
        }
        pcast_compare(a2, "float", size, "main", 23);
    }
    return 0;
}
```

Compile the example using these compiler options:

```
> nvc -fast -o a.out example.c
```

Compiling with redundant or autocompare options are not required to use `pcast_compare`. Once again, running the compiled executable using the options below, results in the following output:

```
> PCAST_COMPARE=summary,rel=1 ./out.o
datafile pcast_compare.dat created with 5 blocks, 5000 elements, 20000 bytes
> PCAST_COMPARE=summary,rel=1 ./out.o
datafile pcast_compare.dat compared with 5 blocks, 5000 elements, 20000 bytes
no errors found
relative tolerance = 0.100000, rel=1
```

Running the program for the first time, the data file "pcast_compare.dat" is created. Subsequent runs compare calculated data against this file. Use the **PCAST_COMPARE** environment variable to set the name of the file, or force the program to create a new file on the disk with **PCAST_COMPARE=create**.

The same example above can be written with the compare directive. Notice how much more concise the directive is to the update host and **pcast_compare** calls.

```
int main() {
    int size = 1000;
```



```

int i, t;
float *a1;
float *a2;

a1 = (float*)malloc(sizeof(float)*size);
a2 = (float*)malloc(sizeof(float)*size);

for (i = 0; i < size; i++) {
    a1[i] = 1.0f;
    a2[i] = 2.0f;
}

for (t = 0; t < 5; t++) {
    for(i = 0; i < size; i++) {
        a2[i] += a1[i];
    }
    #pragma nvidia compare(a2[0:size])
}
return 0;
}

```

With the directive, you will want to add "-Mpcast" to the compilation line to enable the directive. Other than that, the output from this program is identical to the runtime example above.

7.3. PCAST with OpenACC

PCAST can also be used with the NVIDIA OpenACC implementation to compare GPU computation against the same program running on a CPU. In this case, all compute constructs are performed redundantly on both the CPU and GPU. The CPU results are considered to be the "golden master" copy which GPU results are compared against.

There are two ways to perform comparisons with GPU-calculated results. The first is with the explicit call or directive **acc_compare**. To use **acc_compare**, you must compile with **-acc -gpu=redundant** to force the CPU and GPU to compute results redundantly. Then, insert calls to **acc_compare** or put an **acc compare** directive at points where you want to compare the GPU-computed values against those computed by the CPU.

The second approach is to turn on autocompare mode by compiling with **-acc -gpu=autocompare**. In autocompare mode, PCAST will automatically perform a comparison at each point where data is moved from the device to the host. It does not require the programmer to add any additional directives or runtime calls; it's a convenient way to do all comparisons at the end of a data region. If there are multiple compute kernels within a data region, and you're only interested in one specific kernel, you should use the previously-mentioned **acc_compare** to target a specific kernel. Note that autocompare mode implies **-gpu=redundant**.

During redundant execution, the compiler will generate both CPU and GPU code for each compute construct. At runtime, both the CPU and GPU versions will execute redundantly, with the CPU code reading and modifying values in system memory and the GPU reading and modifying values in device memory. Insert calls to **acc_compare()** calls (or the equivalent **acc compare** directive) at points where you want to compare the GPU-computed values against CPU-computed values. PCAST

treats the values generated by the CPU code as the "golden" values. It will compare those results against GPU values. Unlike **pcast_compare**, **acc_compare** does not write to an intermediary file; the comparisons are done in-memory.

acc_compare only has two arguments: a pointer to the data to be compared, *hostptr*, and the number of elements to compare, *count*. The type can be inferred in the OpenACC runtime, so it doesn't need to be specified. The C++ and C interface is given below:

```
void acc_compare(void *, size_t);
```

And in Fortran:

```
subroutine acc_compare(a)
subroutine acc_compare(a, len)
    type(*), dimension(*) :: a
    integer(8), value :: len
```

You can call **acc_compare** on any variable or array that is present in device memory. You can also call **acc_compare_all** (no arguments) to compare all values that are present in device memory against the corresponding values in host memory.

```
void acc_compare_all()
```

```
subroutine acc_compare_all()
```

Directive forms of the **acc_compare** calls exist. They work the same as the API calls and can be used in lieu of them. Similar to PCAST **compare** directives, **acc compare** directives are ignored when redundant or autocompare modes are not enabled on the compilation line.

The **acc compare** directive takes one or more arguments, or the 'all' clause (which corresponds to **acc_compare_all()**). The interfaces are given below in C++ or C, and Fortran respectively. Argument "var-list" can be a variable name, a sub-array specification, and array element, or a composite variable member.

```
#pragma acc compare [ (var-list) | all ]
```

```
$(acc compare [ (var-list) | all ]
```

For example:

```
#pragma acc compare(a[0:N])
#pragma acc compare all
!$acc compare(a, b)
!$acc compare(a(1:N))
!$acc compare all
```

Consider the following OpenACC program that uses the **acc_compare()** API call and an **acc compare** directive. This Fortran example uses real*4 and real*8 arrays.

```

program main
  use openacc
  implicit none
  parameter N = 1000
  integer :: i
  real :: a(N)
  real*4 :: b(N)
  real(4) :: c(N)
  double precision :: d(N)
  real*8 :: e(N)
  real(8) :: f(N)

  d = 1.0d0
  e = 0.1d0

  !$acc data copyout(a, b, c, f) copyin(d, e)

  !$acc parallel loop
  do i = 1,N
    a(i) = 1.0
    b(i) = 2.0
    c(i) = 0.0
  enddo
  !$acc end parallel

  !$acc compare(a(1:N), b(1:N), c(1:N))

  !$acc parallel loop
  do i = 1,N
    f(i) = d(i) * e(i)
  enddo
  !$acc end parallel

  !$acc compare(f)

  !$acc parallel loop
  do i = 1,N
    a(i) = 1.0
    b(i) = 1.0
    c(i) = 1.0
  enddo
  !$acc end parallel

  call acc_compare(a, N)
  call acc_compare(b, N)
  call acc_compare(c, N)

  !$acc parallel loop
  do i = 1,N
    f(i) = 1.0D0
  enddo
  !$acc end parallel

  call acc_compare_all()

  !$acc parallel loop
  do i = 1,N
    a(i) = 3.14;
    b(i) = 3.14;
    c(i) = 3.14;
    f(i) = 3.14d0;
  enddo
  !$acc end parallel

```

```

! In redundant mode, no comparison is performed here. In
! autocompare mode, a comparison is made for a, b, c, and f (but
! not e and d), since they are copied out of the data region.

!$acc end data

call verify(N, a, b, c, f)
end program

subroutine verify(N, a, b, c, f)
integer, intent(in) :: N
real, intent(in) :: a(N)
real*4, intent(in) :: b(N)
real(4), intent(in) :: c(N)
real(8), intent(in) :: f(N)
integer :: i, errcnt

errcnt = 0
do i=1,N
if(abs(a(i) - 3.14e0) .gt. 1.0e-06) then
errcnt = errcnt + 1
endif
end do
do i=1,N
if(abs(b(i) - 3.14e0) .gt. 1.0e-06) then
errcnt = errcnt + 1
endif
end do
do i=1,N
if(abs(c(i) - 3.14e0) .gt. 1.0e-06) then
errcnt = errcnt + 1
endif
end do
do i=1,N
if(abs(f(i) - 3.14d0) .gt. 1.0d-06) then
errcnt = errcnt + 1
endif
end do

if(errcnt /= 0) then
write (*, *) "FAILED"
else
write (*, *) "PASSED"
endif
end subroutine verify

```

The program can be compiled with the following command:

```

> nvfortran -fast -acc -gpu=redundant -Minfo=accel example.F90
main:
16, Generating copyout(a(:),b(:))
Generating copyin(e(:))
Generating copyout(f(:),c(:))
Generating copyin(d(:))
18, Generating Tesla code
19, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
26, Generating acc compare(c(:),b(:),a(:))
28, Generating Tesla code
29, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
34, Generating acc compare(f(:))
36, Generating Tesla code
37, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
48, Generating Tesla code
49, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
56, Generating Tesla code

```

```
57, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

Here, you can see where the acc compare directives are generated on lines 26 and 34. The program can be run with the following command:

```
> ./a.out
PASSED
```

As you can see, no PCAST output is generated when the comparisons match. We can get more information with the summary option:

```
> PCAST_COMPARE=summary ./a.out
PASSED
compared 13 blocks, 13000 elements, 68000 bytes
no errors found
absolute tolerance = 0.0000000000000000e+00, abs=0
```

There are 13 blocks compared. Let's count the blocks in the compare calls.

```
!$acc compare(a(1:N), b(1:N), c(1:N))
```

Compares three blocks, one each for a, b, and c.

```
!$acc compare(f)
```

Compares one block for f.

```
call acc_compare(a, N)
call acc_compare(b, N)
call acc_compare(c, N)
```

Each call compares one block for their respective array.

```
call acc_compare_all()
```

Compares one block for each array present on the device (a, b, c, d, e, and f) for a total of 6 blocks.

If the same example is compiled with autocompare, we'll see four additional comparisons, since the four arrays that are copied out (with the copyout clause) are compared at the end of the data region.

```
> nvfortran -fast -acc -gpu=autocompare example.F90
> PCAST_COMPARE=summary ./a.out
PASSED
compared 17 blocks, 17000 elements, 88000 bytes
no errors found
absolute tolerance = 0.0000000000000000e+00, abs=0
```

7.4. Limitations

There are currently a few limitations with using PCAST that are worth keeping in mind.

- ▶ Comparisons are not thread-safe. If you are using PCAST with multiple threads, ensure that only one thread is doing the comparisons. This is especially true if you are using PCAST with MPI. If you use `pcast_compare` with MPI, you must make sure that only one thread is writing to the comparison file. Or, use a script to set `PCAST_COMPARE` to encode the file name with the MPI rank.
- ▶ Comparisons must be done with like types; you cannot compare one type with another. It is not possible to, for example, check for differing results after changing from double precision to single. Comparisons are limited to those present in table [Table 19](#). Currently there is no support for structured or derived types.
- ▶ The `-gpu=managed` option is incompatible with `autocompre` and `acc_compare`. Both the CPU and GPU need to calculate result separately and to do so they must have their own working memory spaces.
- ▶ If you do any data movement on the device, you must account for it on the host. For example, if you are using CUDA-aware MPI or GPU-accelerated libraries that modify device data, then you must also make the host aware of the changes. In these cases it is helpful to use the `host_data` clause, which allows you to use device addresses within host code.

7.5. Environment Variables

Behavior of PCAST/Autocompare is controlled through the `PCAST_COMPARE` variable. Options can be specified in a comma-separated list:

`PCAST_COMPARE=<opt1>,<opt2>,...`

If no options are specified, the default is to perform comparisons with `abs=0`.

Comparison options are not mutually exclusive. PCAST can compare absolute differences with some `n=3` and relative differences with a different threshold, e.g. `n=5`; `PCAST_COMPARE=abs=3,rel=5,...`

You can specify either an absolute or relative location to be used with the `datafile` option. The parent directory should be owned by the same user executing the comparisons and the datafile should have the appropriate read/write permissions set.

Table 20 PCAST_COMPARE Options

Option	Description
<code>abs=n</code>	Compare absolute difference; tolerate differences up to $10^{(-n)}$, only applicable to floating point types. Default value is 0
<code>create</code>	Specifies that this is the run that will produce the reference file (<code>pcast_compare</code> only)
<code>compare</code>	Specifies that the current run will be compared with a reference file (<code>pcast_compare</code> only)

Option	Description
datafile="name"	Name of the file that data will be saved to, or compared against. If empty will use the default, 'pcast_compare.dat' (pcast_compare only)
disable	Calls to pcast_compare, acc_compare, acc_compare_all, and directives (pcast compare, acc compare, and acc compare) all immediately return from the runtime with no effect. Note that this doesn't disable redundant execution; that will require a recompile.
ieee	Compare IEEE NaN checks (only implemented for floats and doubles)
outputfile="name"	Save comparison output to a specific file. Default behavior is to output to stderr
patch	Patch errors (outside tolerance) with correct values
patchall	Patch all differences (inside and outside tolerance) with correct values
rel=n	Compare relative difference; tolerated differences up to 10^{-n} , only applicable to floating point types. Default value is 0.
report=n	Report up to n (default of 50) passes/fails
reportall	Report all passes and fails (overrides limit set in report=n)
reportpass	Report passes; respects limit set with report=n
silent	Suppress output - overrides all other output options, including summary and verbose
stop	Stop at first differences
summary	Print summary of comparisons at end of run
ulp=n	Compare Unit of Least Precision difference (only for floats and doubles)
verbose	Outputs more details of comparison (including patches)
verboseautocompare	Outputs verbose reporting of what and where the host is comparing (autocompare only)

Chapter 8.

