



NVIDIA HPC Compilers User's Guide

Release 26.11

NVIDIA Corporation

Nov 17, 2026

Contents

1 Overview **5**

2 Creating an Example **7**

3 Invoking the Command-level NVIDIA HPC Compilers **9**
3.1 Command-line Syntax . . . 9
3.2 Command-line Options . . . 10

4 Filename Conventions **11**
4.1 Input Files . . . 11
4.2 Output Files . . . 13

5 Fortran, C++ and C Data Types **15**

6 Platform-specific considerations **17**
6.1 Using the NVIDIA HPC Compilers on Linux . . . 17

7 Site-Specific Customization of the Compilers **19**
7.1 Use siterc Files . . . 19
7.2 Using User rc Files . . . 19

8 Common Development Tasks **21**

9 Use Command-line Options **23**
9.1 Command-line Option Overview . . . 23
9.1.1 Command-line Options Syntax . . . 23
9.1.2 Command-line Suboptions . . . 24
9.1.3 Command-line Conflicting Options . . . 24
9.2 Help with Command-line Options . . . 24
9.3 Getting Started with Performance . . . 25
9.3.1 Using -fast . . . 25
9.3.2 Other Performance-Related Options . . . 26
9.4 Frequently-used Options . . . 27
9.5 Floating-point Subnormal . . . 29

10 Multicore CPU Optimization **31**
10.1 Overview of Optimization . . . 31
10.1.1 Local Optimization . . . 32
10.1.2 Global Optimization . . . 32
10.1.3 Loop Optimization: Unrolling, Vectorization and Parallelization . . . 32
10.1.4 Function Inlining . . . 32
10.2 Getting Started with Optimization . . . 33
10.2.1 -help . . . 34
10.2.2 -Minfo . . . 34

10.2.3 -Mneginfo . . . 34
10.2.4 -dryrun . . . 35
10.2.5 -v . . . 35
10.3 Local and Global Optimization . . . 35
10.3.1 -Msafeptr . . . 35
10.3.2 -O . . . 35
10.4 Loop Unrolling using -Munroll . . . 37
10.5 Vectorization using -Mvect . . . 38
10.5.1 Vectorization Sub-options . . . 39
10.5.2 Vectorization Example Using SIMD Instructions . . . 41

11 Using Function Inlining **45**
11.1 Automatic function inlining in C++ and C . . . 45
11.2 Invoking Procedure Inlining . . . 46
11.3 Using an Inline Library . . . 47
11.4 Creating an Inline Library . . . 47
11.4.1 Working with Inline Libraries . . . 48
11.4.2 Dependencies . . . 48
11.4.3 Updating Inline Libraries – Makefiles . . . 48
11.5 Error Detection during Inlining . . . 49
11.6 Examples . . . 49
11.7 Restrictions on Inlining . . . 50

12 Using GPUs **51**
12.1 Overview . . . 51
12.2 Terminology . . . 52
12.3 Execution Model . . . 53
12.3.1 Host Functions . . . 54
12.4 Memory Model . . . 54
12.4.1 Separate Host and Accelerator Memory Considerations . . . 56
12.4.1.1 Accelerator Memory . . . 56
12.4.1.2 Staging Memory Buffer . . . 56
12.4.1.3 Cache Management . . . 57
12.4.1.4 Environment Variables Controlling Device Memory Management . . . 57
12.4.1.5 NUMA Balancing . . . 59
12.4.2 Managed and Unified Memory Modes . . . 59
12.4.2.1 Managed Memory Mode . . . 59
12.4.2.2 Unified Memory Mode . . . 61
12.4.2.3 Memory Hints and Prefetching . . . 62
12.4.3 Memory Pool Allocator . . . 64
12.4.4 Interception of Deallocations . . . 65
12.4.5 Command-line Options Selecting Compiler Memory Modes . . . 66
12.4.6 Coherent Driver-based Memory Management (CDMM) . . . 66
12.4.7 Memory Bandwidth Tuning . . . 68
12.5 Fortran pointers in device code . . . 70
12.6 Calling routines in a compute kernel . . . 70
12.7 Supported Processors and GPUs . . . 71
12.8 CUDA Versions . . . 71
12.9 Compute Capability . . . 72
12.10 PTX JIT Compilation . . . 72
12.11 Supported Intrinsics . . . 76
12.11.1 Supported Fortran Intrinsics Summary Table . . . 76
12.11.2 Supported C Intrinsics Summary Table . . . 77

13 Using OpenACC **81**
13.1 OpenACC Programming Model . . . 81
13.1.1 Levels of Parallelism . . . 81
13.1.2 Enable OpenACC Directives . . . 82
13.1.3 OpenACC Support . . . 82
13.1.4 OpenACC Extensions . . . 82
13.1.5 C++ Virtual Functions . . . 83
13.2 Compiling an OpenACC Program . . . 83
13.2.1 -[no]acc . . . 83
13.2.2 -gpu . . . 84
13.3 OpenACC for Multicore CPUs . . . 87
13.4 OpenACC with CUDA Unified Memory . . . 88
13.5 OpenACC Memory Bandwidth Tuning . . . 91
13.5.1 Loop Unrolling . . . 92
13.5.2 The cache Directive . . . 93
13.5.3 The prefetch Directive . . . 94
13.5.4 Loop Tiling with Preload . . . 94
13.6 OpenACC Error Handling . . . 95
13.7 OpenACC and CUDA Graphs . . . 99
13.8 Host and Device Trip Count Options . . . 106
13.8.1 When to Use `-gpu=tripcount:device` or `-gpu=tripcount:host` . . . 106
13.9 Environment Variables . . . 107
13.10 Profiling Accelerator Kernels . . . 109
13.11 OpenACC Runtime Libraries . . . 109
13.11.1 Runtime Library Definitions . . . 110
13.11.2 Runtime Library Routines . . . 110

14 Using OpenMP **113**
14.1 Environment Variables . . . 114
14.2 Fallback Mode . . . 118
14.3 Loop . . . 118
14.4 OpenMP Subset . . . 121
14.5 Using metadirective . . . 129
14.6 Mapping target constructs to CUDA streams . . . 131
14.7 Noncontiguous Array Sections . . . 133
14.8 OpenMP with CUDA Unified Memory . . . 133
14.9 Multiple Device Support . . . 136
14.10 Interoperability with CUDA . . . 136
14.11 Interoperability with Other OpenMP Compilers . . . 136
14.12 GNU STL . . . 137

15 Using Stdpar **139**
15.1 GPU Memory Modes . . . 139
15.2 Stdpar C++ . . . 140
15.2.1 Introduction to Stdpar C++ . . . 140
15.2.2 NVC++ Compiler Parallel Algorithms Support . . . 141
15.2.2.1 Enabling Parallel Algorithms with the -stdpar Option . . . 141
15.2.3 Stdpar C++ Simple Example . . . 142
15.2.4 OpenACC Implementation of Parallel Algorithms . . . 143
15.2.5 Coding Guidelines for GPU-accelerating Parallel Algorithms . . . 143
15.2.5.1 Parallel Algorithms and Device Function Annotations . . . 144
15.2.5.2 Data Management in Parallel Algorithms . . . 144
15.2.5.3 Parallel Algorithms and Function Pointers . . . 148
15.2.5.4 Random Access Iterators . . . 149

15.2.5.5 Interoperability with the C++ Standard Library . . . 149
15.2.5.6 No Exceptions in GPU Code . . . 149
15.2.6 Stdpar C++ Standard Library Extensions . . . 150
15.2.6.1 Multi-dimensional Spans . . . 150
15.2.6.2 Linear Algebra . . . 151
15.2.7 NVC++ Experimental Features . . . 153
15.2.7.1 Senders and Receivers . . . 153
15.2.8 Stdpar C++ Larger Example: LULESH . . . 155
15.2.9 Interoperability with OpenACC . . . 156
15.2.9.1 Data Management Directives . . . 156
15.2.9.2 External Device Function Annotations . . . 161
15.2.10 Getting Started with Parallel Algorithms for GPUs . . . 161
15.2.10.1 Supported NVIDIA GPUs . . . 161
15.2.10.2 Supported CUDA Versions . . . 162
15.3 Stdpar Fortran . . . 162
15.3.1 Calling Routines in DO CONCURRENT on the GPU . . . 162
15.3.2 GPU Data Management . . . 163
15.3.3 Interoperability with OpenACC . . . 163
15.3.4 Interoperability with CUDA Fortran . . . 165

16 PCAST **167**
16.1 Overview . . . 167
16.2 PCAST with a “Golden” File . . . 168
16.3 PCAST with OpenACC . . . 171
16.4 Limitations . . . 175
16.5 Environment Variables . . . 176

17 Using MPI **179**
17.1 Using Open MPI on Linux . . . 179
17.2 Using MPI Compiler Wrappers . . . 180
17.3 Testing and Benchmarking . . . 180

18 Creating and Using Libraries **181**
18.1 Using builtin Math Functions in C++ and C . . . 181
18.2 Using System Library Routines . . . 182
18.3 Creating and Using Shared Object Files on Linux . . . 182
18.3.1 Procedure to create a use a shared object file . . . 182
18.3.2 ldd Command . . . 184
18.4 Using LIB3F . . . 184
18.5 LAPACK, BLAS and FFTs . . . 184
18.6 Linking with ScaLAPACK . . . 184
18.7 The C++ Standard Template Library . . . 185
18.8 NVIDIA Performance Libraries (NVPL) . . . 185
18.9 Linking with the nvmalloc Library . . . 186
18.9.1 Using nvmalloc with Shared Libraries . . . 186
18.9.2 Performance Considerations . . . 186

19 Environment Variables **187**
19.1 Setting Environment Variables . . . 187
19.1.1 Setting Environment Variables on Linux . . . 187
19.2 HPC Compiler Related Environment Variables - General . . . 188
19.2.1 HPC Compilers Environment Variables - General . . . 189
19.2.1.1 FORTRANOPT . . . 189
19.2.1.2 FORT_FMT_RECL . . . 189
19.2.1.3 FORT_CONVERT . . . 190

19.2.1.4 FORT_CONVERT_UNIT . . . 190
19.2.1.5 FORT_NO_EMPTY_ALLOCATION . . . 192
19.2.1.6 GMON_OUT_PREFIX . . . 193
19.2.1.7 LD_LIBRARY_PATH . . . 193
19.2.1.8 MANPATH . . . 193
19.2.1.9 NO_STOP_MESSAGE . . . 194
19.2.1.10 PATH . . . 194
19.2.1.11 NVCOMPILER_FPU_STATE . . . 194
19.2.1.12 NVCOMPILER_TERM . . . 195
19.2.1.13 NVCOMPILER_TERM_DEBUG . . . 197
19.2.1.14 PWD . . . 197
19.2.1.15 STATIC_RANDOM_SEED . . . 197
19.2.1.16 TMP . . . 197
19.2.1.17 TMPDIR . . . 198
19.3 HPC Compiler Related Environment Variables - CUDA Selection . . . 198
19.3.1 HPC Compilers Environment Variables - CUDA Selection . . . 198
19.3.1.1 NVHPC_CUDA_HOME or NVCOMPILER_CUDA_HOME . . . 199
19.3.1.2 NVCOMPILER_MATH_LIBS_HOME . . . 199
19.3.1.3 NVCOMPILER_COMM_LIBS_HOME . . . 199
19.3.1.4 NVCOMPILER_NCCL_HOME . . . 200
19.3.1.5 NVCOMPILER_SHMEM_HOME . . . 200
19.3.1.6 NVCOMPILER_CUPTI_LIBS_HOME . . . 200
19.3.1.7 NVCOMPILER_NSIGHT_COMPUTE_HOME . . . 201
19.3.1.8 NVCOMPILER_NSIGHT_SYSTEMS_HOME . . . 201
19.3.1.9 NVCOMPILER_COMPUTE_SANITIZER_HOME . . . 201
19.4 Using Environment Modules on Linux . . . 202
19.5 Stack Traceback and JIT Debugging . . . 203

20 Distributing Files - Deployment **205**
20.1 Deploying Applications on Linux . . . 205
20.1.1 Runtime Library Considerations . . . 205
20.1.2 64-bit Linux Considerations . . . 206
20.1.3 Linux Redistributable Files . . . 206
20.1.4 Restrictions on Linux Portability . . . 206
20.1.5 Licensing for Redistributable (REDIST) Files . . . 207