USING MPI

MPI (the Message Passing Interface) is an industry-standard application programming interface designed for rapid data exchange between processors in a distributed-memory environment. MPI is computer software used in scalable computer systems that allows the processes of a parallel application to communicate with one another.

The NVIDIA HPC SDK includes a pre-compiled version of Open MPI. You can build using alternate versions of MPI with the `-I`, `-L`, and `-l` options.

This section describes how to use Open MPI with the NVIDIA HPC Compilers.

8.1. Using Open MPI on Linux

The NVIDIA HPC Compilers for Linux ship with a pre-compiled version of Open MPI that includes everything required to compile, execute and debug MPI programs using Open MPI.

To build an application using Open MPI, use the Open MPI compiler wrappers: `mpicc`, `mpic++` and `mpifort`. These wrappers automatically set up the compiler commands with the correct include file search paths, library directories, and link libraries.

The following MPI example program uses Open MPI.

```
$ cd my_example_dir
$ cp -r /opt/nvidia/hpc_sdk/Linux_x86_64/2020/examples/MPI/samples/mpihello .
$ cd mpihello
$ export PATH=/opt/nvidia/hpc_sdk/Linux_x86_64/20.9/mpi/openmpi/bin:$PATH
$ mpifort mpihello.f -o mpihello
```

```
$ mpiexec mpihello
Hello world! I'm node 0
```

```
$ mpiexec -np 4 mpihello
Hello world! I'm node 0
Hello world! I'm node 2
Hello world! I'm node 1
Hello world! I'm node 3
```


To build an application using Open MPI for debugging, add `-g` to the compiler wrapper command line arguments.

8.2. Using MPI Compiler Wrappers

When you use MPI compiler wrappers to build with the `-fpic` or `-mmodel=medium` options, then you must specify `-fortranlibs` to link with the correct libraries. Here are a few examples:

For a static link to the MPI libraries, use this command:

```
% mpifort hello.f
```

For a dynamic link to the MPI libraries, use this command:

```
% mpifort hello.f -fortranlibs
```

To compile with `-fpic`, which, by default, invokes dynamic linking, use this command:

```
% mpifort -fpic -fortranlibs hello.f
```

To compile with `-mmodel=medium`, use this command:

```
% mpifort -mmodel=medium -fortranlibs hello.f
```

8.3. Testing and Benchmarking

The `/opt/nvidia/hpc_sdk/Linux_x86_64/2020/examples/MPI` directory contains various benchmarks and tests. Copy this directory into a local working directory by issuing the following command:

```
% cp -r /opt/nvidia/hpc_sdk/Linux_x86_64/2020/examples/MPI .
```

There are several example programs available in this directory.

Chapter 9.

CREATING AND USING LIBRARIES

A library is a collection of functions or subprograms that are grouped for reference and ease of linking. This section discusses issues related to NVIDIA-supplied compiler libraries. Specifically, it addresses the use of C++ and C builtin functions in place of the corresponding libc routines, creation of dynamically linked libraries, known as shared objects or shared libraries, and math libraries.



This section does not duplicate material related to using libraries for inlining, described in [Creating an Inline Library](#) or information related to runtime library routines available to OpenMP programmers, described in [Runtime Library Routines](#).

NVIDIA provides libraries that export C interfaces by using Fortran modules.

9.1. Using builtin Math Functions in C++ and C

The name of the math header file is `math.h`. Include the math header file in all of your source files that use a math library routine as in the following example, which calculates the inverse cosine of 3.5.

```
#include <math.h>
#include<stdio.h>
#define PI 3.1415926535
void main()
{
    double x, y;
    x = PI/3.0;
    y = acos(0.5);
    printf('%f %f\n', x, y);
}
```

Including `math.h` causes the NVIDIA C++ and C compilers to use builtin functions, which are much more efficient than library calls. In particular, if you include `math.h`, the following intrinsics calls are processed using builtins:

<code>abs</code>	<code>acosf</code>	<code>asinf</code>	<code>atan</code>	<code>atan2</code>	<code>atan2f</code>
<code>atanf</code>	<code>cos</code>	<code>cosf</code>	<code>exp</code>	<code>expf</code>	<code>fabs</code>
<code>fabsf</code>	<code>fmax</code>	<code>fmaxf</code>	<code>fmin</code>	<code>fminf</code>	<code>log</code>

log10	log10f	logf	pow	powf	sin
sinf	sqrt	sqrtf	tan	tanf	

9.2. Using System Library Routines

Release 20.9 of the NVIDIA HPC Compilers runtime libraries makes use of Linux system libraries to implement, for example, OpenMP and Fortran I/O. The NVIDIA HPC Compilers runtime libraries make use of several additional system library routines.

On 64-bit Linux systems, the system library routines used include these:

aio_error	aio_write	pthread_mutex_init	sleep
aio_read	calloc	pthread_mutex_lock	
aio_return	getrlimit	pthread_mutex_unlock	
aio_suspend	pthread_attr_init	setrlimit	

9.3. Creating and Using Shared Object Files on Linux

All of the NVIDIA HPC Fortran, C++ and C compilers support creation of shared object files. Unlike statically-linked object and library files, shared object files link and resolve references with an executable at runtime via a dynamic linker supplied with your operating system. The NVIDIA HPC Compilers must generate position independent code to support creation of shared objects by the linker. However, this is not the default. You must create object files with position independent code and shared object files that will include them.

9.3.1. Procedure to create a use a shared object file

The following steps describe how to create and use a shared object file.

1. Create an object file with position independent code.

To do this, compile your code with the appropriate NVIDIA HPC compiler using the `-fpic` option, or one of the equivalent options, such as `-fPIC`, `-Kpic`, and `-KPIC`, which are supported for compatibility with other systems. For example, use the following command to create an object file with position independent code using `nvfortran`:

```
% nvfortran -c -fpic tobeshared.f
```

2. Produce a shared object file.

To do this, use the appropriate NVIDIA HPC compiler to invoke the linker supplied with your system. It is customary to name such files using a `.so` filename extension. On Linux, you do this by passing the `-shared` option to the linker:

```
% nvfortran -shared -o tobeshared.so tobeshared.o
```



Compilation and generation of the shared object can be performed in one step using both the `-fpic` option and the appropriate option for generation of a shared object file.

3. Use a shared object file.

To do this, use the appropriate NVIDIA HPC compiler to compile and link the program which will reference functions or subroutines in the shared object file, and list the shared object on the link line, as shown here:

```
% nvfortran -o myprog myprog.f tobeshared.so
```

4. Make the executable available.

You now have an executable `myprog` which does not include any code from functions or subroutines in `tobeshared.so`, but which can be executed and dynamically linked to that code. By default, when the program is linked to produce `myprog`, no assumptions are made on the location of `tobeshared.so`. Therefore, for `myprog` to execute correctly, you must initialize the environment variable `LD_LIBRARY_PATH` to include the directory containing `tobeshared.so`. If `LD_LIBRARY_PATH` is already initialized, it is important not to overwrite its contents. If you have placed `tobeshared.so` in directory `/home/myusername/bin`, you can initialize `LD_LIBRARY_PATH` to include that directory and preserve its existing contents, as shown in the following:

```
% setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":/home/myusername/bin
```

If you know that `tobeshared.so` always resides in a specific directory, you can create the executable `myprog` in a form that assumes this directory by using the `-R` link-time option. For example, you can link as follows:

```
% nvfortran -o myprog myprog.f tobeshared.so -R/home/myusername/bin
```



As with the `-L` option, there is no space between `-R` and the directory name. If the `-R` option is used, it is not necessary to initialize `LD_LIBRARY_PATH`.

In the previous example, the dynamic linker always looks in `/home/myusername/bin` to resolve references to `tobeshared.so`. By default, if the `LD_LIBRARY_PATH` environment variable is not set, the linker only searches `/usr/lib` and `/lib` for shared objects.

9.3.2. ldd Command

The `ldd` command is a useful tool when working with shared object files and executables that reference them. When applied to an executable, as shown in the following example, `ldd` lists all shared object files referenced in the executable along with the pathname of the directory from which they will be extracted.

```
% ldd myprog
```

If the pathname is not hard-coded using the `-R` option, and if `LD_LIBRARY_PATH` is not initialized, the pathname is listed as "not found". For more information on `ldd`, its options and usage, see the online man page for `ldd`.

9.4. Using LIB3F

The NVFORTRAN compiler includes support for the de facto standard LIB3F library routines. See the Fortran Language Reference manual for a complete list of available routines in the NVIDIA implementation of LIB3F.

9.5. LAPACK, BLAS and FFTs

The NVIDIA HPC SDK includes a BLAS and LAPACK library based on the customized OpenBLAS project source and built with the NVIDIA HPC Compilers. The LAPACK library is called `liblapack.a`. The BLAS library is called `libblas.a`.

To use these libraries, simply link them in using the `-l` option when linking your main program:

```
% nvfortran myprog.f -llapack -lblas
```

9.6. Linking with ScaLAPACK

The ScaLAPACK libraries are automatically installed with each MPI library version which accompanies an NVIDIA HPC SDK installation. You can link with the ScaLAPACK libraries by specifying `-Mscalapack` on any of the MPI wrapper command lines. For example:

```
% mpifort myprog.f -Mscalapack
```

A pre-built version of the BLAS library is automatically added when the `-Mscalapack` switch is specified. If you wish to use a different BLAS library, and still use the `-Mscalapack` switch, then you can list the set of libraries explicitly on your link line.

9.7. The C++ Standard Template Library

On Linux, the GNU-compatible `nvc++` compiler uses the GNU g++ header files and Standard Template Library (STL) directly. The versions used are dependent on the version of the GNU compilers installed on your system, or specified when `makelocalrc` was run during installation of the NVIDIA HPC Compilers.

Chapter 10.

ENVIRONMENT VARIABLES

Environment variables allow you to set and pass information that can alter the default behavior of the NVIDIA HPC compilers and the executables which they generate. This section includes explanations of the environment variables specific to the NVIDIA HPC Compilers. .

- ▶ Standard OpenMP environment variables are used to control the behavior of OpenMP programs. OpenMP-related environment variables are described in the OpenMP section: [OpenMP Environment Variables](#).
- ▶ Several NVIDIA-specific environment variables can be used to control the behavior of OpenACC programs. OpenACC-related environment variables are described in the OpenACC section: [Environment Variables](#) and the [OpenACC Getting Started Guide](#), docs.nvidia.com/hpc-sdk/compilers/pdf/hpc20%RELEASE_VERSION_MINORopenacc_gs.pdf.

10.1. Setting Environment Variables

Before we look at the environment variables that you might use with the HPC compilers and tools, let's take a look at how to set environment variables. To illustrate how to set these variables in various environments, let's look at how a user might initialize a Linux shell environment to enable use of the NVIDIA HPC Compilers.

10.1.1. Setting Environment Variables on Linux

Let's assume that you want access to the NVIDIA products when you log in, and that you installed the NVIDIA HPC SDK in `/opt/nvidia/hpc_sdk`. For access at startup, you can add the following lines to your shell startup files on a Linux_x86_64 system.

For csh, use these commands:

```
% setenv NVHPCSDK /opt/nvidia/hpc_sdk
% setenv MANPATH "$MANPATH":$NVHPCSDK/Linux_x86-64/20.9/compilers/man
% set path = ($NVHPCSDK/Linux_x86_64/20.9/compilers/bin $path)
```

For bash, sh, zsh, or ksh, use these commands:

```
$ NVHPCSDK=/opt/nvidia/hpc_sdk; export NVHPCSDK
$ MANPATH=$MANPATH:$NVHPCSDK/Linux_x86_64/20.9/compilers/man; export MANPATH
$ PATH=$NVHPCSDK/Linux_x86_64/20.9/compilers/bin:$PATH; export PATH
```

On a Linux/OpenPOWER system replace **Linux_x86_64** with **Linux_ppc64le**, and on a Linux/Arm Server system replace it with **Linux_aarch64**.

10.2. HPC Compiler Related Environment Variables

The following table provides a listing of environment variables that affect the behavior of the NVIDIA HPC Compilers and the executables they generate.