21 Inter-language Calling **209**
21.1 Overview of Calling Conventions . . . 209
21.2 Inter-language Calling Considerations . . . 210
21.3 Functions and Subroutines . . . 210
21.4 Upper and Lower Case Conventions, Underscores . . . 211
21.5 Compatible Data Types . . . 211
21.5.1 Fortran Named Common Blocks . . . 213
21.6 Argument Passing and Return Values . . . 213
21.6.1 Passing by Value (%VAL) . . . 214
21.6.2 Character Return Values . . . 214
21.6.3 Complex Return Values . . . 215
21.7 Array Indices . . . 215
21.8 Examples . . . 215
21.8.1 Example – Fortran Calling C . . . 216
21.8.2 Example - C Calling Fortran . . . 217
21.8.3 Example – C++ Calling C . . . 218
21.8.4 Example – C Calling C ++ . . . 219
21.8.5 Example – Fortran Calling C++ . . . 220

21.8.6 Example – C++ Calling Fortran . . . 221

22 Programming Considerations for 64-Bit Environments 223
22.1 Data Types in the 64-Bit Environment . . . 224
22.1.1 C++ and C Data Types . . . 224
22.1.2 Fortran Data Types . . . 224
22.2 Large Static Data in Linux . . . 224
22.3 Large Dynamically Allocated Data . . . 225
22.4 64-Bit Array Indexing . . . 225
22.5 Compiler Options for 64-bit Programming . . . 225
22.6 Practical Limitations of Large Array Programming . . . 226
22.7 Medium Memory Model and Large Array in C . . . 228
22.8 Medium Memory Model and Large Array in Fortran . . . 229
22.9 Large Array and Small Memory Model in Fortran . . . 230

23 C++ and C Inline Assembly and Intrinsics 231
23.1 Inline Assembly . . . 231
23.2 Extended Inline Assembly . . . 232
23.2.1 Output Operands . . . 233
23.2.2 Input Operands . . . 235
23.2.3 Clobber List . . . 236
23.2.4 Additional Constraints . . . 237
23.2.5 Simple Constraints . . . 238
23.2.6 Machine Constraints . . . 239
23.2.7 Multiple Alternative Constraints . . . 241
23.2.8 Constraint Modifiers . . . 242
23.3 Operand Aliases . . . 243
23.4 Assembly String Modifiers . . . 244
23.5 Extended Asm Macros . . . 246
23.6 Intrinsics . . . 247

NVIDIA HPC Compilers User's Guide

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 *NVFORTRAN*, *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 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 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 5.0, November 2018, 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 describes how to use OpenMP for multicore CPU programming.

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

Using Stdpar describes how to use C++/Fortran Standard Language Parallelism for programming an NVIDIA GPU or multicore CPU.

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 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

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*.
- *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).

This section describes how to use the NVIDIA HPC compilers.

Chapter 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.

Chapter 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
   ```

Chapter 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.

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.

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.

  Note: 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.

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

- You may write linker options into a text file prefixed with the '@' symbol, e.g. `@file`, and pass that file to the compiler as an option. The contents of `@file` are passed to the linker.

  ```
  $ echo "foo.o bar.o" > ./option_file.rsp
  $ nvc++ @./option_files.rsp
  ```

  The above will pass "foo.o bar.o" to the compiler as linker arguments.

Chapter 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.

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*.

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
`-E`	preprocessing	Source files	preprocessed file to standard out
`-F`	preprocessing	Source files. This option is not valid for nvc or nvc++.	preprocessed file (`.f`)
`-P`	preprocessing	Source files. This option is not valid for nvfortran.	preprocessed file (`.i`)
`-S`	compilation	Source files or preprocessed files	assembly-language file (`.s`)
`-c`	assembly	Source files, or preprocessed files, or assembly-language files	unlinked object file (`.o` or `.obj`)
none	linking	Source files, or preprocessed files, assembly-language files, object files, or libraries	executable file (`a.out`)

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.

Note: 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.

Chapter 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.

Chapter 6. Platform-specific considerations

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

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.

Chapter 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.

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.

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 `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_64/26.11/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_64/26.11/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_64/26.11/compilers/bin/siterc
Build a Fortran executable for linux that resolves shared objects in the relative directory ./REDIST	`set RPATH=./REDIST;` to ~/.mynvfortranrc

Chapter 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 9. 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.

9.1. 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*.

9.1.1. 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.

9.1.2. 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
```

9.1.3. 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.

Note: 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
```

9.2. 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|prepro|
           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>
  ```

9.3. 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.

9.3.1. 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.

Note: 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...
`-O2`	Specifies a code optimization level of 2.
`-Munroll=c:1`	Unrolls loops, executing multiple instances of the original loop during each iteration.
`-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
```

9.3.2. 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.

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 Guide.

9.4. 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...
`-acc`	Enable parallelization using OpenACC directives. By default the compilers will parallelize and offload OpenACC regions to an NVIDIA GPU. Use `-acc=multicore` to parallelize OpenACC regions for execution on all the cores of a multicore CPU.
`-fast`	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.
`-g`	Instructs the compiler to include symbolic debugging information in the object module; sets the optimization level to zero unless a `-O` option is present on the command line. Conversely, to prevent the generation of DWARF information, use the `-Mnodwarf` option.
`-gopt`	Instructs the compiler to include symbolic debugging information in the object file, and to generate optimized code identical to that generated when `-g` is not specified.
`-gpu`	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.
`-help`	Provides information about available options.
`-mcmodel=medium`	Enables medium=model code generation for 64-bit targets, which is useful when the data space of the program exceeds 4GB.
`-mp`	Enable parallelization using OpenMP directives. By default the compilers will parallelize OpenMP regions for execution on all the cores of a multicore CPU. Use `-mp=gpu` to parallelize OpenMP regions for offload to an NVIDIA GPU.
`-Mconcur`	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 multi-threaded execution context.
`-Minfo`	Instructs the compiler to produce information on standard error.
`-Minline`	Enables function inlining.
`-Mkeepasm`	Keeps the generated assembly files.
`-Munroll`	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 -O or -g options are supplied.
`-M[no]vect`	Enables [Disables] the code vectorizer.
`--[no_]exceptions`	Removes exception handling from user code. For C++, declares that the functions in this file generate no C++ exceptions, allowing more optimal code generation.
`-o`	Names the output file.
`-O <level>`	Specifies code optimization level where <level> is 0, 1, 2, 3, or 4.
`-stdpar`	Enable parallelization and offloading of Standard C++ and Fortran parallel constructs to NVIDIA GPUs; default is -stdpar=gpu.
`-tp <target>`	Specify a CPU target other than the compilation host CPU.
`-Wl, <option>`	Compiler driver passes the specified options to the linker.

9.5. Floating-point Subnormal

Starting with the 22.7 release of the NVIDIA HPC SDK the default setting of how floating-point denormal (IEEE 754 terminology “subnormal”) values are processed at runtime across both x86_64 and aarch64 processors has been changed to be more consistent.

Denormal values can be both operands to, and results of, floating-point operations. The x86_64 ISA differentiate between the two categories, operands and results, and use the terminology “daz” denormals are zeros for operands, and “flushz” flush to zero for results. The Arm V8 ISA as defined can differentiate between the two categories, but currently the processors that NVIDIA HPC SDK support only have a single setting for both operands and results and is defined as “fz” in the floating-point status and control register.

The NVIDIA HPC SDK C, C++, and Fortran compilers have command line switches `-M[no]daz` and `-M[no]flushz`, which when specified for the C/C++ main function or the Fortran main program affect how denormals are handled by the processor at runtime. The values of these two command line switches are passed to the runtime library to configure the floating-point status and control register at program startup.

NVIDIA HPC SDK supports x86_64 processors from both Intel and AMD, and ArmV8.1 and later processors. The following table summarizes the default settings of the `-Mdaz` and `-Mflushz` command line switches pre and post the 22.7 release.

Table 7: Default settings of `-Mdaz` and `-Mflushz`

	Pre 22.7 defaults	**22.7 defaults**
Intel	`-Mdaz` `-Mnoflushz`	`-Mdaz` `-Mflushz`
AMD	`-Mnodaz` `-Mnoflushz`	`-Mdaz` `-Mflushz`
Arm processors	`-Mnodaz`	`-Mdaz`

With the NVIDIA HPC SDK 22.7 release, the default handling of denormals operands and results is to treat them as zero, as if the main function/program were compiled with `-Mdaz-Mflushz`. Consequently, these changes can potentially affect applications that are dependent on subnormal values being non-zero.

Along with the change to the default treatment of denormal values, users now have the ability to configure the floating-point status and control register through the `NVCOMPILER_FPU_STATE` environment variable — effectively overriding how the program was originally compiled. For further information, see the description of the *NVCOMPILER_FPU_STATE* environment variable.

Chapter 10. 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`, and `-Mvect`. 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.

10.1. 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 architecture, instruction set and registers.

For discussion purposes, we categorize optimization:

- *Local Optimization*
- *Global Optimization*
- *Loop Optimization*
- *Optimization Through Function Inlining*

10.1.1. 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.

10.1.2. 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.

10.1.3. 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.

10.1.4. 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.

10.2. 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 prog.f
```

For all of the NVIDIA HPC Fortran, C++ and C compilers, the `-fast` option generally produces 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.

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 8: Typical `-fast` Options

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

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

Table 9: 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*.

10.2.1. -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
```

10.2.2. -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.

10.2.3. -Mneginfo

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

10.2.4. -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.

10.2.5. -v

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

10.3. Local and Global Optimization

This section describes local and global optimization.

10.3.1. -Msafeptr

The `-Msafeptr` 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 `-Msafeptr`:

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

10.3.2. -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 -0. 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 (-02 or -0) 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 -0 or -02 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
```

10.4. 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.

Note: 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 10: Example of Effect of Code Unrolling

Dot Product Code	Unrolled Dot Product Code

`REAL*4 A(100), B(100), Z`
` INTEGER I`
` DO I=1, 100`
`  Z = Z + A(i) * B(i)`
` END DO`
`END`
	`REAL*4 A(100), B(100), Z`
	` INTEGER I`
	` DO I=1, 100, 2`
	`  Z = Z + A(i) * B(i)`
	`  Z = Z + A(i+1) * B(i+1)`
	` END DO`
	`END`

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*.

10.5. 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.

10.5.1. 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 11: -Mvect Suboptions

Use this option ...	To instruct the vectorizer to do this ...
`-Mvect=altcode`	Generate appropriate code for vectorized loops.
`-Mvect=[no]assoc`	**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.
`-Mvect=fuse`	Enable loop fusion.
`-Mvect=gather`	Enable vectorization of indirect array references.
`-Mvect=idiom`	Enable idiom recognition.
`-Mvect=levels:<n>`	Set the maximum next level of loops to optimize.
`-Mvect=nocond`	Disable vectorization of loops with conditions.
`-Mvect=partial`	Enable partial loop vectorization via inner loop distribution.
`-Mvect=prefetch`	Automatically generate prefetch instructions when vectorizable loops are encountered, even in cases where SSESIMD instructions are not generated.
`-Mvect=short`	Enable short vector operations.
`-Mvect=simd`	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.
`-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.

Note: Inserting no in front of an option disables the option. For example, to disable the generation of

SIMD instructions, compile with -Mvect=nosimd.

10.5.2. 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.

Note: 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
```

(continues on next page)

(continued from previous page)

```
 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.

Note: 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).

Chapter 11. 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
```

11.1. 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

11.2. 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.

[pragma]
: Fortran Only: The pragma option is similar to the name option above except it will inline procedures marked with the `!NVF$ INLINE` pragma. To use this option, add `!NVF$ INLINE` on a separate source line immediately before the procedure's SUBROUTINE or FUNCTION statement.

[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
```

11.3. 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 tries to inline 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
```

11.4. 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.

[pragma]
Fortran Only: The pragma option is similar to the name option above except it will inline procedures marked with the `!NVF$ INLINE` pragma. To use this option, add `!NVF$ INLINE` on a separate source line immediately before the procedure's SUBROUTINE or FUNCTION statement.

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

Note: 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.

11.4.1. 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.

11.4.2. 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.

11.4.3. 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
```

11.5. 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
```

11.6. 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).

Note: 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
```

11.7. 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 12. Using GPUs

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 with NVIDIA's HPC Compilers and covers topics that affect GPU programming when using one or more of the GPU programming models.

12.1. Overview

With the NVIDIA HPC Compilers you can program NVIDIA GPUs using certain standard language constructs, OpenACC directives, OpenMP 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 NVC++ 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, NVC++ and NVC 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.

The NVFORTRAN, NVC++, and NVC compilers support a subset of the OpenMP Application Program Interface for CPUs and GPUs. OpenMP applications properly structured for GPUs, meaning they expose massive parallelism and have relatively little or no synchronization in GPU-side code segments,

should compile and execute with performance on par with or close to equivalent OpenACC. Codes that are not well-structured for GPUs may perform poorly but should execute correctly.

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.

12.2. 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 a 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 a data construct, or an implicit data region for a function or subroutine containing 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.

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.

12.3. 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.