Table 21 NVIDIA HPC Compilers Environment Variable Summary

Environment Variable	Description
FORTRANOPT	Allows the user to specify that the NVIDIA Fortran compiler should use VAX I/O or other custom I/O conventions.
GMON_OUT_PREFIX	Specifies the name of the output file for programs that are compiled and linked with the -pg option.
LD_LIBRARY_PATH	Specifies a colon-separated set of directories where libraries should first be searched, prior to searching the standard set of directories.
MANPATH	Sets the directories that are searched for manual pages associated with the command that the user types.
NCPUS	Sets the number of processes or threads used in parallel regions.
NCPUS_MAX	Limits the maximum number of processors or threads that can be used in a parallel region.
NO_STOP_MESSAGE	If used, the execution of a plain STOP statement does not produce the message <code>FORTRAN STOP</code> .
PATH	Determines which locations are searched for commands the user may type.
NVCOMPILER_CONTINUE	If set, when a program compiled with <code>-Mchkfpstk</code> is executed, the stack is automatically cleaned up and execution then continues.
NVCOMPILER_TERM	Controls the stack traceback and just-in-time debugging functionality.
NVCOMPILER_TERM_DEBUG	Overrides the default behavior when <code>NVCOMPILER_TERM</code> is set to debug.
PWD	Allows you to display the current directory.
STATIC_RANDOM_SEED	Forces the seed returned by <code>RANDOM_SEED</code> to be constant.
TMP	Sets the directory to use for temporary files created during execution of the HPC compilers and tools; interchangeable with <code>TMPDIR</code> .
TMPDIR	Sets the directory to use for temporary files created during execution of the HPC compilers and tools.

10.3. HPC Compilers Environment Variables

Use the environment variables listed in [Table 21](#) to alter the default behavior of the NVIDIA HPC Compilers and the executables which they generate. This section provides more detailed descriptions about the variables in this table.

10.3.1. FORTRANOPT

FORTRANOPT allows the user to adjust the behavior of the NVIDIA Fortran compiler.

- ▶ If FORTRANOPT exists and contains the value `vaxio`, the record length in the open statement is in units of 4-byte words, and the \$ edit descriptor only has an effect for lines beginning with a space or a plus sign (+).
- ▶ If FORTRANOPT exists and contains the value `format_relaxed`, an I/O item corresponding to a numerical edit descriptor (such as F, E, I, and so on) is not required to be a type implied by the descriptor.
- ▶ If FORTRANOPT exists and contains the value `no_minus_zero`, an I/O item corresponding to a numerical edit descriptor (such as F, E, I, and so on) equal to negative zero will be output as if it were positive zero.
- ▶ If FORTRANOPT exists and contains the value `crif`, a sequential formatted or list-directed record is allowed to be terminated with the character sequence `\r\n` (carriage return, newline). This approach is useful when reading records from a file produced on a Windows system.

The following example causes the NVIDIA Fortran compiler to use VAX I/O conventions:

```
% setenv FORTRANOPT vaxio
```

10.3.2. GMON_OUT_PREFIX

GMON_OUT_PREFIX specifies the name of the output file for programs that are compiled and linked with the `-pg` option. The default name is `gmon.out`.

If GMON_OUT_PREFIX is set, the name of the output file has GMON_OUT_PREFIX as a prefix. Further, the suffix is the pid of the running process. The prefix and suffix are separated by a dot. For example, if the output file is `mygmon`, then the full filename may look something similar to this: `mygmon.0012348567`.

The following example causes the NVIDIA Fortran compiler to use `nvout` as the output file for programs compiled and linked with the `-pg` option.

```
% setenv GMON_OUT_PREFIX nvout
```

10.3.3. LD_LIBRARY_PATH

The LD_LIBRARY_PATH variable is a colon-separated set of directories specifying where libraries should first be searched, prior to searching the standard set of directories. This variable is useful when debugging a new library or using a nonstandard library for special purposes.

The following csh example adds the current directory to your LD_LIBRARY_PATH variable.

```
% setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":./"
```


10.3.4. MANPATH

The `MANPATH` variable sets the directories that are searched for manual pages associated with the commands that the user types. When using NVIDIA HPC Compilers, it is important that you set your `PATH` to include the location of the compilers and then set the `MANPATH` variable to include the man pages associated with the products.

The following `csh` example targets the `Linux_x86_64` version of the compilers and enables access to the manual pages associated with them. The settings are similar for `Linux_ppc64le` or `Linux_aarch64` targets:

```
% set path = (/opt/nvidia/hpc_sdk/Linux_x86_64/20.9/compilers/bin $path
% setenv MANPATH "$MANPATH":/opt/nvidia/hpc_sdk/Linux_x86_64/20.9/compilers/man
```

10.3.5. NCPUS

You can use the `NCPUS` environment variable to set the number of processes or threads used in parallel regions. The default is to use only one process or thread, which is known as serial mode.



`OMP_NUM_THREADS` has the same functionality as `NCPUS`. For historical reasons, NVIDIA HPC Compilers support the environment variable `NCPUS`. If both `OMP_NUM_THREADS` and `NCPUS` are set, the value of `OMP_NUM_THREADS` takes precedence.

Setting `NCPUS` to a value larger than the number of physical processors or cores in your system can cause parallel programs to run very slowly.

10.3.6. NCPUS_MAX

You can use the `NCPUS_MAX` environment variable to limit the maximum number of processes or threads used in a parallel program. Attempts to dynamically set the number of processes or threads to a higher value, for example using `set_omp_num_threads()`, will cause the number of processes or threads to be set at the value of `NCPUS_MAX` rather than the value specified in the function call.

10.3.7. NO_STOP_MESSAGE

If the `NO_STOP_MESSAGE` variable exists, the execution of a plain `STOP` statement does not produce the message `FORTTRAN STOP`. The default behavior of the NVIDIA Fortran compiler is to issue this message.

10.3.8. PATH

The `PATH` variable determines the directories that are searched for commands that the user types. When using the NVIDIA HPC compilers, it is important that you set your `PATH` to include the location of the compilers.

The following csh example initializes path settings to use the Linux_x86_64 versions of the NVIDIA HPC Compilers. Settings for Linux_ppc64le and Linux_aarch64 are done similarly:

```
% set path = (/opt/nvidia/hpc_sdk/Linux_x86_64/20.9/compilers/bin $path)
```

10.3.9. NVCOMPILER_CONTINUE

You set the `NVCOMPILER_CONTINUE` variable to specify the actions to take before continuing with execution. For example, if the `NVCOMPILER_CONTINUE` environment variable is set and then a program that is compiled with `-Mchkfpstk` is executed, the stack is automatically cleaned up and execution then continues. If `NVCOMPILER_CONTINUE` is set to `verbose`, the stack is automatically cleaned up, a warning message is printed, and then execution continues.



There is a performance penalty associated with the stack cleanup.

10.3.10. NVCOMPILER_TERM

The `NVCOMPILER_TERM` environment variable controls the stack traceback and just-in-time debugging functionality. The runtime libraries use the value of `NVCOMPILER_TERM` to determine what action to take when a program abnormally terminates.

The value of `NVCOMPILER_TERM` is a comma-separated list of options. The commands for setting the environment variable follow.

- In csh:

```
% setenv NVCOMPILER_TERM option[,option...]
```

- In bash, sh, zsh, or ksh:

```
$ NVCOMPILER_TERM=option[,option...]
$ export NVCOMPILER_TERM
```

Table 22 lists the supported values for `option`. Following the table is a complete description of each option that indicates specifically how you might apply the option.

By default, all of these options are disabled.

Table 22 Supported `NVCOMPILER_TERM` Values

[no]debug	Enables/disables just-in-time debugging (debugging invoked on error)
[no]trace	Enables/disables stack traceback on error
[no]signal	Enables/disables establishment of signal handlers for common signals that cause program termination
[no]abort	Enables/disables calling the system termination routine <code>abort()</code>

[no]debug

This enables/disables just-in-time debugging. The default is `nodebug`.

When `NVCOMPILER_TERM` is set to `debug`, the command to which `NVCOMPILER_TERM_DEBUG` is set is invoked on error.

`[no]trace`

This enables/disables stack traceback on error.

`[no]signal`

This enables/disables establishing signal handlers for the most common signals that cause program termination. The default is `nosignal`. Setting `trace` and `debug` automatically enables `signal`. Specifically setting `nosignal` allows you to override this behavior.

`[no]abort`

This enables/disables calling the system termination routine `abort()`. The default is `noabort`. When `noabort` is in effect the process terminates by calling `_exit(127)`.

On Linux, when `abort` is in effect, the abort routine creates a core file and exits with code 127.

A few runtime errors just print an error message and call `exit(127)`, regardless of the status of `NVCOMPILER_TERM`. These are mainly errors such as specifying an invalid environment variable value where a traceback would not be useful.

If it appears that `abort()` does not generate core files on a Linux system, be sure to `unlimit the coredumpsize`. You can do this in these ways:

- ▶ Using `csch`:

```
% limit coredumpsize unlimited
% setenv NVCOMPILER_TERM abort
```

- ▶ Using `bash`, `sh`, `zsh`, or `ksh`:

```
$ ulimit -c unlimited
$ export NVCOMPILER_TERM=abort
```

To debug a core file with `gdb`, invoke `gdb` with the `--core` option. For example, to view a core file named "core" for a program named "a.out":

```
$ gdb --core=core a.out
```

For more information on why to use this variable, refer to [Stack Traceback and JIT Debugging](#).

10.3.11. NVCOMPILER_TERM_DEBUG

The `NVCOMPILER_TERM_DEBUG` variable may be set to override the default behavior when `NVCOMPILER_TERM` is set to `debug`.

The value of `NVCOMPILER_TERM_DEBUG` should be set to the command line used to invoke the program. For example:

```
gdb --quiet --pid %d
```

The first occurrence of %d in the NVCOMPILER_TERM_DEBUG string is replaced by the process id. The program named in the NVCOMPILER_TERM_DEBUG string must be found on the current PATH or specified with a full path name.

10.3.12. PWD

The PWD variable allows you to display the current directory.

10.3.13. STATIC_RANDOM_SEED

You can use STATIC_RANDOM_SEED to force the seed returned by the Fortran 90/95 RANDOM_SEED intrinsic to be constant. The first call to RANDOM_SEED without arguments resets the random seed to a default value, then advances the seed by a variable amount based on time. Subsequent calls to RANDOM_SEED without arguments reset the random seed to the same initial value as the first call. Unless the time is exactly the same, each time a program is run a different random number sequence is generated. Setting the environment variable STATIC_RANDOM_SEED to YES forces the seed returned by RANDOM_SEED to be constant, thereby generating the same sequence of random numbers at each execution of the program.

10.3.14. TMP

You can use TMP to specify the directory to use for placement of any temporary files created during execution of the NVIDIA HPC Compilers. This variable is interchangeable with TMPDIR.

10.3.15. TMPDIR

You can use TMPDIR to specify the directory to use for placement of any temporary files created during execution of the NVIDIA HPC Compilers.

10.4. Using Environment Modules on Linux

On Linux, if you use the Environment Modules package, that is, the module load command, the NVIDIA HPC Compilers include a script to set up the appropriate module files.

Assuming your installation base directory is /opt/nvidia/hpc_sdk, and your **MODULEPATH** environment variable is /usr/local/Modules/modulefiles, execute this command:

```
% /opt/nvidia/hpc_sdk/linux86-64/20.9/etc/modulefiles/nvcompilers.module.install
\
  -all -install /usr/local/Modules/modulefiles

% /opt/nvidia/hpc_sdk/linuxpower/20.9/etc/modulefiles/nvcompilers.module.install
\
  -all -install /usr/local/Modules/modulefiles
```

This command creates module files for all installed versions of the NVIDIA HPC Compilers. You must have write permission to the `modulefiles` directory to enable the module commands:

```
% module load nvcompilers/20.9
% module load nvcompilers/20.9
```

where `"nvcompilers/20.9"` also uses the 64-bit compilers.

To see what versions are available, use this command:

```
% module avail nvcompilers
```

The `module load` command sets or modifies the environment variables as indicated in the following table.

This Environment Variable...	Is set or modified by the module load command
CC	Full path to <code>nvc</code>
CPP	Full path to <code>nvprepro</code>
CXX	Path to <code>nvc++</code>
FC	Full path to <code>nvfortran</code>
LD_LIBRARY_PATH	Prepends the NVIDIA HPC Compilers library directory
MANPATH	Prepends the NVIDIA HPC Compilers man page directory
PATH	Prepends the NVIDIA HPC Compilers <code>bin</code> directory
nvidia/hpc_sdk	The base installation directory



NVIDIA does not provide support for the Environment Modules package. For more information about the package, go to: <http://modules.sourceforge.net>.

10.5. Stack Traceback and JIT Debugging

When a programming error results in a runtime error message or an application exception, a program will usually exit, perhaps with an error message. The NVIDIA HPC Compilers runtime library includes a mechanism to override this default action and instead print a stack traceback, start a debugger, or, on Linux, create a core file for post-mortem debugging.

The stack traceback and just-in-time debugging functionality is controlled by an environment variable, `NVCOMPILER_TERM`, described in [NVCOMPILER_TERM](#). The runtime libraries use the value of `NVCOMPILER_TERM` to determine what action to take when a program abnormally terminates.

When the NVIDIA HPC Compilers runtime library detects an error or catches a signal, it calls the routine `nvcompiler_stop_here()` prior to generating a stack traceback or starting the debugger. The `nvcompiler_stop_here()` routine is a convenient spot to set a breakpoint when debugging a program.

Chapter 11.

DISTRIBUTING FILES – DEPLOYMENT

Once you have successfully built, debugged and tuned your application, you may want to distribute it to users who need to run it on a variety of systems. This section addresses how to effectively distribute applications built using NVIDIA HPC Compilers. The application must be installed in such a way that it executes accurately on a system other than the one on which it was built, and which may be configured differently.

11.1. Deploying Applications on Linux

To successfully deploy your application on Linux, some of the issues to consider include:

- ▶ Runtime Libraries
- ▶ 64-bit Linux Systems
- ▶ Redistribution of Files

11.1.1. Runtime Library Considerations

On Linux systems, the system runtime libraries can be linked to an application either statically or dynamically. For example, for the C runtime library, `libc`, you can use either the static version `libc.a` or the shared object version `libc.so`. If the application is intended to run on Linux systems other than the one on which it was built, it is generally safer to use the shared object version of the library. This approach ensures that the application uses a version of the library that is compatible with the system on which the application is running. Further, it works best when the application is linked on a system that has an equivalent or earlier version of the system software than the system on which the application will be run.



Building on a newer system and running the application on an older system may not produce the desired output.

To use the shared object version of a library, the application must also link to shared object versions of the NVIDIA HPC Compilers runtime libraries. To execute an application built in such a way on a system on which NVIDIA HPC Compilers are

not installed, those shared objects must be available. To build using the shared object versions of the runtime libraries, use the `-Bdynamic` option, as shown here:

```
$ nvfortran -Bdynamic myprog.f90
```

11.1.2. 64-bit Linux Considerations

On 64-bit Linux systems, 64-bit applications that use the `-mcmodel=medium` option sometimes cannot be successfully linked statically. Therefore, users with executables built with the `-mcmodel=medium` option may need to use shared libraries, linking dynamically. Also, runtime libraries built using the `-fpic` option use 32-bit offsets, so they sometimes need to reside near other runtime `libs` in a shared area of Linux program memory.



If your application is linked dynamically using shared objects, then the shared object versions of the NVIDIA HPC Compilers runtime are required.

11.1.3. Linux Redistributable Files

The method for installing the shared object versions of the runtime libraries required for applications built with NVIDIA HPC Compilers is manual distribution.

When the NVIDIA HPC Compilers are installed, there are directories that have a name that begins with `REDIST`; these directories contain the redistributed shared object libraries. These may be redistributed by licensed NVIDIA HPC Compilers users under the terms of the End-User License Agreement.

11.1.4. Restrictions on Linux Portability

You cannot expect to be able to run an executable on any given Linux machine. Portability depends on the system you build on as well as how much your program uses system routines that may have changed from Linux release to Linux release. For example, an area of significant change between some versions of Linux is in `libpthread.so` and `libnuma.so`. NVIDIA HPC Compilers use these dynamically linked libraries for the options `-acc` (OpenACC), `-mp` (OpenMP) and `-Mconcur` (multicore auto-parallel). Statically linking these libraries may not be possible, or may result in failure at execution.

Typically, portability is supported for forward execution, meaning running a program on the same or a later version of Linux. But not for backward compatibility, that is, running on a prior release. For example, a user who compiles and links a program under RHEL 7.2 should not expect the program to run without incident on a RHEL 5.2 system, an earlier Linux version. It *may* run, but it is less likely. Developers might consider building applications on earlier Linux versions for wider usage. Dynamic linking of Linux and gcc system routines on the platform executing the program can also reduce problems.

11.1.5. Licensing for Redistributable (REDIST) Files

The files in the REDIST directories may be redistributed under the terms of the End-User License Agreement for the product in which they were included.

Chapter 12.

INTER-LANGUAGE CALLING

This section describes inter-language calling conventions for C, C++, and Fortran programs using the HPC compilers. Fortran 2003 ISO_C_Binding provides a mechanism to support the interoperability with C. This includes the ISO_C_Binding intrinsic module, binding labels, and the BIND attribute. In the absence of this mechanism, the following sections describe how to call a Fortran function or subroutine from a C or C++ program and how to call a C or C++ function from a Fortran program.

This section provides examples that use the following options related to inter-language calling.

`-c` `-Mnomain` `-Miface` `-Mupcase`

12.1. Overview of Calling Conventions

This section includes information on the following topics:

- ▶ Functions and subroutines in Fortran, C, and C++
- ▶ Naming and case conversion conventions
- ▶ Compatible data types
- ▶ Argument passing and special return values
- ▶ Arrays and indexes

The sections [Inter-language Calling Considerations](#) through [Example – C++ Calling Fortran](#) describe how to perform inter-language calling using the Linux or Win64 convention.

12.2. Inter-language Calling Considerations

In general, when argument data types and function return values agree, you can call a C or C++ function from Fortran as well as call a Fortran function from C or C++. When data types for arguments do not agree, you may need to develop custom mechanisms to handle them. For example, the Fortran `COMPLEX` type has a matching type in C99

but does not have a matching type in C89; however, it is still possible to provide inter-language calls but there are no general calling conventions for such cases.



- ▶ If a C++ function contains objects with constructors and destructors, calling such a function from either C or Fortran is not possible unless the initialization in the main program is performed from a C++ program in which constructors and destructors are properly initialized.
- ▶ In general, you can call a C or Fortran function from C++ without problems as long as you use the extern "C" keyword to declare the function in the C++ program. This declaration prevents name mangling for the C function name. If you want to call a C++ function from C or Fortran, you also have to use the extern "C" keyword to declare the C++ function. This keeps the C++ compiler from mangling the name of the function.
- ▶ You can use the __cplusplus macro to allow a program or header file to work for both C and C++. For example, the following defines in the header file stdio.h allow this file to work for both C and C++.

```
#ifndef _STDIO_H
#define _STDIO_H
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
.
. /* Functions and data types defined... */
.
#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif
```

- ▶ C++ member functions cannot be declared extern, since their names will always be mangled. Therefore, C++ member functions cannot be called from C or Fortran.

12.3. Functions and Subroutines

Fortran, C, and C++ define functions and subroutines differently.

For a Fortran program calling a C or C++ function, observe the following return value convention:

- ▶ When a C or C++ function returns a value, call it from Fortran as a function.
- ▶ When a C or C++ function does not return a value, call it as a subroutine.

For a C/C++ program calling a Fortran function, the call should return a similar type. [Table 23, Fortran and C/C++ Data Type Compatibility](#), lists compatible types. If the call is to a Fortran subroutine, or a Fortran CHARACTER function, or a Fortran COMPLEX function, call it from C/C++ as a function that returns void. The exception to this convention is when a Fortran subroutine has alternate returns; call such a subroutine from C/C++ as a function returning int whose value is the value of the integer expression specified in the alternate RETURN statement.

12.4. Upper and Lower Case Conventions, Underscores

By default on Linux and Win64 systems, all Fortran symbol names are converted to lower case. C and C++ are case sensitive, so upper-case function names stay upper-case. When you use inter-language calling, you can either name your C/C++ functions with lower-case names, or invoke the Fortran compiler command with the option `-Mupcase`, in which case it will not convert symbol names to lower-case.

When programs are compiled using one of the HPC Fortran compilers on Linux and Win64 systems, an underscore is appended to Fortran global names (names of functions, subroutines and common blocks). This mechanism distinguishes Fortran name space from C/C++ name space. Use these naming conventions:

- ▶ If you call a C/C++ function from Fortran, you should rename the C/C++ function by appending an underscore or use `C$PRAGMA C` in the Fortran program. For more information on `C$PRAGMA C`, refer to [#unique_178](#).
- ▶ If you call a Fortran function from C/C++, you should append an underscore to the Fortran function name in the calling program.

12.5. Compatible Data Types

Table 23 shows compatible data types between Fortran and C/C++. Table 24, [Fortran and C/C++ Representation of the COMPLEX Type](#) shows how the Fortran COMPLEX type may be represented in C/C++.



Tip If you can make your function/subroutine parameters as well as your return values match types, you should be able to use inter-language calling.

Table 23 Fortran and C/C++ Data Type Compatibility

Fortran Type (lower case)	C/C++ Type	Size (bytes)
character x	char x	1
character*n x	char x[n]	n
real x	float x	4
real*4 x	float x	4
real*8 x	double x	8
double precision	double x	8
integer x	int x	4
integer*1 x	signed char x	1
integer*2 x	short x	2

Fortran Type (lower case)	C/C++ Type	Size (bytes)
integer*4 x	int x	4
integer*8 x	long long x	8
logical x	int x	4
logical*1 x	char x	1
logical*2 x	short x	2
logical*4	int x	4
logical*8	long x	8

Table 24 Fortran and C/C++ Representation of the `COMPLEX` Type

Fortran Type (lower case)	C/C++ Type	Size (bytes)
complex x	struct {float r,i;} x; float complex x;	8 8
complex*8 x	struct {float r,i;} x; float complex x;	8 8
double complex x	struct {double dr,di;} x; double complex x;	16 16
complex *16 x	struct {double dr,di;} x; double complex x;	16 16



For C/C++, the `complex` type implies C99 or later.

12.5.1. Fortran Named Common Blocks

A named Fortran common block can be represented in C/C++ by a structure whose members correspond to the members of the common block. The name of the structure in C/C++ must have the added underscore. For example, here is a Fortran common block:

```
INTEGER I
COMPLEX C
DOUBLE COMPLEX CD
DOUBLE PRECISION D
COMMON /COM/ i, c, cd, d
```

This Fortran Common Block is represented in C with the following equivalent:

```
extern struct {
    int i;
    struct {float real, imag;} c;
    struct {double real, imag;} cd;
    double d;
} com_;
```

This same Fortran Common Block is represented in C++ with the following equivalent:

```
extern "C" struct {
```

```
int i;
struct {float real, imag;} c;
struct {double real, imag;} cd;
double d;
} com_;
```



Tip For global or external data sharing, `extern "C"` is not required.

12.6. Argument Passing and Return Values

In Fortran, arguments are passed by reference, that is, the address of the argument is passed, rather than the argument itself. In C/C++, arguments are passed by value, except for strings and arrays, which are passed by reference. Due to the flexibility provided in C/C++, you can work around these differences. Solving the parameter passing differences generally involves intelligent use of the `&` and `*` operators in argument passing when C/C++ calls Fortran and in argument declarations when Fortran calls C/C++.

For strings declared in Fortran as type `CHARACTER`, an argument representing the length of the string is also passed to a calling function.

On the following systems, the compiler places the length argument(s) at the end of the parameter list, following the other formal arguments:

- ▶ On Linux systems
- ▶ On Win64 systems, except when using the option `-Miface=ceref`

The length argument is passed by value, not by reference.

12.6.1. Passing by Value (%VAL)

When passing parameters from a Fortran subprogram to a C/C++ function, it is possible to pass by value using the `%VAL` function. If you enclose a Fortran parameter with `%VAL()`, the parameter is passed by value. For example, the following call passes the integer `i` and the logical `bvar` by value.

```
integer*1 i
logical*1 bvar
call cvalue (%VAL(i), %VAL(bvar))
```

12.6.2. Character Return Values

[Functions and Subroutines](#) describes the general rules for return values for C/C++ and Fortran inter-language calling. There is a special return value to consider. When a Fortran function returns a character, two arguments need to be added at the beginning of the C/C++ calling function's argument list:

- ▶ The address of the return character or characters
- ▶ The length of the return character

The following example illustrates the extra parameters, `tmp` and `10`, supplied by the caller:

Character Return Parameters

```
! Fortran function returns a character
CHARACTER*(*) FUNCTION CHF(C1,I)
  CHARACTER*(*) C1
  INTEGER I
END
```

```
/* C declaration of Fortran function */
extern void chf_();
char tmp[10];
char c1[9];
int i;
chf_(tmp, 10, c1, &i, 9);
```

If the Fortran function is declared to return a character value of constant length, for example `CHARACTER*4 FUNCTION CHF()`, the second extra parameter representing the length must still be supplied, but is not used.



The value of the character function is not automatically NULL-terminated.