12.3.1. 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

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

12.4. 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 many GPUs. For example:

- The host cannot read or write accelerator memory directly 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 for the compiler to assume the accelerator can read or write host memory directly. This is well-defined starting with the OpenACC 2.7 and OpenMP 5.0 specifications.

The systems with the latest GPUs provide a unified single address space between CPU and GPU for some or all memory regions, as detailed in the *Managed and Unified Memory Modes* subsection below. In these systems data can be accessed from host and accelerator subprograms without the need for explicit data movement.

The NVIDIA HPC Compilers support the following system memory modes:

Table 12: GPU Memory Modes

Memory Mode	Description	Compiler flags
Separate	All data accessed in host and accelerator programs are in separate (CPU and GPU) memories. Data in the application need to be physically moved between CPU and GPU memory either by adding explicit annotations or by relying on a compiler to detect and migrate the data.	`-gpu=mem:separate`
Managed	Dynamically allocated host data are placed in CUDA Managed Memory which is a unified single address space between host and accelerator programs and can therefore be accessed on device without explicit data movement. All other data (host, stack, or global data) remain in separate memory.	`-gpu=mem:managed`
Unified	All host data are placed in a unified single address space between the host and accelerator subprograms; no explicit data movements are required. This mode is intended for targets with full CUDA Unified Memory capability and it may utilize CUDA Managed Memory for dynamic allocations.	`-gpu=mem:unified`

If the memory mode is not selected explicitly by passing one of the above `-gpu=mem:*` options, the compiler selects a default memory mode. The default memory mode for Stdpar is explained in *Using Stdpar*. When Stdpar is not enabled, the default memory mode is Separate Memory. Memory modes may have specific semantics in each programming language and the compilers can sometimes implicitly determine the data movement that's required. More details can be found in the subsections of each programming model.

The following options `-gpu=[no]managed`, `-gpu=[no]unified` and `-gpu=pinned` are deprecated but still accepted. Refer to *Command-line Options Selecting Compiler Memory Modes* for compatibility between the current and deprecated memory specific flags.

The compiler implicitly defines the following macros corresponding to the memory mode it compiles for:

- When the code is compiled for Separate Memory Mode, the compiler defines `__NVCOMPILER_GPU_SEPARATE_MEM` macro.
- When the code is compiled for Managed Memory Mode, the compiler defines `__NVCOMPILER_GPU_MANAGED_MEM` macro.
- When the code is compiled for Unified Memory Mode, the compiler defines `__NVCOMPILER_GPU_UNIFIED_MEM` macro. If CUDA Managed Memory is utilised, the compiler defines additionally `__NVCOMPILER_GPU_MANAGED_MEM`.

When a binary is compiled for one memory mode it may need to be run on a system with specific memory capabilities as follows:

- Applications compiled for Separate Memory Mode can run on any CUDA platforms.
- Applications compiled for Managed Memory Mode must be run on platforms with CUDA Managed Memory or full CUDA Unified Memory capabilities.
- Applications compiled for Unified Memory Mode must be run on platforms with full CUDA Unified Memory.

Note: Memory allocated in the accelerator subprogram can't be accessed or deallocated from the host.

12.4.1. 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.

12.4.1.1 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.

Stack data in accelerator subprograms are allocated per thread. Stack data from one thread are not accessible by the other threads.

12.4.1.2 Staging Memory Buffer

Memory transfers between the accelerator and host may not always be asynchronous with respect to the host, even if the chosen programming model (for instance, OpenACC) declares that. This limitation may be due to the specific GPU and host memory architectures.

In order to help the host program proceed while a memory transfer to or from the accelerator is underway, the NVIDIA HPC Compilers Runtime maintains a designated staging memory area, also known as a pinned buffer. This memory area is registered with the CUDA API, which makes it suitable for asynchronous memory transfers between the GPU and the host. When an asynchronous memory transfer is started, the data being transferred is staged through the pinned buffer. Multiple asynchronous operations on the same data can be issued - in that case, the runtime system will operate on the data staged in the pinned buffer, not on the original host memory. When the host program issues an explicit or implicit synchronization request, the data is moved from the pinned buffer to its destination transparently to the application.

The runtime has the discretion to enable or disable the pinned buffer depending on the host and GPU memory architecture. Also, the size of the pinned buffer is determined by the runtime system as appropriate. The user can control some of these decisions using environment variables at the start of the application. Please refer to *Environment Variables Controlling Device Memory Management* to learn more.

12.4.1.3 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. The OpenACC programming model provides directives the programmer can use for cache management.

12.4.1.4 Environment Variables Controlling Device Memory Management

This section summarizes the environment variables that NVIDIA HPC Compilers use to control device memory management.

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

Table 13: Memory Management Environment Variables

Environment Variable	Use
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_CHECK_UNIFIED	For NVIDIA CUDA devices, in the Unified Memory mode, when set to a non-zero integer value, this environment variable enables the run-time data validation checks for data shared between host and device. For more information, please refer to *OpenACC with CUDA Unified Memory*.
NVCOMPILER_ACC_CUDA_CTX_SCHED	For NVIDIA CUDA devices, sets flags to be used when creating a new CUDA context. By default, the `CU_CTX_SCHED_YIELD` flag is used. Please refer to the CUDA Toolkit Documentation for the detailed description of the `cuCtxCreate` function and the possible flag values.
NVCOMPILER_ACC_CUDA_HEAPSIZE	For NVIDIA CUDA devices, sets the heap size limit for `malloc()` when called on device.
NVCOMPILER_ACC_CUDA_MAX_L2_FETCH_GRANULARITY	For NVIDIA CUDA devices, sets the maximum L2 cache fetch granularity size in bytes. A correct value is an integer between 0 and 128.
NVCOMPILER_ACC_CUDA_MEMALLOCASYNC	For NVIDIA CUDA devices, when set to a non-zero integer value, enables CUDA asynchronous memory allocations from the default CUDA memory pool as descibed in the CUDA Toolkit Documentation. By default, an internal NVIDIA HPC Runtime memory pool is used instead.
NVCOMPILER_ACC_CUDA_MEMALLOCASYNC_POOLSIZE	For NVIDIA CUDA devices, sets the size of the default CUDA memory pool for asynchronous allocations if the `NVCOMPILER_ACC_CUDA_MEMALLOCASYNC` environment variable is also set to a non-zero integer value.
NVCOMPILER_ACC_CUDA_NOCOPY	Disables the use of the pinned buffer when transferring user data between host and NVIDIA CUDA devices. When this variable is set to a non-zero integer value, user data will be transferred directly bypassing the pinned buffer. Asynchronous execution of such data transfers can be limited when this setting is in effect.
NVCOMPILER_ACC_CUDA_PIN	For NVIDIA CUDA devices, enables host memory pinning at data directives. When host memory is pinned, data transfers to and from the device can be asynchronous, which can potentially improve program performance. A non-zero integer value enables this mechanism. A value of 2 or greater additionally disallows unpinning the host data after it is pinned. A value of 3 or greater also enables pinning the whole array referenced in a data directive (provided that the size of the array is known), rather than its subarray specified in the data directive. By default, host data referenced at data directives is not pinned un-

12.4.1.5 NUMA Balancing

In system configurations where GPU memory is visible to the operating system as a NUMA node, automatic NUMA balancing (`/proc/sys/kernel/numa_balancing=1`) can degrade application performance and increase run-to-run performance variation in certain situations.

For best performance, disable automatic NUMA balancing system-wide when possible. Otherwise, use an explicit memory mapping policy, such as `numactl -m 0 ./app`, to bind CPU memory allocations to the CPU NUMA node closest to the GPU used by the application.

12.4.2. Managed and Unified Memory Modes

The NVIDIA HPC Compilers support interoperability with CUDA Unified Memory. This feature is available with the x86-64 and Arm Server compilers. Unified memory provides a single address space for CPU and GPU; data movement between CPU and GPU memories is implicitly handled by the NVIDIA CUDA 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.

12.4.2.1 Managed Memory Mode

In Managed Memory Mode, all Fortran, C++ and C explicit allocation statements (e.g. `allocate`, `new`, and `malloc`, respectively) in a program unit are replaced by equivalent CUDA managed data allocation calls that place the data in CUDA Managed Memory. The result is that OpenACC and OpenMP data clauses and directives are not needed to manage data movement. They are essentially ignored and can be omitted. For Stdpar this is the minimal required memory mode since there are no specific annotations for data used in the parallel region.

To enable Managed Memory Mode, add the option `-gpu=mem:managed` to the compiler and linker command lines.

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 deallocate operations. More details can be found in *Memory Pool Allocator*.

Managed Memory Mode has the following limitations:

- Use of managed memory applies only to dynamically-allocated data.
- Given an allocatable aggregate with a member that points to local, global, or static data, compiling with `-gpu=mem:managed` and attempting to access memory through that pointer from the compute kernel will cause a failure at runtime.
- A C++ virtual call is supported when the function written at the call has an explicit `acc routine` directive and the object is constructed in device code. Each override that can run on the device needs the same directive. A host constructor stores the host virtual-table address in the object, and copying the object to the device copies that address. A call through a function pointer or a pointer to a member is not supported. See C++ Virtual Functions in the OpenACC Getting Started Guide.

- The `-gpu=mem:managed` compiler option must be used to compile the files in which variables (accessed from GPU) are allocated, even if there is no code to accelerate on the GPU in the source file.
- When linking multiple translation units, the application must ensure that all data are deallocated using the scheme corresponding to their allocation. For example if the data are allocated in managed memory the deallocation must be performed using CUDA API calls for managed memory. More details and extra compiler support is detailed in *Interception of Deallocations*.

Managed Memory Mode 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 Compiler Runtime enforces synchronous execution of kernels when `-gpu=mem: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.

Memory Allocations/Deallocations Automatically Changed to Managed Memory

When the compiler utilizes CUDA Managed Memory capability either with `-gpu=mem:managed` or `-gpu=mem:unified`, the following explicit allocations/deallocations are automatically changed into `cudaMallocManaged`/`cudaFree`-type allocations/deallocations:

- For C++:
  - All calls to global `operator new` and `operator delete` that allocate or deallocate memory, such as:

```
operator new(std::size_t size)
operator new(std::size_t size, const std::nothrow_t &nothrow_value)
operator new(std::size_t size, std::align_val_t align)
operator new(std::size_t size, std::align_val_t align, const std::nothrow_t &
↪nothrow_value)
operator delete(void *p)
operator delete(void *p, std::size_t size)
operator delete(void *p, std::align_val_t align)
operator delete(void *p, std::size_t size, std::align_val_t align)
operator delete(void *p, const std::nothrow_t &nothrow_value)
operator delete(void *p, std::align_val_t align, const std::nothrow_t &
↪nothrow_value)
```

  - All the array forms of the above overloads.
  - All calls to `malloc`/`free` functions.
- For C: all calls to `malloc`/`free` functions.
- For Fortran:
  - All allocations of automatic arrays.
  - all `allocate`/`deallocate` statements with allocatable arrays or pointer variables.

12.4.2.2 Unified Memory Mode

In Unified Memory Mode, the requirements for the program are further relaxed compared to Managed Memory Mode. Specifically, not only is dynamically allocated system memory accessible on the GPU, but global and local memory are also accessible.

To enable this feature, add the option `-gpu=mem:unified` to the compiler and linker command lines.

Programs compiled with `-gpu=mem:unified` must be run on systems that support full CUDA Unified Memory capability. At this time, full CUDA Unified Memory is supported on NVIDIA Grace Hopper Superchip systems, NVIDIA Grace Blackwell Superchip systems, and Linux x86-64 systems running with the Heterogeneous Memory Management (HMM) feature enabled in the Linux kernel. Details about these platforms are available in the following blog posts on the NVIDIA website: Simplifying GPU Programming for HPC with NVIDIA Grace Hopper Superchip and Simplifying GPU Application Development with Heterogeneous Memory Management.

In Unified Memory Mode, the compiler assumes that any system memory is accessible on the GPU. Even so, the compiler may generate managed memory allocations for explicit data allocations when it considers them beneficial for program performance. If you would like to enforce or prohibit the use of managed memory for dynamic allocations pass `-gpu=mem:unified:[no]managedalloc` to compilation and linking.

If the program is compiled with the `-gpu=mem:unified` switch, you can also use the `NVCOMPILER_ACC_UNIFIED_ALLOC_MODE` environment variable to select the dynamic memory allocation mode at runtime. Please refer to *Environment Variables Controlling Device Memory Management* for a comprehensive list of its recognized values.