12.6.3. Complex Return Values

When a Fortran function returns a complex value, an argument needs to be added at the beginning of the C/C++ calling function's argument list; this argument is the address of the complex return value. [COMPLEX Return Values](#) illustrates the extra parameter, `cplx`, supplied by the caller.

COMPLEX Return Values

```
COMPLEX FUNCTION CF(C, I)
  INTEGER I
  . . .
END
```

```
extern void cf_();
typedef struct {float real, imag;} cplx;
cplx c1;
int i;
cf_(&c1, &i);
```

12.7. Array Indices

C/C++ arrays and Fortran arrays use different default initial array index values. By default, arrays in C/C++ start at 0 and arrays in Fortran start at 1. If you adjust your array comparisons so that a Fortran second element is compared to a C/C++ first element, and adjust similarly for other elements, you should not have problems working with this difference. If this is not satisfactory, you can declare your Fortran arrays to start at zero.

Another difference between Fortran and C/C++ arrays is the storage method used. Fortran uses column-major order and C/C++ uses row-major order. For one-dimensional arrays, this poses no problems. For two-dimensional arrays, where there are an equal number of rows and columns, row and column indexes can simply be reversed. For

arrays other than single dimensional arrays, and square two-dimensional arrays, inter-language function mixing is not recommended.

12.8. Examples

This section contains examples that illustrate inter-language calling.

12.8.1. Example – Fortran Calling C



There are other solutions to calling C from Fortran than the one presented in this section. For example, you can use the `iso_c_binding` intrinsic module which NVIDIA does support. For more information on this module and for examples of how to use it, search the web using the keyword `iso_c_binding`.

C function `f2c_func_` shows a C function that is called by the Fortran main program shown in **Fortran Main Program `f2c_main.f`**. Notice that each argument is defined as a pointer, since Fortran passes by reference. Also notice that the C function name uses all lower-case and a trailing `_`.

Fortran Main Program `f2c_main.f`

```
logical*1 bool1
character letter1
integer*4 numint1, numint2
real numfloat1
double precision numdoub1
integer*2 numshor1
external f2c_func

call f2c_func(bool1, letter1, numint1, numint2, numfloat1, numdoub1, numshor1)

write( *, "(L2, A2, I5, I5, F6.1, F6.1, I5)")
+ bool1, letter1, numint1, numint2, numfloat1, numdoub1, numshor1

end
```

C function `f2c_func_`

```
#define TRUE 0xff
#define FALSE 0
void f2c_func_( bool1, letter1, numint1, numint2, numfloat1,\
  numdoub1, numshor1, len_letter1)
char *bool1, *letter1;
int *numint1, *numint2;
float *numfloat1;
double *numdoub1;
short *numshor1;
int len_letter1;
{
  *bool1 = TRUE;  *letter1 = 'v';
  *numint1 = 11;  *numint2 = -44;
  *numfloat1 = 39.6 ;
  *numdoub1 = 39.2;
  *numshor1 = 981;
}
```

Compile and execute the program `f2c_main.f` with the call to `f2c_func_` using the following command lines:

```
$ nvc -c f2c_func.c
$ nvfortran f2c_func.o f2c_main.f
```

Executing the `a.out` file should produce the following output:

```
T v 11 -44 39.6 39.2 981
```

12.8.2. Example – C Calling Fortran

The example [C Main Program `c2f_main.c`](#) shows a C main program that calls the Fortran subroutine shown in [Fortran Subroutine `c2f_sub.f`](#).

- ▶ Each call uses the `&` operator to pass by reference.
- ▶ The call to the Fortran subroutine uses all lower-case and a trailing `"_"`.

C Main Program `c2f_main.c`

```
void main () {
    char bool1, letter1;
    int numint1, numint2;
    float numfloat1;
    double numdoubl;
    short numshor1;
    extern void c2f_func();
    c2f_sub_(&bool1,&letter1,&numint1,&numint2,&numfloat1,&numdoubl,&numshor1, 1);
    printf(" %s %c %d %d %3.1f %.0f %d\n",
        bool1?"TRUE":"FALSE", letter1, numint1, numint2,
        numfloat1, numdoubl, numshor1);
}
```

Fortran Subroutine `c2f_sub.f`

```
subroutine c2f_func ( bool1, letter1, numint1, numint2,
+ numfloat1, numdoubl, numshor1)
    logical*1 bool1
    character letter1
    integer numint1, numint2
    double precision numdoubl
    real numfloat1
    integer*2 numshor1

    bool1 = .true.
    letter1 = "v"
    numint1 = 11
    numint2 = -44
    numdoubl = 902
    numfloat1 = 39.6
    numshor1 = 299
    return
end
```

To compile this Fortran subroutine and C program, use the following commands:

```
$ nvc -c c2f_main.c
$ nvfortran -Mnomain c2f_main.o c2_sub.f
```

Executing the resulting `a.out` file should produce the following output:

```
TRUE v 11 -44 39.6 902 299
```


12.8.3. Example – C++ Calling C

[C++ Main Program cp2c_main.C](#) Calling a C Function shows a C++ main program that calls the C function shown in [Simple C Function c2cp_func.c](#).

C++ Main Program cp2c_main.C Calling a C Function

```
extern "C" void cp2c_func(int n, int m, int *p);
#include <iostream>
main()
{
    int a,b,c;
    a=8;
    b=2;
    c=0;
    cout << "main: a = "<<a<<" b = "<<b<<"ptr c = "<<hex<<&c<< endl;
    cp2c_func(a,b,&c);
    cout << "main: res = "<<c<<endl;
}
```

Simple C Function c2cp_func.c

```
void cp2c_func(num1, num2, res)
int num1, num2, *res;
{
    printf("func: a = %d b = %d ptr c = %x\n",num1,num2,res);
    *res=num1/num2;
    printf("func: res = %d\n",*res);
}
```

To compile this C function and C++ main program, use the following commands:

```
$ nvc -c cp2c_func.c
$ nvc++ cp2c_main.C cp2c_func.o
```

Executing the resulting a.out file should produce the following output:

```
main: a = 8 b = 2 ptr c = 0xbffffb94
func: a = 8 b = 2 ptr c = bffffb94
func: res = 4
main: res = 4
```

12.8.4. Example – C Calling C ++

The example in [C Main Program c2cp_main.c](#) Calling a C++ Function shows a C main program that calls the C++ function shown in [Simple C++ Function c2cp_func.C with Extern C](#).

C Main Program c2cp_main.c Calling a C++ Function

```
extern void c2cp_func(int a, int b, int *c);
#include <stdio.h>
main() {
    int a,b,c;
    a=8; b=2;
    printf("main: a = %d b = %d ptr c = %x\n",a,b,&c);
    c2cp_func(a,b,&c);
    printf("main: res = %d\n",c);
}
```

Simple C++ Function c2cp_func.C with Extern C

```
#include <iostream>
extern "C" void c2cp_func(int num1,int num2,int *res)
```

```
{
cout << "func: a = "<<num1<<" b = "<<num2<<"ptr c = "<<res<<endl;
*res=num1/num2;
cout << "func: res = "<<res<<endl;
}
```

To compile this C function and C++ main program, use the following commands:

```
$ nvc -c c2cp_main.c
$ nvc++ c2cp_main.o c2cp_func.C
```

Executing the resulting a.out file should produce the following output:

```
main: a = 8 b = 2 ptr c = 0xbffffb94
func: a = 8 b = 2 ptr c = bffffb94
func: res = 4
main: res = 4
```



You cannot use the extern "C" form of declaration for an object's member functions.

12.8.5. Example – Fortran Calling C++

The Fortran main program shown in [Fortran Main Program f2cp_main.f](#) calling a C++ function calls the C++ function shown in [C++ function f2cp_func.C](#).

Notice:

- ▶ Each argument is defined as a pointer in the C++ function, since Fortran passes by reference.
- ▶ The C++ function name uses all lower-case and a trailing "_":

Fortran Main Program f2cp_main.f calling a C++ function

```
logical*1 bool1
character letter1
integer*4 numint1, numint2
real numfloat1
double precision numdoubl
integer*2 numshor1
external f2cpfunc
call f2cp_func (bool1, letter1, numint1,
+ numint2, numfloat1, numdoubl, numshor1)
write( *, "(L2, A2, I5, I5, F6.1, F6.1, I5)")
+ bool1, letter1, numint1, numint2, numfloat1,
+ numdoubl, numshor1
end
```

C++ function f2cp_func.C

```
#define TRUE 0xff
#define FALSE 0
extern "C"
{
extern void f2cp_func_ (
char *bool1, *letter1,
int *numint1, *numint2,
float *numfloat1,
double *numdoubl,
short *numshort1,
int len_letter1)
{
*bool1 = TRUE;      *letter1 = 'v';
*numint1 = 11;      *numint2 = -44;
```

```
*numfloat1 = 39.6; *numdoub1 = 39.2;   *numshort1 = 981;
}
}
```

Assuming the Fortran program is in a file `fmain.f`, and the C++ function is in a file `cpfunc.C`, create an executable, using the following command lines:

```
$ nvc++ -c f2cp_func.C
$ nvfortran f2cp_func.o f2cp_main.f -c++libs
```

Executing the `a.out` file should produce the following output:

```
T v 11 -44 39.6 39.2 981
```

12.8.6. Example – C++ Calling Fortran

Fortran Subroutine `cp2f_func.f` shows a Fortran subroutine called by the C++ main program shown in C++ main program `cp2f_main.C`. Notice that each call uses the `&` operator to pass by reference. Also notice that the call to the Fortran subroutine uses all lower-case and a trailing `_`:

C++ main program `cp2f_main.C`

```
#include <iostream>
extern "C" { extern void cp2f_func_(char *,char *,int *,int *,
    float *,double *,short *); }
main ()
{
    char bool1, letter1;
    int numint1, numint2;
    float numfloat1;
    double numdoub1;
    short numshor1;

    cp2f_func(&bool1,&letter1,&numint1,&numint2,&numfloat1, &numdoub1,&numshor1);
    cout << " bool1 = ";
    bool1?cout << "TRUE " :cout << "FALSE "; cout <<endl;
    cout << " letter1 = " << letter1 <<endl;
    cout << " numint1 = " << numint1 <<endl;
    cout << " numint2 = " << numint2 <<endl;
    cout << " numfloat1 = " << numfloat1 <<endl;
    cout << " numdoub1 = " << numdoub1 <<endl;
    cout << " numshor1 = " << numshor1 <<endl;
}
```

Fortran Subroutine `cp2f_func.f`

```
subroutine cp2f_func ( bool1, letter1, numint1,
+ numint2, numfloat1, numdoub1, numshor1)
logical*1 bool1
character letter1
integer numint1, numint2
double precision numdoub1
real numfloat1
integer*2 numshor1
bool1 = .true. ; letter1 = "v"
numint1 = 11 ; numint2 = -44
numdoub1 = 902 ; numfloat1 = 39.6 ; numshor1 = 299
return
end
```

To compile this Fortran subroutine and C++ program, use the following command lines:

```
$ nvfortran -c cp2f_func.f
$ nvc++ cp2f_func.o cp2f_main.C -fortranlibs
```

Executing this C++ main should produce the following output:

```
bool1 = TRUE letter1 = v numint1 = 11 numint2 = -44 numfloat1 = 39.6 numdoub1 = 902  
numshor1 = 299
```



You must explicitly link in the NVFORTRAN runtime support libraries when linking nvfortran-compiled program units into C++ or C main programs.

Chapter 13.

PROGRAMMING CONSIDERATIONS FOR 64-BIT ENVIRONMENTS

NVIDIA provides 64-bit compilers for 64-bit Linux operating systems running on x86-64 (Linux_x86_64), OpenPOWER (Linux_ppc64) and Arm Server (Linux_aarch64) architectures. You can use these compilers to create programs that use 64-bit memory addresses. The GNU toolchain on 64-bit Linux systems implements an option to control 32-bit vs 64-bit code generation, as described in [Large Static Data in Linux](#). This section describes the specifics of how to use the NVIDIA compilers to make use of 64-bit memory addressing.



The NVIDIA HPC compilers themselves are 64-bit applications which can only run on 64-bit CPUs running 64-bit Operating Systems.

This section describes how to use the following options related to 64-bit programming.

<code>-fPIC</code>	<code>-mmodel=medium</code>	<code>-Mlarge_arrays</code>
<code>-i8</code>	<code>-Mlargeaddressaware</code>	

13.1. Data Types in the 64-Bit Environment

The size of some data types can differ across 64-bit environments. This section describes the major differences.

13.1.1. C++ and C Data Types

On 64-bit Linux operating systems, the size of an int is 4 bytes, a long is 8 bytes, a long long is 8 bytes, and a pointer is 8 bytes.

13.1.2. Fortran Data Types

In Fortran, the default size of the INTEGER type is 4 bytes. The `-i8` compiler option may be used to make the default size of all INTEGER data in the program 8 bytes.

When using the `-Mlarge_arrays` option, described in [64-Bit Array Indexing](#), any 4-byte INTEGER variables that are used to index arrays are silently promoted by the compiler to 8 bytes. This promotion can lead to unexpected consequences, so 8-byte INTEGER variables are recommended for array indexing when using the option `-Mlarge_arrays`.

13.2. Large Static Data in Linux

64-bit Linux operating systems support two different memory models. The default model used by the NVIDIA HPC compilers on `Linux_x86_64` and `Linux_aarch64` targets is the small memory model, which can be specified using `-mcmodel=small`. This is the 32-bit model, which limits the size of code plus statically allocated data, including system and user libraries, to 2GB. The medium memory model, specified by `-mcmodel=medium`, allows combined code and static data areas (`.text` and `.bss` sections) larger than 2GB and is the default on `Linux_ppc64le` targets. The `-mcmodel=medium` option must be used on both the compile command and the link command in order to take effect.

There are implications to using `-mcmodel=medium`. The generated code requires increased addressing overhead to support the large data range. This can affect performance, though the compilers seek to minimize the added overhead through careful instruction selection and optimization. Also, on `Linux_x86_64` and `Linux_aarch64` platforms it is not possible to mix `-fpic` and `-mcmodel=medium` when compiling an object file, which means that it is not possible to include files compiled with `-mcmodel=medium` in shared object files for dynamic linking.

13.3. Large Dynamically Allocated Data

Dynamically allocated data objects in programs compiled by the NVIDIA HPC compilers can be larger than 2GB. No special compiler options are required to enable this functionality. The size of the allocation is only limited by the system. However, to correctly access dynamically allocated arrays with more than 2G elements you should use the `-Mlarge_arrays` option, described in the following section.

13.4. 64-Bit Array Indexing

The NVIDIA Fortran compilers provide an option, `-Mlarge_arrays`, that enables 64-bit indexing of arrays. This means that, as necessary, 64-bit INTEGER constants and variables are used to index arrays.



In the presence of `-Mlarge_arrays`, the compiler may silently promote 32-bit integers to 64 bits, which can have unexpected side effects.

On 64-bit Linux, the `-Mlarge_arrays` option also enables single static data objects larger than 2 GB. This option is the default in the presence of `-mcmodel=medium`.

13.5. Compiler Options for 64-bit Programming

The usual switches that apply to 64-bit programmers seeking to increase the data range of their applications are in the following table.

Table 25 64-bit Compiler Options

Option	Purpose	Considerations
-mmodel=medium	Allow for data declarations larger than 2GB. Default on Linux_ppc64le.	On Linux_x86_64 and Linux_aarch64, cannot be used with -fpic. Objects cannot be put into shared libraries.
-Mlarge_arrays	Perform all array-location-to-address calculations using 64-bit integer arithmetic.	Slightly slower execution. Is implicit with -mmodel=medium. Can be used with option -mmodel=small.
-fpic	Position independent code. Necessary for shared libraries.	Dynamic linking restricted to a 32-bit offset. External symbol references should refer to other shared lib routines, rather than the program calling them.
-i8	All INTEGER functions, data, and constants not explicitly declared INTEGER*4 are assumed to be INTEGER*8.	Users should take care to explicitly declare INTEGER functions as INTEGER*4.

The following table summarizes the limits of these programming models under the specified conditions. The compiler options you use vary by processor.

Table 26 Effects of Options on Memory and Array Sizes


Condition	Addr. Math		Max Size Gbytes		
	A	I	AS	DS	TS
64-bit addr limited by option -mmodel=small	64	32	2	2	2
-fpic incompatible with -mmodel=medium	64	32	2	2	2
Enable full support for 64-bit data addressing	64	64	>2	>2	>2

A	Address Type – size in bits of data used for address calculations, 64-bits.
I	Index Arithmetic -bit-size of data used to index into arrays and other aggregate data structures. If 32-bit, total range of any single data object is limited to 2GB.
AS	Maximum Array Size - the maximum size in gigabytes of any single data object.
DS	Maximum Data Size - max size in gigabytes combined of all data objects in .bss
TS	Maximum Total Size - max size in gigabytes, in aggregate, of all executable code and data objects in a running program.

13.6. Practical Limitations of Large Array Programming

The 64-bit addressing capability of 64-bit Linux environments can cause unexpected issues when data sizes are enlarged significantly. The following table describes the most common occurrences of practical limitations of large array programming.

Table 27 64-Bit Limitations

array initialization	Initializing a large array with a data statement may result in very large assembly and object files, where a line of assembler source is required for each element in the initialized array. Compilation and linking can be very time consuming as well. To avoid this issue, consider initializing large arrays in a loop at runtime rather than in a data statement.
stack space	Stack space can be a problem for data that is stack-based. On Linux, stack size is increased in your shell environment. If setting stacksize to unlimited is not large enough, try setting the size explicitly: <pre>limit stacksize new_size ! in csh</pre> <pre>ulimit -s new_size ! in bash</pre>
page swapping	If your executable is much larger than the physical size of memory, page swapping can cause it to run dramatically slower; it may even fail. This is not a compiler problem. Try smaller data sets to determine whether or not a problem is due to page thrashing.
configured space	Be sure your Linux system is configured with swap space sufficiently large to support the data sets used in your application(s). If your memory+swap space is not sufficiently large, your application will likely encounter a segmentation fault at runtime.
support for large address offsets in object file format	Arrays that are not dynamically allocated are limited by how the compiler can express the 'distance' between them when generating code. A field in the object file stores this 'distance' value, which is limited to 32-bits on Linux with -mcmodel=small. It is 64-bits on Linux with -mcmodel=medium. <div>  Without the 64-bit offset support in the object file format, large arrays cannot be declared statically, or locally on the stack. </div>

13.7. Medium Memory Model and Large Array in C

Consider the following example, where the aggregate size of the arrays exceeds 2GB.

Medium Memory Model and Large Array in C

```
% cat bigadd.c
#include <stdio.h>
#define SIZE 600000000 /* > 2GB/4 */
static float a[SIZE], b[SIZE];
int
main()
{
```



```

long long i, n, m;
float c[SIZE]; /* goes on stack */
n = SIZE;
m = 0;
for (i = 0; i < n; i += 10000) {
    a[i] = i + 1;
    b[i] = 2.0 * (i + 1);
    c[i] = a[i] + b[i];
    m = i;
}
printf("a[0]=%g b[0]=%g c[0]=%g\n", a[0], b[0], c[0]);
printf("m=%lld a[%lld]=%g b[%lld]=%gc[%lld]=%g\n", m, m, a[m], m, b[m], m, c[m]);
return 0;
}

```

```
% nvc -mcmodel=medium -o bigadd bigadd.c
```

When SIZE is greater than 2G/4, and the arrays are of type float with 4 bytes per element, the size of each array is greater than 2GB. With nvc, using the `-mcmodel=medium` switch, a static data object can now be > 2GB in size. If you execute with these settings in your environment, you may see the following:

```
% bigadd
Segmentation fault
```

Execution fails because the stack size is not large enough. You can most likely correct this error by using the **limit stacksize** command to reset the stack size in your environment:

```
% limit stacksize 3000M
```



The command **limit stacksize unlimited** probably does not provide as large a stack as we are using in the [this example](#).

```
% bigadd
a[0]=1 b[0]=2 c[0]=3
n=599990000 a[599990000]=5.9999e+08 b[599990000]=1.19998e+09
c[599990000]=1.79997e+09
```

13.8. Medium Memory Model and Large Array in Fortran

The following example works with the NVFORTRAN compiler. It uses 64-bit addresses and index arithmetic when the `-mcmodel=medium` option is used.

Consider the following example:

Medium Memory Model and Large Array in Fortran

```

% cat mat.f
program mat
  integer i, j, k, size, l, m, n
  parameter (size=16000) ! >2GB
  parameter (m=size,n=size)
  real*8 a(m,n), b(m,n), c(m,n), d
  do i = 1, m
    do j = 1, n
      a(i,j)=10000.0D0*dble(i)+dble(j)
    
```

```

        b(i,j)=20000.0D0*dble(i)+dble(j)
    enddo
enddo
!$omp parallel
!$omp do
do i = 1, m
    do j = 1, n
        c(i,j) = a(i,j) + b(i,j)
    enddo
enddo
!$omp do
do i=1,m
    do j = 1, n
        d = 30000.0D0*dble(i)+dble(j)+dble(j)
        if (d .ne. c(i,j)) then
            print *, "err i=", i, "j=", j
            print *, "c(i,j)=", c(i,j)
            print *, "d=", d
            stop
        endif
    enddo
enddo
!$omp end parallel
print *, "M =", M, ", N =", N
print *, "c(M,N) = ", c(m,n)
end

```

When compiled with the NVFORTRAN compiler using `-mcmodel=medium`:

```

% nvfortran -Mfree -mp -o mat mat.f -i8 -mcmodel=medium
% setenv OMP_NUM_THREADS 2
% mat
M = 16000 , N = 16000
c(M,N) = 480032000.0000000

```

13.9. Large Array and Small Memory Model in Fortran

The following example uses large, dynamically-allocated arrays. The code is divided into a main and subroutine so you could put the subroutine into a shared library. Dynamic allocation of large arrays saves space in the size of executable and saves time initializing data.

Large Array and Small Memory Model in Fortran

```
% cat mat_allo.f90
```

```

program mat_allo
    integer i, j
    integer size, m, n
    parameter (size=16000)
    parameter (m=size,n=size)
    double precision, allocatable::a(:,,:),b(:,,:),c(:,,:)
    allocate(a(m,n), b(m,n), c(m,n))
    do i = 100, m, 1
        do j = 100, n, 1
            a(i,j) = 10000.0D0 * dble(i) + dble(j)
            b(i,j) = 20000.0D0 * dble(i) + dble(j)
        enddo
    enddo
    call mat_add(a,b,c,m,n)
    print *, "M =", m, ", N =", n
end

```

```
    print *, "c(M,N) = ", c(m,n)
end

subroutine mat_add(a,b,c,m,n)
    integer m, n, i, j
    double precision a(m,n), b(m,n), c(m,n)
    do i = 1, m
        do j = 1, n
            c(i,j) = a(i,j) + b(i,j)
        enddo
    enddo
    return
end

% nvfortran -o mat_allo mat_allo.f90 -i8 -Mlarge_arrays -mp -fast
```

Chapter 14.

C++ AND C INLINE ASSEMBLY AND INTRINSICS

The examples in this section are shown using x86-64 assembly instructions. Inline assembly is supported on OpenPOWER and Arm Server platforms as well, but is not documented in detail in this section.

14.1. Inline Assembly

Inline Assembly lets you specify machine instructions inside a "C" function. The format for an inline assembly instruction is this:

```
{ asm | __asm__ } ("string");
```

The `asm` statement begins with the `asm` or `__asm__` keyword. The `__asm__` keyword is typically used in header files that may be included in ISO "C" programs.

string is one or more machine specific instructions separated with a semi-colon (;) or newline (\n) character. These instructions are inserted directly into the compiler's assembly-language output for the enclosing function.

Some simple `asm` statements are:

```
asm ("cli");  
asm ("sti");
```

These `asm` statements disable and enable system interrupts respectively.

In the following example, the `eax` register is set to zero.

```
asm( "pushl %eax\n\t" "movl $0, %eax\n\t" "popl %eax");
```

Notice that `eax` is pushed on the stack so that it is not clobbered. When the statement is done with `eax`, it is restored with the `popl` instruction.

Typically a program uses macros that enclose `asm` statements. The following two examples use the interrupt constructs created previously in this section:

```
#define disableInt __asm__ ("cli");  
#define enableInt __asm__ ("sti");
```

14.2. Extended Inline Assembly

Inline Assembly explains how to use inline assembly to specify machine specific instructions inside a "C" function. This approach works well for simple machine operations such as disabling and enabling system interrupts. However, inline assembly has three distinct limitations:

1. The programmer must choose the registers required by the inline assembly.
2. To prevent register clobbering, the inline assembly must include push and pop code for registers that get modified by the inline assembly.
3. There is no easy way to access stack variables in an inline assembly statement.