Unified Memory Mode has the following limitations:

- Unified memory support for OpenACC, OpenMP and Stdpar Fortran is not mix-and-match; all object files containing OpenACC/OpenMP directives or Fortran `DO CONCURRENT` constructs must be compiled and linked with `-gpu=mem:unified` to ensure correct execution.
- A C++ virtual call is supported when the function written at the call has an explicit `acc routine` directive and the object is constructed in device code. Each override that can run on the device needs the same directive. A host constructor stores the host virtual-table address in the object, and copying the object to the device copies that address. A call through a function pointer or a pointer to a member is not supported. See C++ Virtual Functions in the OpenACC Getting Started Guide.

Transitioning to Unified Memory Mode

Applications transitioning to architectures that support Unified Memory Mode can be recompiled with `-gpu=mem:unified` without any code modifications.

The programmer should be aware that in Unified Memory Mode, the whole program state becomes essentially shared between the CPU and the GPU. By implication, modifications to program variables made on the GPU are visible on the CPU. That is, the GPU does not operate on a copy of the data even if the program contains respective directives, but instead the GPU operates directly on the data in system memory. To understand the importance of this idea, consider the following OpenACC C program:

```
int x[N];
void foo() {
  #pragma acc enter data create(x[0:N])
  #pragma acc parallel loop
  for (int i = 0; i < N; i++) {
    x[i] = i;
```

(continues on next page)

(continued from previous page)

```
  }
}
```

When compiled in Separate Memory Mode, in the `foo()` function a copy of the array `x` is created in GPU memory and initialized as written in the `loop` construct. When `-gpu=mem:unified` is added, however, the compiler ignores the `acc enter data` construct, and the `loop` construct initializes the array `x` in system memory.

Another implication of which to be aware, *asynchronous* code execution on the GPU can introduce race conditions over access to program data. More details about code patterns to avoid when writing application sources for Unified Memory Mode can be found in the sections about specific programming models of this guide e.g. OpenACC, OpenMP, or CUDA Fortran.

12.4.2.3 Memory Hints and Prefetching

In Managed and Unified Memory Modes, applications can benefit from automatically issued memory hints and prefetching at both explicit and implicit data constructs.

Consider the following example:

```
float *a = malloc(N * sizeof(float));
for (int i = 0; i < N; i++)
   a[i] = i;
#pragma acc enter data copyin(a) async
#pragma acc parallel loop async
for (int i = 0; i < N; i++)
   a[i] = a[i] * 2.0;
#pragma acc update self(a) async
```

If the array `a` is allocated in managed or unified memory, no explicit memory copies are performed. However, at the `acc enter data` directive, it is often beneficial to automatically set the preferred memory location of the data to the device. At the `acc update` directive, the data is updated on the host, which the OpenACC runtime can translate into asynchronous memory prefetching from the device.

Similarly, copying data to the host or updating data on the device can be automatically translated into setting the preferred memory location to the host or prefetching to the device, respectively.

This behavior of the OpenACC runtime is controlled by the `NVCOMPILER_ACC_MEMHINTS` and `NVCOMPILER_ACC_MEMPREFETCH` environment variables. They similarly control the behavior of the NVIDIA OpenMP Target Offload runtime. Refer to the section *Environment Variables Controlling Device Memory Management* to learn more.

OpenACC Runtime Interface (Fortran)

```
integer(accx_mem_hints_kind), parameter :: accx_mem_hints_default = 0
integer(accx_mem_hints_kind), parameter :: accx_mem_hints_disable = 1
integer(accx_mem_hints_kind), parameter :: accx_mem_hints_enable_explicit = 2
integer(accx_mem_hints_kind), parameter :: accx_mem_hints_enable_all = 3

integer(accx_mem_prefetch_kind), parameter :: accx_mem_prefetch_default = 0
integer(accx_mem_prefetch_kind), parameter :: accx_mem_prefetch_disable = 1
integer(accx_mem_prefetch_kind), parameter :: accx_mem_prefetch_enable_update = 2
integer(accx_mem_prefetch_kind), parameter :: accx_mem_prefetch_enable_explicit = 3
integer(accx_mem_prefetch_kind), parameter :: accx_mem_prefetch_enable_all = 4
```

(continues on next page)

(continued from previous page)

```
subroutine accx_set_mem_hints( mode ) bind(c,name='accx_set_mem_hints')
subroutine accx_set_mem_prefetch( mode ) bind(c,name='accx_set_mem_prefetch')
```

OpenACC Runtime Interface (C)

```
typedef enum {
  accx_mem_hints_default,
  accx_mem_hints_disable,
  accx_mem_hints_enable_explicit
  accx_mem_hints_enable_all
} accx_mem_hints_t;

typedef enum {
  accx_mem_prefetch_default,
  accx_mem_prefetch_disable,
  accx_mem_prefetch_enable_update,
  accx_mem_prefetch_enable_explicit
  accx_mem_prefetch_enable_all
} accx_mem_prefetch_t;

void accx_set_mem_hints(accx_mem_hints_t value);
void accx_set_mem_prefetch(accx_mem_prefetch_t value);
```

The `accx_set_mem_hints` and `accx_set_mem_prefetch` APIs operate on a per-thread basis, meaning they affect only the calling host thread. The parameter values correspond directly to the values of the `NVCOMPILER_ACC_MEMHINTS` and `NVCOMPILER_ACC_MEMPREFETCH` environment variables.

Calling these APIs will similarly affect OpenMP target data constructs when the application is compiled with `-mp=gpu`.

The following APIs operate on a specific region of memory as a convenient shortcut you can use instead of relying on the CUDA Runtime API:

OpenACC Runtime Interface (Fortran)

```
integer, parameter :: accx_mem_advise_kind = 4
integer(accx_mem_advise_kind), parameter :: accx_mem_advise_set_read_mostly = 0
integer(accx_mem_advise_kind), parameter :: accx_mem_advise_unset_read_mostly = 1
integer(accx_mem_advise_kind), parameter :: accx_mem_advise_set_preferred_location = 2
integer(accx_mem_advise_kind), parameter :: accx_mem_advise_unset_preferred_location
↪= 3
integer(accx_mem_advise_kind), parameter :: accx_mem_advise_set_accessed_by = 4
integer(accx_mem_advise_kind), parameter :: accx_mem_advise_unset_accessed_by = 5

integer, parameter :: accx_mem_advise_location_kind = 4
integer(accx_mem_advise_location_kind), parameter :: accx_mem_advise_location_none = 0
integer(accx_mem_advise_location_kind), parameter :: accx_mem_advise_location_host = 1
integer(accx_mem_advise_location_kind), parameter :: accx_mem_advise_location_device
↪= 2

integer, parameter :: accx_mem_prefetch_location_kind = 4
integer(accx_mem_prefetch_location_kind), parameter :: accx_mem_prefetch_location_
↪host = 0
integer(accx_mem_prefetch_location_kind), parameter :: accx_mem_prefetch_location_
↪device = 1

subroutine accx_mem_advise (advice, location, ptr, bytes) bind(c,name='accx_mem_advise
↪')
```

(continues on next page)

(continued from previous page)

```
subroutine accx_mem_prefetch (location, ptr, bytes, async) bind(c,name='accx_mem_
↪prefetch')
```

OpenACC Runtime Interface (C)

```
typedef enum {
  accx_mem_advise_set_read_mostly,
  accx_mem_advise_unset_read_mostly,
  accx_mem_advise_set_preferred_location,
  accx_mem_advise_unset_preferred_location,
  accx_mem_advise_set_accessed_by,
  accx_mem_advise_unset_accessed_by,
} accx_mem_advise_t;

typedef enum {
  accx_mem_advise_location_none,
  accx_mem_advise_location_host,
  accx_mem_advise_location_device
} accx_mem_advise_location_t;

typedef enum {
  accx_mem_prefetch_location_host,
  accx_mem_prefetch_location_device
} accx_mem_prefetch_location_t;

void accx_mem_advise(accx_mem_advise_t advice,
                     accx_mem_advise_location_t location,
                     void *ptr, size_t size);

void accx_mem_prefetch(accx_mem_prefetch_location_t location,
                       void *ptr, size_t size,
                       int async);
```

The `accx_mem_advise` function maps directly to the CUDA Runtime function `cudaMemAdvise`.

The `accx_mem_prefetch` function maps to the CUDA Runtime function `cudaMemPrefetchAsync`. A notable difference is the `async` parameter, which is associated with an OpenACC async queue, rather than a specific CUDA stream.

For more information, refer to the Memory Management section in CUDA Toolkit Documentation.

12.4.3. Memory Pool Allocator

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

To mitigate the overhead of `cudaMallocManaged()` or other CUDA allocation API calls, there is a pool allocator enabled by default in the presence of the `-gpu=mem:managed`, `-gpu=mem:separate:pinnedalloc`, or `-gpu=mem:unified` compiler options. It can be disabled, or its behavior modified, using these environment variables:

Table 14: Pool Allocator Environment Variables

Environment Variable	Use
NVCOMPILER_ACC_POOL_ALLOC	Disable the pool allocator. The pool allocator is enabled by default; to disable it, set NVCOMPILER_ACC_POOL_ALLOC to 0.
NVCOMPILER_ACC_POOL_SIZE	Set the 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 NVCOMPILER_ACC_POOL_THRESHOLD value.
NVCOMPILER_ACC_POOL_ALLOC_MAXSIZE	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.
NVCOMPILER_ACC_POOL_ALLOC_MINSIZE	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.
NVCOMPILER_ACC_POOL_THRESHOLD	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.

Note: Note that where the size is specified if the unit suffix (B, KB, MB or GB) is ommited, the value is set by default in bytes.

12.4.4. Interception of Deallocations

While NVIDIA HPC Compilers facilitate the use of managed or pinned memory automatically, the application must ensure that memory is deallocated using the API which "matches" the API used to allocate said memory. For example, if `cudaMallocManaged` is used to allocate, then `cudaFree` must be used to deallocate; if `cudaMallocHost` is used for allocations, `cudaFreeHost` must be used for deallocations. Understanding this requirement is particularly important when third party or standard libraries are used; these libraries may have been compiled without any memory mode settings which sets up a situation where the deallocation routines in the libraries may not match the allocations made. When data is deallocated with an unmatching API call, the application may exhibit undefined behavior including crashing. To mitigate this issue, the compiler supports an interception mode in which calls to the standard deallocation function (e.g. free in C, delete in C++, or deallocate in Fortran) are inspected by the runtime and, if the memory is not detected as being system-allocated, the runtime replaces the standard deallocation function with the deallocation API corresponding to the allocation scheme in use. To activate this interception mode, use the `-gpu=interceptdeallocations` compiler flag. The interception is enabled by default for Stdpar in the presence of managed memory allocations. To deactivate the interception use the `-gpu=nointerceptdeallocations` compiler switch. This interception can incur extra runtime overhead.

12.4.5. Command-line Options Selecting Compiler Memory Modes

The following table maps the new memory model flags to their deprecated equivalents.

Table 15: Command-line Options Corresponding to Compiler Memory Modes

Current Flags	Deprecated Flags	Brief Description
`-gpu=mem:managed`	`-gpu=managed`	Managed Memory Mode
`-gpu=mem:managed -stdpar`	`-gpu=nounified -stdpar`	Managed Memory Mode
`-gpu=mem:unified`	`-gpu=unified`	Unified Memory Mode
`-gpu=mem:unified:managedalloc`	`-gpu=unified,managed`	Unified Memory Mode, all dynamically allocated data are implicitly in CUDA Managed Memory.
`-gpu=mem:unified:nomanagedalloc`	`-gpu=unified, nomanaged`	Unified Memory Mode, CUDA Managed Memory is not used implicitly.
`-gpu=mem:separate`	`-gpu=nomanaged`	Separate Memory Mode
`-gpu=mem:separate`	`-gpu=nounified`	Separate Memory Mode
`-gpu=mem:separate`	`-gpu=nomanaged, nounified`	Separate Memory Mode
`-gpu=mem:separate:pinnedalloc`	`-gpu=pinned`	Separate Memory Mode, dynamically allocated data are in CPU pinned memory implicitly.

12.4.6. Coherent Driver-based Memory Management (CDMM)

NVIDIA hardware-coherent architectures (including Grace Hopper, Grace Blackwell, and Vera Rubin) allow the NVIDIA driver to operate in either NUMA or CDMM mode. While NUMA mode exposes both CPU and GPU memory directly to the operating system, CDMM mode (available in driver version 13.0 and later) prevents GPU memory from being exposed to the operating system as a NUMA node. In this configuration, the NVIDIA driver manages GPU memory separately from CPU system memory. Detailed information is available in the blog post Understanding Memory Management on Hardware-Coherent Platforms. Starting with CUDA driver 13.4 (branch R615), CDMM mode is default.

All applications will continue to function correctly in CDMM mode. Performance will remain unchanged for OpenACC, OpenMP, and Stdpar applications compiled for Separate or Managed Memory Mode.