Extended Inline Assembly was created to address these limitations. The format for extended inline assembly, also known as *extended asm*, is as follows:

```
{ asm | __asm__ } [ volatile | __volatile__ ]
("string" [: [output operands]] [: [input operands]] [: [clobberlist]]);
```

- ▶ Extended asm statements begin with the *asm* or *__asm__* keyword. Typically the *__asm__* keyword is used in header files that may be included by ISO "C" programs.
- ▶ An optional *volatile* or *__volatile__* keyword may appear after the *asm* keyword. This keyword instructs the compiler not to delete, move significantly, or combine with any other asm statement. Like *__asm__*, the *__volatile__* keyword is typically used with header files that may be included by ISO "C" programs.
- ▶ "string" is one or more machine specific instructions separated with a semi-colon (;) or newline (\n) character. The string can also contain operands specified in the [output operands], [input operands], and [clobber list]. The instructions are inserted directly into the compiler's assembly-language output for the enclosing function.
- ▶ The [output operands], [input operands], and [clobber list] items each describe the effect of the instruction for the compiler. For example:

```
asm( "movl %1, %%eax\n" "movl %%eax, %0" : "=r" (x) : "r" (y) : "%eax" );
```

where

"=r" (x) is an output operand.

"r" (y) is an input operand.

"%eax" is the clobber list consisting of one register, "%eax".

The notation for the output and input operands is a constraint string surrounded by quotes, followed by an expression, and surrounded by parentheses. The constraint string describes how the input and output operands are used in the asm "string". For example, "r" tells the compiler that the operand is a register. The "=" tells the compiler that the operand is write only, which means that a value is stored in an output operand's expression at the end of the asm statement.

Each operand is referenced in the asm "string" by a percent "%" and its number. The first operand is number 0, the second is number 1, the third is number 2, and so on. In the preceding example, "%0" references the output operand, and "%1" references the input operand. The asm "string" also contains "%%eax", which references machine register "%eax". Hard coded registers like "%eax" should be specified in the

clobber list to prevent conflicts with other instructions in the compiler's assembly-language output. *[output operands]*, *[input operands]*, and *[clobber list]* items are described in more detail in the following sections.

14.2.1. Output Operands

The *[output operands]* are an optional list of output constraint and expression pairs that specify the result(s) of the asm statement. An output constraint is a string that specifies how a result is delivered to the expression. For example, "*=r*" (*x*) says the output operand is a write-only register that stores its value in the "C" variable *x* at the end of the asm statement. An example follows:

```
int x;
void example()
{
    asm( "movl $0, %0" : "=r" (x) );
}
```

The previous example assigns 0 to the "C" variable *x*. For the function in this example, the compiler produces the following assembly. If you want to produce an assembly listing, compile the example with the `nvc -S` compiler option:

```
example:
..Dcfb0:
    pushq %rbp
..Dcfi0:
    movq %rsp, %rbp
..Dcfi1:
..ENL:
## lineno: 8
    movl $0, %eax
    movl %eax, x(%rip)
## lineno: 0
    popq %rbp
    ret
```

In the generated assembly shown, notice that the compiler generated two statements for the asm statement at line number 5. The compiler generated "*movl \$0, %eax*" from the asm "*string*". Also notice that *%eax* appears in place of "*%0*" because the compiler assigned the *%eax* register to variable *x*. Since item 0 is an output operand, the result must be stored in its expression (*x*).

In addition to write-only output operands, there are read/write output operands designated with a "+" instead of a "=". For example, "*+r*" (*x*) tells the compiler to initialize the output operand with variable *x* at the beginning of the asm statement.

To illustrate this point, the following example increments variable *x* by 1:

```
int x=1;
void example2()
{
    asm( "addl $1, %0" : "+r" (x) );
}
```

To perform the increment, the output operand must be initialized with variable *x*. The *read/write* constraint modifier ("*+*") instructs the compiler to initialize the output operand with its expression. The compiler generates the following assembly code for the `example2()` function:

```
example2:
..Dcfb0:
```

```

pushq %rbp
..Dcfi0:
movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 5
movl x(%rip), %eax
addl $1, %eax
movl %eax, x(%rip)
## lineno: 0
popq %rbp
ret

```

From the `example2()` code, two extraneous moves are generated in the assembly: one `movl` for initializing the output register and a second `movl` to write it to variable `x`. To eliminate these moves, use a memory constraint type instead of a register constraint type, as shown in the following example:

```

int x=1;
void example2()
{
    asm( "addl $1, %0" : "+m" (x) );
}

```

The compiler generates a memory reference in place of a memory constraint. This eliminates the two extraneous moves. Because the assembly uses a memory reference to variable `x`, it does not have to move `x` into a register prior to the `asm` statement; nor does it need to store the result after the `asm` statement. Additional constraint types are found in [Additional Constraints](#).

```

example2:
..Dcfb0:
pushq %rbp
..Dcfi0:
movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 5
addl $1, x(%rip)
## lineno: 0
popq %rbp
ret

```

The examples thus far have used only one output operand. Because extended `asm` accepts a list of output operands, `asm` statements can have more than one result, as shown in the following example:

```

void example4()
{
    int x=1; int y=2;
    asm( "addl $1, %1\n" "addl %1, %0": "+r" (x), "+m" (y) );
}

```

This example increments variable `y` by 1 then adds it to variable `x`. Multiple output operands are separated with a comma. The first output operand is item 0 ("`%0`") and the second is item 1 ("`%1`") in the `asm` "string". The resulting values for `x` and `y` are 4 and 3 respectively.

14.2.2. Input Operands

The *[input operands]* are an optional list of input constraint and expression pairs that specify what "C" values are needed by the `asm` statement. The input constraints specify

how the data is delivered to the asm statement. For example, "*r*" (*x*) says that the input operand is a register that has a copy of the value stored in "C" variable *x*. Another example is "*m*" (*x*) which says that the input item is the *memory* location associated with variable *x*. Other constraint types are discussed in [Additional Constraints](#). An example follows:

```
void example5()
{
    int x=1;
    int y=2;
    int z=3;
    asm( "addl %2, %1\n" "addl %2, %0" : "+r" (x), "+m" (y) : "r" (z) );
}
```

The previous example adds variable *z*, item 2, to variable *x* and variable *y*. The resulting values for *x* and *y* are 4 and 5 respectively.

Another type of input constraint worth mentioning here is the *matching constraint*. A matching constraint is used to specify an operand that fills both an input as well as an output role. An example follows:

```
int x=1;
void example6()
{
    asm( "addl $1, %1"
        : "=r" (x)
        : "0" (x) );
}
```

The previous example is equivalent to the *example2()* function shown in [Output Operands](#). The constraint/expression pair, "*0*" (*x*), tells the compiler to initialize output item 0 with variable *x* at the beginning of the *asm* statement. The resulting value for *x* is 2. Also note that "*%1*" in the asm "*string*" means the same thing as "*%0*" in this case. That is because there is only one operand with both an input and an output role.

Matching constraints are very similar to the *read/write* output operands mentioned in [Output Operands](#). However, there is one key difference between *read/write* output operands and *matching constraints*. The *matching constraint* can have an *input expression* that differs from its *output expression*.

The following example uses different values for the input and output roles:

```
int x;
int y=2;
void example7()
{
    asm( "addl $1, %1"
        : "=r" (x)
        : "0" (y) );
}
```

The compiler generates the following assembly for *example7()*:

```
example7:
..Dcfb0:
    pushq %rbp
..Dcfi0:
    movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 8
    movl y(%rip), %eax
    addl $1, %eax
    movl %eax, x(%rip)
```



```
## lineno: 0
popq %rbp
ret
```

Variable *x* gets initialized with the value stored in *y*, which is 2. After adding 1, the resulting value for variable *x* is 3.

Because *matching constraints* perform an input role for an output operand, it does not make sense for the output operand to have the read/write ("+") modifier. In fact, the compiler disallows *matching constraints* with read/write output operands. The output operand must have a write only ("=") modifier.

14.2.3. Clobber List

The *[clobber list]* is an optional list of strings that hold machine registers used in the asm "string". Essentially, these strings tell the compiler which registers may be clobbered by the asm statement. By placing registers in this list, the programmer does not have to explicitly save and restore them as required in traditional inline assembly (described in [Inline Assembly](#)). The compiler takes care of any required saving and restoring of the registers in this list.

Each machine register in the [clobber list] is a string separated by a comma. The leading '%' is optional in the register name. For example, "%eax" is equivalent to "eax". When specifying the register inside the asm "string", you must include two leading '%' characters in front of the name (for example, "%%eax"). Otherwise, the compiler will behave as if a bad input/output operand was specified and generate an error message.

An example follows:

```
void example8()
{
    int x;
    int y=2;
    asm( "movl %1, %%eax\n"
        "movl %1, %%edx\n"
        "addl %%edx, %%eax\n"
        "addl %%eax, %0"
        : "=r" (x)
        : "0" (y)
        : "eax", "edx" );
}
```

This code uses two hard-coded registers, *eax* and *edx*. It performs the equivalent of $3*y$ and assigns it to *x*, producing a result of 6.

In addition to machine registers, the clobber list may contain the following special flags:

"cc"

The asm statement may alter the control code register.

"memory"

The asm statement may modify memory in an unpredictable fashion.

When the "memory" flag is present, the compiler does not keep memory values cached in registers across the asm statement and does not optimize stores or loads to that memory. For example:

```
asm("call MyFunc":::"memory");
```

This asm statement contains a "memory" flag because it contains a call. The callee may otherwise clobber registers in use by the caller without the "memory" flag.

The following function uses extended asm and the "cc" flag to compute a power of 2 that is less than or equal to the input parameter n.

```
#pragma noline
int asmDivideConquer(int n)
{
    int ax = 0;
    int bx = 1;
    asm (
        "LogLoop:n"
        "cmp %2, %1n"
        "jnl Done:n"
        "inc %0n"
        "add %1,%1n"
        "jmp LogLoop:n"
        "Done:n"
        "dec %0n"
        : "+r" (ax), "+r" (bx) : "r" (n) : "cc");
    return ax;
}
```

The 'cc' flag is used because the asm statement contains some control flow that may alter the control code register. The #pragma noline statement prevents the compiler from inlining the asmDivideConquer() function. If the compiler inlines asmDivideConquer(), then it may illegally duplicate the labels LogLoop and Done in the generated assembly.

14.2.4. Additional Constraints

Operand constraints can be divided into four main categories:

- ▶ Simple Constraints
- ▶ Machine Constraints
- ▶ Multiple Alternative Constraints
- ▶ Constraint Modifiers

14.2.5. Simple Constraints

The simplest kind of constraint is a string of letters or characters, known as *Simple Constraints*, such as the "r" and "m" constraints introduced in [Output Operands](#). [Table 28](#) describes these constraints.

Table 28 Simple Constraints

Constraint	Description
whitespace	Whitespace characters are ignored.
E	An immediate floating point operand.
F	Same as "E".
g	Any general purpose register, memory, or immediate integer operand is allowed.
i	An immediate integer operand.
m	A memory operand. Any address supported by the machine is allowed.
n	Same as "i".
o	Same as "m".

Constraint	Description
p	An operand that is a valid memory address. The expression associated with the constraint is expected to evaluate to an address (for example, "p" (&x)).
r	A general purpose register operand.
X	Same as "g".
0,1,2,..9	Matching Constraint. See Output Operands for a description.

The following example uses the general or "g" constraint, which allows the compiler to pick an appropriate constraint type for the operand; the compiler chooses from a general purpose register, memory, or immediate operand. This code lets the compiler choose the constraint type for "y".

```
void example9()
{
    int x, y=2;
    asm( "movl %1, %0\n" : "=r"
        (x) : "g" (y) );
}
```

This technique can result in more efficient code. For example, when compiling `example9()` the compiler replaces the load and store of `y` with a constant 2. The compiler can then generate an immediate 2 for the `y` operand in the example. The assembly generated by `nvc` for our example is as follows:

```
example9:
..Dcfb0:
    pushq %rbp
..Dcfi0:
    movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 3
    movl $2, %eax
## lineno: 6
    popq %rbp
    ret
```

In this example, notice the use of \$2 for the "y" operand.

Of course, if `y` is always 2, then the immediate value may be used instead of the variable with the "i" constraint, as shown here:

```
void example10()
{
    int x;
    asm( "movl %1, %0\n"
        : "=r" (x)
        : "i" (2) );
}
```

Compiling `example10()` with `nvc` produces assembly similar to that produced for `example9()`.

14.2.6. Machine Constraints

Another category of constraints is *Machine Constraints*. The `x86_64` architectures has several classes of registers. To choose a particular class of register, you can use the `x86_64` machine constraints described in [Table 29](#).