The behavior and performance of Unified Memory applications may be affected by CDMM mode. In NVIDIA driver versions up to and including 13.5, system-allocated memory will not migrate to GPU memory. The GPU is able to access system-allocated memory, but there will be no migration of memory pages. This may hurt application performance if the data is accessed frequently by the GPU. NVHPC-inserted memory advice and prefetch hints have no effect in CDMM mode.

We recommend that HPC systems are configured in NUMA mode rather than CDMM mode for best performance of Unified Memory applications. To configure NUMA mode, see the NVIDIA Data Center GPU Driver Release Notes: select the release notes for the driver version you are installing and follow its NUMA mode configuration instructions. To check which mode a system is in:

```
$ grep Coherent /proc/driver/nvidia/params
```

This will return CoherentGPUMemoryMode: "driver" for CDMM, or "numa" for NUMA mode. When in NUMA mode, please also check that migration of system-allocated memory is enabled:

```
$ cat /sys/module/nvidia_uvm/parameters/uvm_disable_sam_migration
```

This will return "N" if migration is enabled and "Y" if it is disabled; it should be "N". If it returns "Y", ask your system administrator to reload the `nvidia_uvm` kernel module with `uvm_disable_sam_migration=0`. If the file does not exist, the installed driver predates this parameter and no action is needed.

If CDMM mode is necessary, please see the following recommendations.

To facilitate data migration, compile applications using the `-gpu=mem:unified` flag so that dynamic memory allocations use CUDA managed memory (refer to *Command-line Options Selecting Compiler Memory Modes*). Avoid the `-gpu=mem:unified:nomanagedalloc` flag, as it will use system-allocated memory. Additionally, since static, global, and local data are not allocated dynamically, they will remain in CPU memory; transforming these into dynamic allocations can lead to better performance. Set `NVCOMPILER_ACC_MEMHINTS=DISABLE` to eliminate overhead from the memory hint API. Please note that these recommendations are subject to change as CDMM Unified Memory capabilities evolve in subsequent NVIDIA driver versions.

Table 16: CDMM vs. NUMA Comparison and NVHPC Guidance

Mode/Version	**Memory Management/OS Exposure**	**Key Behavior/Support**	**NVHPC Compiler/Runtime Suggestion**
NUMA Mode	Both CPU and GPU memory exposed directly to the OS	Preferred for optimal Unified Memory application performance	N/A (optimal configuration)
CDMM (Driver <= 13.5)	GPU memory is hidden from the OS; managed by NVIDIA driver	System-allocated memory will not migrate to GPU memory. NVHPC memory advice and prefetch hints have no effect	Compile: Use `-gpu=mem:unified`, avoid `-gpu=mem:unified:nomanagedalloc` Data: Convert static/global/local data to dynamic allocations Runtime: Set `NVCOMPILER_ACC_MEMHINTS=DISABLE`
CDMM (Driver > 13.5)	GPU memory is hidden from the OS; managed by NVIDIA driver	Capabilities evolving	Follow the guidance for Driver <= 13.5 until new recommendations are provided

12.4.7. Memory Bandwidth Tuning

For some GPU generations, memory bandwidth (BW) grows faster than the number of streaming multiprocessors (SMs): from Hopper to Blackwell, for example, memory BW more than doubles while the number of SMs grows only slightly, so the memory BW per SM roughly doubles and each SM must drive more memory traffic to saturate the memory BW. Code that is parallelized well enough to saturate the memory BW of one GPU may not saturate it on a newer GPU.

The memory BW a kernel achieves is determined by the number of *bytes in flight*: the total number of bytes that have been requested from memory at a point in time.

The bytes in flight needed per SM to saturate the memory BW increase with each GPU architecture. For an analysis of bytes in flight on recent NVIDIA GPUs and of the techniques described in this section, refer to the GTC 2025 presentation CUDA Techniques to Maximize Memory Bandwidth and Hide Latency (S72683).

Data Preload

In typical GPU code, loads from global memory are interleaved with the computation that uses them: a thread loads some data, waits for it, computes on it, and then loads the next data, so only a few loads are in flight at a time. *Data preload* is a technique that increases the bytes in flight by issuing the loads of data together, ahead of the computation that uses them, into a fast memory close to the SM. The computation then uses the data in fast memory instead of waiting for each load from global memory.

Preload applies within a single loop iteration when the loop body loads a large amount of data: the independent loads of the whole iteration are clustered and issued before the computation. When a single iteration does not load enough data, preload can be extended across iterations: the data needed by several iterations are loaded together, and the iterations then compute on the preloaded data.

Data can be preloaded into registers, CUDA shared memory, or the L1 cache:

Table 17: Data Preload Variants

Variant	Description	Considerations
Register preload	Independent loads, within a loop iteration or across several iterations, are clustered and issued together, and the data are kept in registers of each thread until they are used.	Needs no extra memory, but increases the number of registers per thread. Too many registers per thread reduce occupancy or cause register spills.
Shared memory preload	A slice of an array is copied from global memory into CUDA shared memory, where it is available to all threads of the thread block. On GPUs with compute capability 9.0 and later, the copy can use the Tensor Memory Accelerator (TMA).	Works best for contiguous slices with no scatter or gather. Uses shared memory, which can limit occupancy.
L1 preload	Prefetch instructions request data into the L1 cache without holding registers or shared memory.	Has nearly no constraints and no impact on occupancy, but data placement is less predictable: prefetched data may be evicted before they are used.

The amount of data that can be preloaded is limited by the registers and shared memory available within the occupancy limit, and by the L1 cache capacity. Preloading more data increases the bytes in flight but may reduce occupancy, so consider the trade-off between occupancy and preload size. Preload only as much data as needed to reach the maximum bytes in flight the memory subsystem can

sustain: preloading beyond that does not increase the memory BW further and only uses resources, so bound the preload size accordingly. To choose a preload strategy:

1. Assess the volume of data loaded in each program point / loop iteration. Compiler optimizations such as loop unrolling and load clustering change this volume; use `-Minfo` to see what the compiler did. If a single iteration does not load enough data to reach the memory BW target, unroll or tile the loop so that the preload covers more than one iteration.
2. Choose the resource to preload into:
   - If registers are available within the occupancy limit, register preload by the compiler, without extra directives, may be enough.
   - If registers are tight or register preload is limited, use L1 preload.
   - If shared memory is available without an impact on occupancy and the data form a contiguous slice, use shared memory preload, preferably with TMA if the contiguous block is large enough.

Automatic Preload

Starting with the 26.9 release, the NVIDIA HPC compilers automatically preload data into registers: loads from global memory are clustered and issued ahead of the computation that uses them, using the available registers. No directives or source changes are needed.

Increasing Bytes in Flight with Language Features

To further tune the memory BW by increasing the bytes in flight, you can use language features that control the preload explicitly, such as loop unrolling to preload more data into registers, and directives that preload data into shared memory or the L1 cache. These features are specific to each programming model; for OpenACC, refer to *OpenACC Memory Bandwidth Tuning*.

Measuring Memory Bandwidth

Use NVIDIA Nsight Compute to check whether a preload improves a kernel. Prefetching can increase the total data moved, because of L1 evictions, register spills, or TMA copying more than is used, so check the kernel time as well as the memory BW. The following metrics are useful:

Table 18: Nsight Compute Metrics for Memory Bandwidth Tuning

Metric or section	Use
`gpu__time_duration.avg`	Kernel time; a higher memory BW only matters if the kernel runs faster.
`dram__bytes.sum`	Total data moved to and from global memory.
`dram__bytes.sum.per_second`	Achieved memory BW; the key indicator of the effect of a preload.
`launch__registers_per_thread`	Whether the parallelism of the kernel is limited by registers.
`LaunchStats` section	Static and dynamic shared memory per block; whether the parallelism is limited by shared memory.
`Occupancy` section	Achieved occupancy, and whether it is limited by registers or shared memory (`Block Limit Registers`, `Block Limit Shared Mem`).

For example:

```
$ ncu --metrics gpu__time_duration.avg,dram__bytes.sum,dram__bytes.sum.per_second,
↪launch__registers_per_thread --section LaunchStats --section Occupancy ./a.out
```

12.5. Fortran pointers in device code

A Fortran pointer variable is implemented with a pointer and a descriptor, where the descriptor (often called a "dope vector") holds the array bounds and strides for each dimension, among other information, such as the size for each element and whether the pointer is associated. A Fortran scalar pointer has no bounds information, but does have a minimal descriptor. In Fortran, referring to the pointer variable always refers to the pointer target. There is no syntax to explicitly refer to the pointer and descriptor that implement the pointer variable.

Fortran allocatable arrays and variables are implemented much the same way as pointer arrays and variables. Much of the discussion below applies both to allocatables and pointers.

In OpenACC and OpenMP, when a pointer variable reference appears in a data clause, it's the pointer target that gets allocated or moved to device memory. The pointer and descriptor are neither allocated nor moved.

When a pointer variable is declared in a module declaration section and appears in an `!$acc declare create()` or `!$omp declare target to()` directive, then the pointer and descriptor are statically allocated in device memory. When the pointer variable appears in a data clause, the pointer target is allocated or copied to the device, and the pointer and descriptor are 'attached' to the device copy of the data. If the pointer target is already present in device memory, no new memory is allocated or copied, but the pointer and descriptor are still 'attached', making the pointer valid in device memory. An important side effect of adding `declare create` in the module declaration section is that when the program executes an 'allocate' statement for the pointer (or allocatable), memory is allocated in both CPU and device memory. This means the newly allocated data is already present in device memory. To get values from CPU to device memory or back, you'll have to use `update` directives.

When a pointer variable is used in an OpenACC or OpenMP compute construct, the compiler creates a private copy of the pointer and descriptor for each thread, unless the pointer variable was in a module as described above. The private pointer and descriptor will contain information about the device copy of the pointer target. In the compute construct, the pointer variables may be used pretty much as they can in host code outside a compute construct. However, there are some limitations. The program can do a pointer assignment to the pointer, changing the pointer, but that will only change the private pointer for that thread. The modified pointer in the compute construct will not change the corresponding pointer and descriptor in host memory.

12.6. Calling routines in a compute kernel

Using explicit interfaces is a common occurrence when writing Fortran applications. Here are some cases where doing so is required for GPU programming.

- Explicit interfaces are required when using OpenACC `routine bind` or OpenMP `declare variant`.
- Fortran `do concurrent` requires routines to be `pure` which creates the need for an explicit interface.

12.7. Supported Processors and GPUs

This NVIDIA HPC Compilers release supports x86-64 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.

To direct the compilers to generate code for NVIDIA GPUs, use the `-acc` flag to enable OpenACC directives, the `-mp=gpu` flag to enable OpenMP directives, the `-stdpar` flag for standard language parallelism, and the `-cuda` flag for CUDA Fortran. Use the `-gpu` flag to select specific options for GPU code generation. 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

12.8. 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 puts the CUDA Toolkit components into an HPC SDK installation sub-directory; the HPC SDK currently bundles two versions of recently-released Toolkits.

You can compile a program for an NVIDIA GPU on any system supported by the HPC compilers. You will be able to run that program only on a system with an NVIDIA GPU and an installed NVIDIA CUDA driver. NVIDIA HPC SDK products do not contain CUDA device drivers. You must download and install the appropriate CUDA Driver from NVIDIA.

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 26.11 includes components from the following versions of the CUDA Toolkit:

- CUDA 12.9U1
- CUDA 13.0

If you are compiling a program for GPU execution on a system *without* an installed CUDA driver, the compiler selects the version of the CUDA Toolkit to use based on the value of the `DEFCUDAVERSION` variable contained in a file called `localrc` which is created during installation of the HPC SDK.

If you are compiling a program for GPU execution on a system *with* an installed CUDA driver, the compiler detects the version of the CUDA driver and selects the appropriate CUDA Toolkit version to use from those bundled with the HPC SDK.

The compilers look for a CUDA Toolkit version in the /opt/nvidia/hpc_sdk/*target*/26.11/cuda directory that matches the version of the CUDA Driver installed on the system. If an exact match is not found, the compiler searches for the closest match. For CUDA Driver versions 12.0 through 12.9, the compiler will use the CUDA 12.9 Toolkit. For CUDA Driver versions 13.0 and later, the compiler will use the CUDA 13.0 Toolkit.

You can change the compiler's default selection of CUDA Toolkit version using 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 Toolkit version for one invocation of the compiler. For example, to compile an OpenACC C file with the CUDA 12.9 Toolkit you would use:

```
nvc -acc -gpu=cuda12.9
```

12.9. Compute Capability

The compilers can generate code for NVIDIA GPU compute capabilities 5.0 through 12.1. 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 generate code for every supported compute capability.