Table 29 x86_64 Machine Constraints

Constraint	Description
a	a register (e.g., %al, %ax, %eax, %rax)
A	Specifies a or d registers. The d register holds the most significant bits and the a register holds the least significant bits.
b	b register (e.g., %bl, %bx, %ebx, %rbx)
c	c register (e.g., %cl, %cx, %ecx, %rcx)
C	Not supported.
d	d register (e.g., %dl, %dx, %edx, %rdx)
D	di register (e.g., %dil, %di, %edi, %rdi)
e	Constant in range of 0xffffffff to 0x7fffffff
f	Not supported.
G	Floating point constant in range of 0.0 to 1.0.
I	Constant in range of 0 to 31 (e.g., for 32-bit shifts).
J	Constant in range of 0 to 63 (e.g., for 64-bit shifts)
K	Constant in range of 0 to 127.
L	Constant in range of 0 to 65535.
M	Constant in range of 0 to 3 constant (e.g., shifts for lea instruction).
N	Constant in range of 0 to 255 (e.g., for out instruction).
q	Same as "r" simple constraint.
Q	Same as "r" simple constraint.
R	Same as "r" simple constraint.
S	si register (e.g., %sil, %si, %edi, %rsi)
t	Not supported.
u	Not supported.
x	XMM SSE register
y	Not supported.
Z	Constant in range of 0 to 0x7fffffff.

The following example uses the "x" or XMM register constraint to subtract c from b and store the result in a.

```
double example11()
{
    double a;
    double b = 400.99;
    double c = 300.98;
    asm ( "subpd %2, %0;"
        : "=x" (a)
        : "0" (b), "x" (c)
        );
    return a;
}
```

The generated assembly for this example is this:

```
example11:
..Dcfb0:
    pushq %rbp
..Dcfi0:
    movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 4
    movsd .C00128(%rip), %xmm1
    movsd .C00130(%rip), %xmm2
    movapd %xmm1, %xmm0
    subpd %xmm2, %xmm0;
## lineno: 10
## lineno: 11
    popq %rbp
    ret
```

If a specified register is not available, the nvc and nvc++ compilers issue an error message.

14.2.7. Multiple Alternative Constraints

Sometimes a single instruction can take a variety of operand types. For example, the x86-64 permits register-to-memory and memory-to-register operations. To allow this flexibility in inline assembly, use *multiple alternative constraints*. An *alternative* is a series of constraints for each operand.

To specify multiple alternatives, separate each alternative with a comma.

Table 30 Multiple Alternative Constraints

Constraint	Description
,	Separates each alternative for a particular operand.
?	Ignored
!	Ignored

The following example uses multiple alternatives for an add operation.

```
void example13()
{
    int x=1;
    int y=1;
    asm( "addl %1, %0\n"
        : "+ab,cd" (x)
        : "db,cam" (y) );
}
```

The preceding *example13()* has two alternatives for each operand: "ab,cd" for the output operand and "db,cam" for the input operand. Each operand must have the same number of alternatives; however, each alternative can have any number of constraints (for example, the output operand in *example13()* has two constraints for its second alternative and the input operand has three for its second alternative).

The compiler first tries to satisfy the left-most alternative of the first operand (for example, the output operand in *example13()*). When satisfying the operand, the compiler starts with the left-most constraint. If the compiler cannot satisfy an alternative with

this constraint (for example, if the desired register is not available), it tries to use any subsequent constraints. If the compiler runs out of constraints, it moves on to the next alternative. If the compiler runs out of alternatives, it issues an error similar to the one mentioned in *example12()*. If an alternative is found, the compiler uses the same alternative for subsequent operands. For example, if the compiler chooses the "c" register for the output operand in *example13()*, then it will use either the "a" or "m" constraint for the input operand.

14.2.8. Constraint Modifiers

Characters that affect the compiler's interpretation of a constraint are known as *Constraint Modifiers*. Two constraint modifiers, the "=" and the "+", were introduced in *Output Operands*. The following table summarizes each constraint modifier.

Table 31 Constraint Modifier Characters

Constraint Modifier	Description
=	This operand is write-only. It is valid for output operands only. If specified, the "=" must appear as the first character of the constraint string.
+	This operand is both read and written by the instruction. It is valid for output operands only. The output operand is initialized with its expression before the first instruction in the asm statement. If specified, the "+" must appear as the first character of the constraint string.
&	A constraint or an alternative constraint, as defined in <i>Multiple Alternative Constraints</i> , containing an "&" indicates that the output operand is an early clobber operand. This type operand is an output operand that may be modified before the asm statement finishes using all of the input operands. The compiler will not place this operand in a register that may be used as an input operand or part of any memory address.
%	Ignored.
#	Characters following a "#" up to the first comma (if present) are to be ignored in the constraint.
*	The character that follows the "*" is to be ignored in the constraint.

The "=" and "+" modifiers apply to the operand, regardless of the number of alternatives in the constraint string. For example, the "+" in the output operand of *example13()* appears once and applies to both alternatives in the constraint string. The "&", "#", and "*" modifiers apply only to the alternative in which they appear.

Normally, the compiler assumes that input operands are used before assigning results to the output operands. This assumption lets the compiler reuse registers as needed inside the asm statement. However, if the asm statement does not follow this convention, the compiler may indiscriminately clobber a result register with an input operand. To prevent this behavior, apply the early clobber "&" modifier. An example follows:

```
void example15()
{
    int w=1;
    int z;
    asm( "movl $1, %0\n"
        "addl %2, %0\n"
```

```
"movl %2, %1"
: "=a" (w), "=r" (z) : "r" (w) );
}
```

The previous code example presents an interesting ambiguity because "w" appears both as an output and as an input operand. So, the value of "z" can be either 1 or 2, depending on whether the compiler uses the same register for operand 0 and operand 2. The use of constraint "r" for operand 2 allows the compiler to pick any general purpose register, so it may (or may not) pick register "a" for operand 2. This ambiguity can be eliminated by changing the constraint for operand 2 from "r" to "a" so the value of "z" will be 2, or by adding an early clobber "&" modifier so that "z" will be 1. The following example shows the same function with an early clobber "&" modifier:

```
void example16()
{
    int w=1;
    int z;
    asm( "movl $1, %0\n"
        "addl %2, %0\n"
        "movl %2, %1"
        : "&a" (w), "=r" (z) : "r" (w) );
}
```

Adding the early clobber "&" forces the compiler not to use the "a" register for anything other than operand 0. Operand 2 will therefore get its own register with its own copy of "w". The result for "z" in *example16()* is 1.

14.3. Operand Aliases

Extended asm specifies operands in assembly strings with a percent '%' followed by the operand number. For example, "%0" references operand 0 or the output item "&a" (w) in function *example16()* in the previous example. Extended asm also supports operand aliasing, which allows use of a symbolic name instead of a number for specifying operands, as illustrated in this example:

```
void example17()
{
    int w=1, z=0;
    asm( "movl $1, %[output1]\n"
        "addl %[input], %[output1]\n"
        "movl %[input], %[output2]"
        : [output1] "&a" (w), [output2] "=r"
        (z)
        : [input] "r" (w));
}
```

In *example18()*, "%0" and "%[output1]" both represent the output operand.

14.4. Assembly String Modifiers

Special character sequences in the assembly string affect the way the assembly is generated by the compiler. For example, the "%" is an escape sequence for specifying an operand, "%%" produces a percent for hard coded registers, and "\n" specifies a new line. [Table 32](#) summarizes these modifiers, known as *Assembly String Modifiers*.

Table 32 Assembly String Modifier Characters

Modifier	Description
\	Same as \ in printf format strings.
%%	Adds a '%' in the assembly string.
%*	Adds a '*' in the assembly string.
%A	Adds a '*' in front of an operand in the assembly string. (For example, %A0 adds a '*' in front of operand 0 in the assembly output.)
%B	Produces the byte op code suffix for this operand. (For example, %b0 produces 'b' on x86-64.)
%L	Produces the word op code suffix for this operand. (For example, %L0 produces 'l' on x86-64.)
%P	If producing Position Independent Code (PIC), the compiler adds the PIC suffix for this operand. (For example, %P0 produces @PLT on x86-64.)
%Q	Produces a quad word op code suffix for this operand if it is supported by the target. Otherwise, it produces a word op code suffix. (For example, %Q0 produces 'q' on x86-64.)
%S	Produces 's' suffix for this operand. (For example, %S0 produces 's' on x86-64.)
%T	Produces 't' suffix for this operand. (For example, %T0 produces 't' on x86-64.)
%W	Produces the half word op code suffix for this operand. (For example, %W0 produces 'w' on x86-64.)
%a	Adds open and close parentheses () around the operand.
%b	Produces the byte register name for an operand. (For example, if operand 0 is in register 'a', then %b0 will produce '%al'.)
%c	Cuts the '\$' character from an immediate operand.
%k	Produces the word register name for an operand. (For example, if operand 0 is in register 'a', then %k0 will produce '%eax'.)
%q	Produces the quad word register name for an operand if the target supports quad word. Otherwise, it produces a word register name. (For example, if operand 0 is in register 'a', then %q0 produces %rax on x86-64.)
%w	Produces the half word register name for an operand. (For example, if operand 0 is in register 'a', then %w0 will produce '%ax'.)
%z	Produces an op code suffix based on the size of an operand. (For example, 'b' for byte, 'w' for half word, 'l' for word, and 'q' for quad word.)
%+ %C %D %F %O %X %f %h %l %n %s %y are not supported.	

These modifiers begin with either a backslash "\ " or a percent "%".

The modifiers that begin with a backslash "\ " (e.g., "\n") have the same effect as they do in a printf format string. The modifiers that are preceded with a "%" are used to modify a particular operand.

These modifiers begin with either a backslash "\ " or a percent "%". For example, "%b0" means, "produce the byte or 8 bit version of operand 0". If operand 0 is a register, it will produce a byte register such as %al, %bl, %cl, and so on.

14.5. Extended Asm Macros

As with traditional inline assembly, described in [Inline Assembly](#), extended asm can be used in a macro. For example, you can use the following macro to access the runtime stack pointer.

```
#define GET_SP(x) \
asm("mov %%sp, %0": "=m" (##x) :: "%sp" );
void example20()
{
    void * stack_pointer;
    GET_SP(stack_pointer);
}
```

The GET_SP macro assigns the value of the stack pointer to whatever is inserted in its argument (for example, stack_pointer). Another "C" extension known as *statement expressions* is used to write the GET_SP macro another way:

```
#define GET_SP2 ({ \
void *my_stack_ptr; \
asm("mov %%sp, %0": "=m" (my_stack_ptr) :: "%sp" ); \
my_stack_ptr; \
})
void example21()
{
    void * stack_pointer = GET_SP2;
}
```

The statement expression allows a body of code to evaluate to a single value. This value is specified as the last instruction in the statement expression. In this case, the value is the result of the asm statement, my_stack_ptr. By writing an asm macro with a statement expression, the asm result may be assigned directly to another variable (for example, void * stack_pointer = GET_SP2) or included in a larger expression, such as: void * stack_pointer = GET_SP2 - sizeof(long).

Which style of macro to use depends on the application. If the asm statement needs to be a part of an expression, then a macro with a statement expression is a good approach. Otherwise, a traditional macro, like GET_SP(x), will probably suffice.

14.6. Intrinsics

Inline intrinsic functions map to actual x86-64 machine instructions. Intrinsics are inserted inline to avoid the overhead of a function call. The compiler has special knowledge of intrinsics, so with use of intrinsics, better code may be generated as compared to extended inline assembly code.

The NVIDIA HPC Compilers intrinsics library implements MMX, SSE, SS2, SSE3, SSSE3, SSE4a, ABM, and AVX instructions. The intrinsic functions are available to C and C++ programs. Unlike most functions which are in libraries, intrinsics are implemented internally by the compiler. A program can call the intrinsic functions from C/C++ source code after including the corresponding header file.

The intrinsics are divided into header files as follows:

Table 33 Intrinsic Header File Organization

Instructions	Header File		Instructions	Header File
ABM	intrin.h		SSE2	emmintrin.h
AVX	immintrin.h		SSE3	pmmmintrin.h
MMX	mmintrin.h		SSSE3	tmmintrin.h
SSE	xmmmintrin.h		SSE4a	ammintrin.h

The following is a simple example program that calls XMM intrinsics.

```
#include <xmmmintrin.h>
int main(){
    __m128 __A, __B, result;
    __A = _mm_set_ps(23.3, 43.7, 234.234, 98.746);
    __B = _mm_set_ps(15.4, 34.3, 4.1, 8.6);
    result = _mm_add_ps(__A, __B);
    return 0;
}
```

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, CUDA, CUDA-X, GPUDirect, HPC SDK, NGC, NVIDIA Volta, NVIDIA DGX, NVIDIA Nsight, NVLink, NVSwitch, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2013–2020 NVIDIA Corporation. All rights reserved.