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=80 90;
```

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.

12.10. PTX JIT Compilation

As of HPC SDK 22.9, support for PTX JIT compilation is enabled in all compilers for relocatable device code mode. This means that applications built with `-gpu=rdc` (that is, with relocatable device code enabled, which is the default mode) are forward-compatible with newer GPUs thanks to the embedded PTX code. The embedded PTX code is dynamically compiled when the application runs on a GPU architecture newer than the architecture specified at compile time.

The support for PTX JIT compilation is enabled automatically, which means that you do not need to change the compiler invocation command lines for your existing projects.

Use scenarios

- As an example, you can compile your application targeting the Ampere GPU without having to worry about the Hopper GPU architecture. Once the application runs on a Hopper GPU, it will seamlessly use the embedded PTX code.
- In CUDA Fortran, or with the CUDA Interoperability mode enabled, you can mix in object files compiled with the CUDA NVCC compiler containing PTX code. This PTX code from NVCC will be handled by the JIT compiler alongside the PTX code contained in object files produced by the HPC SDK compilers. When using the CUDA NVCC compiler, the relocatable device code generation must be enabled explicitly using the NVCC `--relocatable-device-code` true switch, as explained in the CUDA Compiler Driver guide. More information is available in the *Interoperability with CUDA* section of this guide and in the CUDA Fortran Programming Guide.

By default, the compiler will choose the compute capability that matches the GPU on the system where the code is being compiled. For code that is going to run on the system where it is compiled, we recommend letting the compiler set the compute capability.

When the default won't work, we recommend compiling applications for a range of compute capabilities that the application is expected to run against, for example, using the `-gpu=ccall` compiler option. When running the application on a system that supports one of those compute capabilities, the CUDA driver minor version is allowed to be less than the version of the CUDA toolkit used at compile time, as covered in section *CUDA Versions*.

Performance considerations

PTX JIT compilation, when it occurs, can have a start-up overhead for the application. The JIT compiler keeps a cached copy of the produced device code, which reduces the overhead on subsequent runs. Please refer to the CUDA Programming Guide for detailed information about how the JIT compiler works.

Known limitations

In general, in order for PTX JIT compilation to work, the CUDA driver installed on the deployment system must be at least of the version that matches the CUDA toolkit used to compile the application. This requirement is stricter than those explained in section *CUDA Versions*.

For example, as explained in that section, the compilers will use the CUDA 12.9 toolkit that is shipped as part of the HPC SDK toolkit when the CUDA driver installed in the system is at least 12.0. However, while the CUDA 12.0 driver is commonly sufficient to run the application, it will not be able to compile the PTX code produced by the CUDA 12.9 toolkit. This means that any deployment system where the PTX JIT compilation is expected to be used must have at least the CUDA 12.9 driver installed. Please refer to the CUDA Compatibility guide for further information about the CUDA Driver compatibility with CUDA Toolkits.

When the application is expected to run on a newer GPU architecture than specified at compile time, we recommend having a CUDA driver installed on the deployment system matching the CUDA toolkit used to build the application. One way to achieve that is to use the `NVHPC_CUDA_HOME` environment variable at compile time to provide a specific CUDA toolkit.

Below are a few examples of how the PTX version incompatibility can be diagnosed and fixed. As a general rule, if the CUDA driver is unable to run the application due to incompatible PTX, the application will terminate with an error message indicating the cause. OpenACC and OpenMP applications will in most cases suggest compiler flags to target the current CUDA installation.

OpenACC

Consider this program that we will compile for Volta GPU and attempt to run on an Ampere GPU, on a system that has CUDA 12.1 installed:

```
#include <stdio.h>
#define N 1000
int array[N];
int main() {
#pragma acc parallel loop copy(array[0:N])
   for(int i = 0; i < N; i++) {
      array[i] = 3.0;
```

(continues on next page)

(continued from previous page)

```
   }
   printf("Success!\n");
}
```

When we build the program, HPC SDK will choose the CUDA 12.9 toolkit that is included as the default. When we attempt to run it, it fails because code generated with 12.9 does not work with the 12.1 driver:

```
$ nvc -acc -gpu=cc70 app.c
$ ./a.out
Accelerator Fatal Error: This file was compiled: -acc=gpu -gpu=cc70
Rebuild this file with -gpu=cc80 to use NVIDIA Tesla GPU 0
NVIDIA CUDA PTX JIT Compiler was unable to compile device code.
This file was compiled with NVIDIA CUDA Toolkit 12.9, the current CUDA Driver is 12.1.
Rebuild the application with the NVHPC_CUDA_HOME environment variable set to the
↪matching CUDA Tookit.
Please consult the NVIDIA HPC Compilers User's Guide for details.
File: /tmp/app.c
Function: main:5
Line: 5
```

From the error message it follows that the system is unable to execute the Volta GPU instructions on the current system. The embedded Volta PTX could not be compiled, which implies a CUDA driver incompatibility. A way to fix this is to use the installed CUDA 12.1 toolkit at compile time:

```
$ export NVHPC_CUDA_HOME=/usr/local/cuda-12.1
$ nvc -acc -gpu=cc70 app.c
$ ./a.out
Success!
```

OpenMP

Likewise, an OpenMP program will compile but not run:

```
#include <stdio.h>
#define N 1000
int array[N];
int main() {
#pragma omp target loop
   for(int i = 0; i < N; i++) {
      array[i] = 0;

   }
   printf("Success!\n");
}
```

```
$ nvc -mp=gpu -gpu=cc70 app.c
$ ./a.out
Accelerator Fatal Error: Failed to find device function 'nvkernel_main_F1L7_2'! File
↪was compiled with: -gpu=cc70
Rebuild this file with -gpu=cc80 to use NVIDIA Tesla GPU 0
NVIDIA CUDA PTX JIT Compiler was unable to compile device code.
This file was compiled with NVIDIA CUDA Toolkit 12.9, the current CUDA Driver is 12.1.
Rebuild the application with the NVHPC_CUDA_HOME environment variable set to the
↪matching CUDA Tookit.
```

(continues on next page)

(continued from previous page)

```
Please consult the NVIDIA HPC Compilers User's Guide for details.
File: /tmp/app.c
Function: main:4
Line: 7
```

We can also fix it by having NVHPC_CUDA_HOME point at the matching CUDA toolkit location:

```
$ export NVHPC_CUDA_HOME=/usr/local/cuda-12.1
$ nvc -acc -gpu=cc70 app.c
$ ./a.out
Success!
```

C++

In contrast to OpenACC and OpenMP applications that simply terminate when PTX JIT encounters an insufficient CUDA driver version, C++ applications throw a system exception when there is a PTX incompatibility:

```
#include <vector>
#include <algorithm>
#include <execution>
#include <iostream>
#include <assert.h>
int main() {
  std::vector<int> x(1000, 0);
  x[1] = -20;
  auto result = std::count(std::execution::par, x.begin(), x.end(), -20);
  assert(result == 1);
  std::cout << "Success!" << std::endl;
}
```

```
$ nvc++ -stdpar -gpu=cc70 app.cpp
$ ./a.out
terminate called after throwing an instance of 'thrust::system::system_error'
  what():  after reduction step 1: cudaErrorUnsupportedPtxVersion: the provided PTX
↪was compiled with an unsupported toolchain.
Aborted (core dumped)
```

The exception message contains a direct reference to an incompatible PTX, which in turn implies an mismatch between the CUDA toolkit and the CUDA driver version.

We can fix it similarly by setting NVHPC_CUDA_HOME:

```
$ export NVHPC_CUDA_HOME=/usr/local/cuda-12.1
$ nvc++ -stdpar -gpu=cc70 app.cpp
$ ./a.out
Success!
```

12.11. 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.

12.11.1. Supported Fortran Intrinsics Summary Table

Table 19 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 D for double precision real	C for single precision complex Z for double precision complex

Table 19: Supported Fortran Intrinsics

This intrinsic		Return value
ABS	I,S,D	absolute value of the argument.
ACOS		arccosine of the specified argument.
AINT		truncation of the argument to a whole number.
ANINT		nearest whole number of the real argument.
ASIN		arcsine of the argument.
ATAN		arctangent of the argument.
ATAN2		angle in radians of the complex value first-argument + i*second-argument.
COS	S,D,C,Z	cosine of the argument.
COSH		hyperbolic cosine of the argument.
DBLE	S,D	conversion of the argument to double precision real.
DPROD		double precision product of two single precision arguments.
EXP	S,D,C,Z	natural exponential value of the argument.
IAND		result of logical AND of the two integer arguments.

continues on next page

Table 19 – continued from previous page

This intrinsic		Return value
IEOR		result of the boolean exclusive OR of the two integer arguments.
INT	I,S,D	conversion of the argument to integer type.
IOR		result of the boolean inclusive OR of the two integer arguments.
LOG	S,D,C,Z	base-e (natural logarithm) of the argument.
LOG10		base-10 logarithm of the argument.
MAX		maximum value of the arguments.
MIN		minimum value of the arguments.
MOD	I	remainder of the first argument divided by the second argument.
NINT		nearest integer of the real argument.
NOT		logical complement of the integer argument.
REAL	I,S,D	conversion of the argument to real.
SIGN		absolute value of first argument times the sign of second argument.
SIN	S,D,C,Z	sine of the argument.
SINH		hyperbolic sine of the argument.
SQRT	S,D,C,Z	square root of the argument.
TAN		tangent of the argument.
TANH		hyperbolic tangent of the argument.

12.11.2. 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 20: Supported C Intrinsic Double Functions

This intrinsic	Return value
acos	arccosine of the argument.
asin	arcsine of the argument.
atan	arctangent of the argument.
atan2	arctangent of y/x, where y is the first argument, x the second.
cos	cosine of the argument.
cosh	hyperbolic cosine of the argument.
exp	exponential value of the argument.
fabs	absolute value of the argument.
fmax	maximum value of the two arguments
fmin	minimum value of the two arguments
log	natural logarithm of the argument.
log10	base-10 logarithm of the argument.
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 argument.
tanh	hyperbolic tangent of the argument.

Table 21: Supported C Intrinsic Float Functions

This intrinsic	Return value
acosf	arccosine of the argument.
asinf	arcsine of the argument.
atanf	arctangent of the argument.
atan2f	arctangent of y/x, where y is the first argument, x the second.
cosf	cosine of the argument.
coshf	hyperbolic cosine of the argument.
expf	exponential value of the argument.
fabsf	absolute value of the argument.
logf	natural logarithm of the argument.
log10f	base-10 logarithm of the argument.
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 argument.
tanhf	hyperbolic tangent of the argument.

Chapter 13. Using OpenACC

This chapter gives an overview of directive-based OpenACC programming in which compiler directives are used to specify regions of code in Fortran, C and C++ programs to be offloaded from a *host* CPU to an NVIDIA GPU. For complete details on using OpenACC with NVIDIA GPUs, see the OpenACC Getting Started Guide.

13.1. 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.

13.1.1. 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.

13.1.2. 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.

13.1.3. 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
- Subarrays and composite variables in `reduction` clauses
- The `self` clause
- The `default` clause on data constructs

13.1.4. 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.

The NVIDIA C, C++, and Fortran compilers support the `__nv_minctapersm` clause on the OpenACC `parallel`, `kernels`, and `serial` constructs:

```
__nv_minctapersm(n)
```

The argument n must be an integer constant greater than 1. If the argument is not an integer constant, or if its value is less than or equal to 1, the compiler issues a warning and ignores the clause.

The clause requests that at least n thread blocks of each kernel generated for the construct be resident on a streaming multiprocessor. The compiler uses this value as the CUDA launch-bounds minimum number of blocks per multiprocessor, which limits register allocation so that occupancy of n blocks is possible. A larger value leaves fewer registers for each thread.

When the clause is accepted, it takes precedence over `-gpu=maxregcount` for those kernels. The `-gpu=minctapersm:n` option applies the same minimum to every kernel in the compilation; a valid `__nv_minctapersm` clause overrides that option for the construct on which it appears. Like `num_gangs`, `num_workers`, and `vector_length`, the clause may follow a `device_type` clause.

13.1.5. C++ Virtual Functions

The NVIDIA C++ compiler supports a virtual call in an OpenACC compute region when the virtual function written at the call has an explicit `#pragma acc routine` directive and the object is constructed in device code. Put the same directive on each override that can run on the device. A directive on a derived-class override does not stand in for the function written at the call.

A host constructor stores the host virtual-table address in the object. Copying that object to the device copies the address, and the compiler does not replace it with the device virtual table. A virtual call with no explicit directive, and a call through a function pointer or a pointer to a member, are accelerator restrictions. `-Minfo=accel` reports `Indirect function/procedure calls are not supported`, and the region stays on the host. For an example and the full list of restrictions, see C++ Virtual Functions in the OpenACC Getting Started Guide.

13.2. Compiling an OpenACC Program

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

13.2.1. -[no]acc

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

gpu 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]routinepar
Infer parallelism level clause (`gang`, `worker`, `vector`) in implicit routine directives. This sub-option is supported by the C/C++ compilers. The inference is performed for C/C++ regular functions and C++ lambdas by analysing contained parallel loops and acceleration routines as detailed in OpenACC Technical Report 24-1.

[no]routineseq
Compile every routine for the devicee. 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.

Default

By default OpenACC directives are compiled for GPU and sequential CPU host execution (i.e. equivalent to explicitly setting `-acc=gpu,host`).

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
```

Predefined Macros

The following macros corresponding to the target compiled for are added implicitly:

- `__NVCOMPILER_OPENACC_GPU` when the OpenACC directives are compiled for GPU.
- `__NVCOMPILER_OPENACC_MULTICORE` when the OpenACC directives are compiled for multicore CPU.
- `__NVCOMPILER_OPENACC_HOST` when the OpenACC directives are compiled for serial execution on CPU.

13.2.2. -gpu

Used in combination with the -acc, -cuda, -mp, and -stdpar flags to specify options for GPU code generation. The following sub-options 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

cache:{[no]assumesafe|[no]tma[:n]}
Control software cache usage and `cache` directive in OpenACC

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.

ccall-major
Compile for all major supported compute capabilities.

ccnative
Detects the visible GPUs on the system and generates codes for them. If no device is available, the compute capability matching NVCC's default will be used.

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 on the GPU; default at `-O1`. This can be used in conjunction with the global `-M[no]fma` option to explicitly enable/disable FMAs on the CPU or GPU.

[no]implicitsections
Change [do not change] array element references in a data clause into an array section. In C++, the `implicitsections` option will change `update device(a[n])` to `update device(a[0:n])`. In Fortran, it will change `enter data copyin(a(n))` to `enter data copyin(a(:n))`. The default behavior, `noimplicitsections`, can also be changed using rcfiles; for example, one could add `set IMPLICITSECTIONS=0;` to siterc or another rcfile.

[no]interceptdeallocations
Intercept [do not intercept] calls to standard library memory deallocations (e.g. `free`) and call the corresponding CUDA memory deallocation version if address is in pinned or managed memory, regular version otherwise.

keep
Keep the kernel files (.cubin, .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

[no]managed
Allocate [do not allocate] any dynamically allocated data in CUDA Managed memory. Use

`-gpu=nomanaged` with `-stdpar` to prevent that flag's implicit use of `-gpu=managed` when CUDA Managed memory capability is detected. This option is deprecated.

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

minctapersm:n
Specify the minimum number of thread blocks resident on each streaming multiprocessor. The value must be an integer greater than 1. This limit applies to GPU kernels in the compilation and takes precedence over `maxregcount`. An OpenACC `__nv_minctapersm` clause overrides this value for the construct on which it appears.

mem:{separate|managed|unified}
Select GPU memory mode for the generated binary. This controls CUDA memory capability to be utilised such as separate GPU memory only (`separate`), GPU Managed Memory for the dynamically allocated data (`managed`), or system memory aka full CUDA Unified Memory (`unified`). Use of Managed or Unified Memory facilitates simpler programming by eliminating the need to detect all data to be copied into and outside of the code region executing on the GPU.

pinned
Use CUDA Pinned Memory. This option is deprecated.

ptxinfo
Print PTX info

[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

sm_XY
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.

stacklimit:<l>nostacklimit
Sets the limit (l) of stack variables in a procedure or kernel, in KB. This option is deprecated.

tripcount:{host|device|[no]check|[no]warn}
Determine whether the trip count values for loops in compute constructs are calculated on the host (default) or the device. Also can be used to enable [disable] runtime checks and compile-time warnings related to using host vs. device trip count values.

[no]unified
Compile [do not compile] for CUDA Unified memory capability, where system memory is accessible from the GPU. This mode utilizes system and managed memory for dynamically allocated data unless explicit behavior is set through `-gpu=[no]managed`. Use `-gpu=nounified` with `-stdpar` to prevent that flag's implicit use of `-gpu=unified` when CUDA Unified memory capability is detected. This option must appear in both the compile and link lines. This option is deprecated.

[no]unroll
Enable [disable] automatic inner loop unrolling; default at -03

zeroinit
Initialize allocated device memory with zero

Usage

In the following example, the compiler generates code for NVIDIA GPUs with compute capabilities 8.0 and 9.0.

```
$ nvfortran -acc -gpu=cc80,cc90 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 -cuda, -mp, or -stdpar.

DWARF Debugging Formats

Use the `-g` option to enable generation of full 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`.

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

13.4. OpenACC with CUDA Unified Memory

When developing OpenACC source for a target supporting CUDA Unified Memory, you can take advantage of a simplified approach to programming because there is no need for data clauses and directives, either in full or in part, depending on the exact memory capability the target supports and the compiler options used.

The discussion in this section assumes you have become familiar with the Separate, Managed, and Unified Memory Modes covered in the *Memory Model* and *Managed and Unified Memory Modes* sections.

In Managed Memory Mode, only dynamically-allocated data are implicitly managed by the CUDA runtime; OpenACC data clauses and directives are therefore not needed for movement of this "managed" data. Data clauses and directives are still required to handle static data (C static and extern variables, Fortran module, common block and save variables) and function local data.

In Unified Memory Mode, all data is managed by the CUDA runtime. Explicit data clauses and directives are no longer required to indicate which data should reside in GPU memory. All variables are accessible from the OpenACC compute regions executing on the GPU. The NVHPC compiler implementation closely adheres to the shared memory mode detailed in the OpenACC specification, meaning that `copy`, `copyin`, `copyout`, and `create` clauses will not result in any device allocation or data transfer. The `device_resident` clause is still honored as in discrete memory mode and results in an allocation of data only accessible from device code. Device memory can also be allocated or deallocated in OpenACC programs in Unified Memory Mode by using the `acc_malloc` or `acc_free` API calls.

Understanding Data Movement

In the absence of visible data clauses or directives, when the compiler encounters a compute construct it attempts to determine what data is required for correct execution of the region on the GPU. When the compiler is unable to determine the size and shape of data needing to be accessible on the device, it behaves as follows:

- In Separate Memory Mode, the compiler emits an error requesting an explicit data clause be added to specify size/shape of the data to be copied.
- In Managed Memory Mode (`-gpu=mem:managed`), the compiler assumes the data is allocated in managed memory and thus is accessible from the device; if this assumption is wrong, if the data was defined globally or is located on the CPU stack, the program may fail at runtime.
- In Unified Memory Mode (`-gpu=mem:unified`), all data is accessible from the device making information about size and shape unnecessary.

Take the following example in C:

```
void set(int* ptr, int i, int j, int dim){
  int idx = i * dim + j;
  return ptr[idx] = someval(i, j);
}

void fill2d(int* ptr, int dim){
#pragma acc parallel loop
  for (int i = 0; i < dim; i++)
    for (int j = 0; j < dim; j++)
      set(ptr, i, j, dim);
}
```

In Separate Memory Mode, the only way to guarantee correctness for this example is to change the line with the `acc` directive as follows:

```
#pragma acc parallel loop create(ptr[0:dim*dim]) copyout(ptr[0:dim*dim])
```

This change explicitly instructs the OpenACC implementation about the precise data segment used within the parallel loop.

In Unified Memory Mode, that is, by compiling with `-acc -gpu=mem:unified` and executing on a platform with unified memory capability, the `create` and `copyout` clauses are not required.

The next example, in Fortran, illustrates how a global variable can be accessed in an OpenACC routine without requiring any explicit annotation.

```
module m
integer :: globmin = 1234
contains
subroutine findmin(a)
!$acc routine seq
  integer, intent(in)  :: a(:)
  integer :: i
  do i = 1, size(a)
    if (a(i) .lt. globmin) then
      globmin = a(i)
    endif
  end do
end subroutine
end module m
```

Compile the example above for Unified Memory Mode:

```
nvfortran -acc -gpu=mem:unified example.f90
```

The source does not need any OpenACC directives to access module variable `globmin`, to either read or update its value, in the routine invoked from CPU and GPU. Moreover, any access to `globmin` will be made to the same exact instance of the variable from CPU and GPU; its value is synchronized automatically. In Separate or Managed Memory Modes, such behavior can only be achieved with a combination of OpenACC `declare` and `update` directives in the source code.

In most cases, migrating existing OpenACC applications written for Separate Memory Mode should be a seamless process requiring no source changes. Some data access patterns, however, may lead to different results produced during application execution in Unified Memory Mode.

Applications which rely on having separate data copies in GPU memory to conduct temporary computations on the GPU – without maintaining data synchronization with the CPU – pose a challenge for migration to Unified Memory.

For the following Fortran example, the value of variable `c` after the last loop will differ depending on whether the example is compiled with or without `-gpu=mem:unified`.

```
b(:) = ...
c = 0

!$acc kernels copyin(b) copyout(a)
!$acc loop
do i = 1, N
  b(i) = b(i) * i
end do
!$acc loop
do i = 1, N
  a(i) =  b(i) + i
```

(continues on next page)

(continued from previous page)

```
end do
!$acc end kernels

do i = 1, N
  c = c + a(i) + b(i)
end do
```

Without Unified Memory, array `b` is copied into the GPU memory at the beginning of the OpenACC `kernels` region. It is then updated in the GPU memory and used to compute elements of array `a`. As instructed by the data clause `copyin(b)`, `b` is not copied back to the CPU memory at the end of the `kernels` region and therefore its initial value is used in the computation of `c`. With `-acc -gpu=mem:unified`, the updated value of `b` in the first loop is automatically visible in the last loop leading to a different value of `c` at its end. **Implications of Asynchronous Execution**

Additional complexities can arise when dealing with asynchronous execution, particularly when CPU-GPU shared data is accessed within `async` compute regions instead of using an independent data copy on GPU. The programmer should be especially careful about accessing local variables in asynchronous GPU code. Unless the GPU code execution is explicitly synchronized before the end of the scope in which local variables are defined, the GPU can access stale data thus resulting in undefined behavior. Consider the following OpenACC C example, where a local array is used to hold temporary data on the GPU:

```
void bar() {
  int x[N];
  #pragma acc enter data create(x[0:N]) async
  #pragma acc parallel loop async
  for (int i = 0; i < N; i++)
    x[i] = i;
  ...
  #pragma acc exit data delete(x[0:N]) async
}
```

When compiled for Separate Memory Mode, the `bar()` function creates a copy of the array `x` in GPU memory and initializes it as written in the `loop` construct. That copy is eventually deleted. In Unified Memory Mode, however, the compiler ignores the `acc enter data` and `acc exit data` directives, so the `loop` construct executed on the GPU accesses the array `x` in local CPU memory. Moreover, since all constructs in this example are made asynchronous, the access to `x` on the GPU leads to undefined behavior of the program because the variable `x` goes out of scope once the `bar()` function finishes.

Here is a similar example, where the local array `x` is dynamically allocated:

```
void bar() {
  int *x = (int *)malloc(N * sizeof(int));
  #pragma acc enter data create(x[0:N]) async
  #pragma acc parallel loop async
  for (int i = 0; i < N; i++)
    x[i] = i;
  ...
  #pragma acc exit data delete(x[0:N]) async
  free(x);
}
```

To help detect programming mistakes like this, the NVIDIA OpenACC Runtime provides the `NVCOMPILER_ACC_CHECK_UNIFIED` environment variable. When set to a non-zero integer value, it enables the runtime to emit information to the standard error output about variables still used on the device after they have been deallocated on the host.

For the example with the x array dynamically allocated with `malloc`, you can compile the application normally and run it as follows:

```
$ nvc -acc -gpu=mem:unified app.c
$ NVCOMPILER_ACC_CHECK_UNIFIED=1 ./a.out
Deallocated pointer 0x4a1d00000a80 still referenced on device! File ./app.c, function
↪bar, line 6, variable 'x[:1000]'.
```

When the x array is automatically allocated in stack memory, the OpenACC runtime will be unable to recognize the problem on its own. You can compile the application with the `-Mstack_frame_trace` switch added, which will result in generating additional facilities that will assist the runtime when automatic memory is deallocated:

```
$ nvc -acc -gpu=mem:unified -Mstack_frame_trace app.c
$ NVCOMPILER_ACC_CHECK_UNIFIED=1 ./a.out
Stack pointer 0xffffd71a13c0 still referenced on device! File ./app.c, function bar,
↪line 6, variable 'x[:]'.
```

Note that the `NVCOMPILER_ACC_CHECK_UNIFIED` environment variable as well as the compiler switch `-Mstack_frame_trace`, are designed for diagnostic purposes. Either of them can add run-time overhead, impacting application performance. We recommend avoiding their use when the application is deployed in a production environment.

Performance Considerations

In Unified Memory Mode, the OpenACC runtime may leverage data action information such as `create/delete` or `copyin/copyout` to communicate preferable data placement to the CUDA runtime by means of memory hint APIs as elaborated in the following blog post on the NVIDIA website: Simplifying GPU Application Development with Heterogeneous Memory Management. Such actions originate either from explicit data clauses in the source code or via implicit data movement generated by the compiler. This approach can minimize the amount of automatic data migration and may let a developer fine-tune application performance. For the C example above, while adding the data clauses `create(ptr[0:dim*dim])` and `copyout(ptr[0:dim*dim])` becomes optional with `-gpu=mem:unified`, their uses in the OpenACC `parallel loop` directive may improve performance.

13.5. OpenACC Memory Bandwidth Tuning

This section describes the OpenACC features you can use to preload data and increase the bytes in flight in OpenACC compute regions. It assumes you are familiar with bytes in flight and the data preload variants covered in the *Memory Bandwidth Tuning* section.

The NVIDIA HPC compilers provide the following OpenACC features for each preload variant:

Table 22: OpenACC Features for Data Preload

Preload variant	OpenACC feature	Effect
Register preload	`unroll` modifier of the `vector` or `seq` loop clause	Unrolls the loop so that more iterations' loads are clustered and preloaded into registers, adding bytes in flight on top of the automatic register preload.
Shared memory preload	`cache` directive	Copies array sections into CUDA shared memory, using TMA where enabled.
L1 preload	`prefetch` directive	Prefetches array sections into the L1 cache.

The NVIDIA HPC Fortran, C and C++ compilers support the `cache` directive. The `unroll` modifier and the `prefetch` directive are currently supported only by the NVIDIA HPC Fortran compiler. The examples in this section are in Fortran.

For the best results, pair the `cache` and `prefetch` directives with loop unrolling or loop tiling, so that each preload covers the data of several iterations. Use `-Minfo=accel` to see what the compiler preloads.

The `__nv_minctapersm(n)` clause, described in *OpenACC Extensions*, sets the minimum number of thread blocks (CTAs) per SM for the generated kernels, like the `minBlocksPerMultiprocessor` parameter of `__launch_bounds__` described in the CUDA C++ Programming Guide. To fit `n` CTAs on each SM, the compiler limits the registers each thread uses, so you can use this clause to control the resources available for preload and therefore the occupancy. For example:

```
!$acc parallel loop gang vector collapse(2) __nv_minctapersm(2)
```

```
#pragma acc parallel loop gang vector collapse(2) __nv_minctapersm(2)
```

13.5.1. Loop Unrolling

The compiler may unroll inner loops automatically at `-O3`; `-gpu=[no]unroll` enables [disables] the automatic unrolling. In Fortran, to set the unroll factor of a vector or sequential loop explicitly, use the `unroll` modifier of the `vector` or `seq` loop clause:

```
!$acc loop vector(unroll:n)
!$acc loop seq(unroll:n)
```

where `n` is the unroll factor. In the following example, each thread clusters the loads of four iterations of the `j` loop:

```
!$acc parallel loop gang
do i = 1, m
  s = 0.0d0
  !$acc loop vector(unroll:4) reduction(+:s)
  do j = 1, n
    s = s + a(j,i) * x(j)
  end do
  y(i) = s
end do
```

The `-Minfo=accel` output shows the unroll factor:

```
!$acc loop vector(128) unroll(4) ! threadidx%x
```

For a sequential loop, such as the `j` loop of a `gang vector` loop over `i`, use `seq(unroll:n)`:

```
!$acc parallel loop gang vector
do i = 1, m
  s = 0.0d0
  !$acc loop seq(unroll:4)
  do j = 1, n
    s = s + a(j,i) * x(j)
  end do
  y(i) = s
end do
```

The `-Minfo=accel` output reports the unrolling:

```
!$acc loop seq
   Loop unrolled 4 times
```

A larger unroll factor may increase the bytes in flight but may also increase the number of registers per thread.

For the complete specification of the `unroll` modifier, refer to the OpenACC technical report of November 2026 on the OpenACC www.openacc.org website.

13.5.2. The cache Directive

The OpenACC `cache` directive preloads array sections into CUDA shared memory:

```
!$acc cache(arr(lb:ub))
```

```
#pragma acc cache(arr[lb:len])
```

The NVIDIA HPC compilers copy the cached sections into shared memory using a high-bandwidth copy. Starting with the 26.11 release, on GPUs with compute capability 9.0 and later, the copy can use the Tensor Memory Accelerator (TMA). TMA is not used by default; enable it with the `cache` sub-option of `-gpu`:

cache:tma[:n]
: Use TMA for `cache` directives on GPUs with compute capability 9.0 and later, for contiguous cached array sections whose size is at or above a threshold. The compiler has a default size threshold below which it does not use TMA; use `n` to set the threshold in bytes. TMA is used only when the cached section is contiguous in memory.

cache:notma
: Do not use TMA for `cache` directives.

For example:

```
$ nvfortran -acc -gpu=cc90,cache:tma -Minfo=accel prog.f90
$ nvc -acc -gpu=cc90,cache:tma -Minfo=accel prog.c
```

The cached sections must fit in shared memory, which constrains the size of the sections and can limit occupancy. The `cache` directive works best for contiguous sections with no scatter or gather, and when paired with loop tiling, as shown in *Loop Tiling with Preload*. The `cache` directive also imposes some constraints on how cached data is used. In particular, all references to a cached array inside the cache scope must fall within the cached region of the array. The compiler does not verify this, and code which does not adhere to this restriction will compile, but may crash or produce incorrect results.

For the complete specification of the `cache` directive, refer to the OpenACC technical report of November 2026 on the OpenACC www.openacc.org website.

13.5.3. The prefetch Directive

The `prefetch` directive preloads array sections into the L1 cache:

```
!$acc prefetch(arr(lb:ub))
```

The `prefetch` directive has nearly no constraints and no impact on occupancy, because the prefetched data are not held in registers or shared memory. The prefetched data do occupy the L1 cache, however, so prefetching can evict other data from L1 and affect the performance of other memory accesses that rely on L1, such as loads of reused data or register spills. The placement of the data is less predictable than with the `cache` directive: prefetched data may be evicted from L1 before they are used.

The `prefetch` directive is supported by the NVIDIA HPC Fortran compiler starting with the 26.11 release and requires CUDA 13.4 or later. For example:

```
$ nvfortran -acc -gpu=cc90,cuda13.4 -Minfo=accel prog.f90
```

For the complete specification of the `prefetch` directive, refer to the OpenACC technical report of November 2026 on the OpenACC www.openacc.org website.

13.5.4. Loop Tiling with Preload

By scheduling a preload paired with a loop tile at different levels of a loop nest, you control the volume of the preloaded data and therefore the bytes in flight. Consider the following example with two nested loops, a SAXPY-like update of the `n` x `m` arrays `x` and `y`:

```
!$acc parallel loop gang vector collapse(2)
do i = 1, m
  do j = 1, n
    y(j,i) = alpha * x(j,i) + y(j,i)
  end do
end do
```

You can tile either the inner `j` loop or the outer `i` loop and schedule the preload at the level of the tile, using either the `cache` or the `prefetch` directive.

When you tile the inner `j` loop, the tile loop becomes part of the gang loop, and the preload is scheduled at the start of each `j` tile. Each gang then preloads `TILE` contiguous elements of `x` and of `y`, so the preload volume per gang is set by `TILE` alone. The following version caches each tile in shared memory:

```
integer, parameter :: TILE = 256

!$acc parallel loop gang collapse(2)
do i = 1, m
  do jt = 1, n, TILE
    !$acc cache(x(jt:min(jt+TILE-1,n),i), y(jt:min(jt+TILE-1,n),i))
    !$acc loop vector
    do j = jt, min(jt+TILE-1, n)
      y(j,i) = alpha * x(j,i) + y(j,i)
    end do
  end do
end do
```

Each tile of 256 double precision elements is large enough for the compiler's TMA size threshold, so with `-gpu=cache:tma` the compiler uses TMA for the copy:

```
!$acc loop gang collapse(2) ! blockidx%x
   Cached references to size [256] block of x,y
   CUDA TMA copy is used for caching size [256] block of y,x
```

When you tile the outer `i` loop instead, the preload is scheduled at the gang level, above both loops. Each gang then preloads the whole block of `TILE` columns of `x` and of `y`, so the preload volume per gang is `n` x `TILE` elements: moving the preload up one loop level multiplies the volume by the extent of the inner loop. The following version prefetches each block into `L1`:

```
integer, parameter :: TILE = 4

!$acc parallel loop gang
do it = 1, m, TILE
  !$acc prefetch(x(1:n,it:min(it+TILE-1,m)), y(1:n,it:min(it+TILE-1,m)))
  !$acc loop vector collapse(2)
  do i = it, min(it+TILE-1, m)
    do j = 1, n
      y(j,i) = alpha * x(j,i) + y(j,i)
    end do
  end do
end do
```

The `-Minfo=accel` output reports the prefetches:

```
!$acc loop gang ! blockidx%x
   Prefetching x: dynamic schedule, cooperative (vector)
   Prefetching y: dynamic schedule, cooperative (vector)
```

You can replace `cache` with `prefetch` in the inner-loop tile to preload into `L1`. In the outer-loop tile, the size of the preloaded block depends on `n`, so the block can be cached in shared memory with the `cache` directive only when `n` is a constant. In C and C++, the size of a cached section must likewise be a compile-time constant.

For the best results, multiple arrays may need to be preloaded, as `x` and `y` are in both examples above: the bytes in flight include the loads of every array the loop reads, and an array that is not preloaded can still stall the computation.

Choose the tile size so that the preload is large enough to increase the bytes in flight but still fits in registers, shared memory, or `L1` within the occupancy limit.

13.6. 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`.

Note: 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);
```

(continues on next page)

(continued from previous page)

```
  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
```

(continues on next page)

(continued from previous page)

```
...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.

13.7. OpenACC and CUDA Graphs

NVIDIA provides an optimized model for work submission onto GPUs called CUDA Graphs. A graph is a series of operations, such as kernel launches and other stream-oriented tasks, connected by their dependencies. A graph can be defined once, "captured", then launched repeatedly. This has potential benefits in reducing launch latencies and other overheads associated with kernel setup.

A complete write-up explaining CUDA Graphs and the CUDA API for graph definition, instantiation, and execution can be found in Chapter 3 of the CUDA C Programming Guide. In OpenACC, we currently expose just the minimal set of operations to allow capture and replay of a graph containing OpenACC compute regions and data directives. The code executed between a "begin capture" call, `accx_begin_capture_async()`, and the "end capture" call, `accx_end_capture_async()`, is called the capture region.

The CUDA graph API captures (or records) all the device work between accx_begin_capture_async and accx_end_capture_async. The host code in the capture region will be executed once normally, with the exception that no device work is actually executed on the device. Instead, a graph object is created that can be used to replay the captured work multiple times.

Note: Graph capture is similar to a closure concept in many programming languages, like lambda-functions in C++. In lambda-function terms, CUDA graphs capture all the variables by value. That means that all the FIRSTPRIVATE scalars, array shapes, and those derived types, arrays and scalar addresses for data resident on the GPU, are baked into the graph object and cannot be altered. The device data behind the pointers, of course, can be updated by the graph execution normally, and updated by the host between replays.

It is important to understand both what can and cannot be captured within a CUDA Graph capture region:

- Asynchronous data directives, including `data create`, can be captured. The OpenACC runtime will use the stream-ordered `cudaMallocAsync()` call in the capture region for variables which need allocation in data clauses, an API call allowed in CUDA Graphs.
- Asynchronous compute regions, preferably ACC `parallel` regions, can be captured. For ACC `kernels` regions, verify that no work is performed on the host. Host compute sections cannot be captured.
- Asynchronous ACC `update host (self)` and `update device` directives can be captured. The host and device addresses which are captured must be valid during the graph replay/execution.

- Since only the device work is captured and replayed, any data dependencies between the host and device inside the capture region are erroneous. For example, downloading data from the device, processing it on host and uploading it back to the device within the capture region is invalid.
- Host code, even host code containing conditionals, can occur within a capture region. Note though that the path taken through the host code will be the path captured by the graph, i.e. the conditionals must likely be consistent during the replay for correct results. Host code which updates host variables, such as `i=i+1` will not be captured in the graph, which might affect proper indexing into device-side arrays or other kernel arguments.
- Similarly, device work initiated in host code loops can be captured in the CUDA Graph. The graph will not contain a notion of looping, just the sequence of device operations submitted to the device during the loop.
- Subroutine and function calls within a capture region, which contain further compute regions or other work which runs on the device, are captured. Care must be taken that the device data addresses passed to the kernels are valid throughout graph execution, and don't come and go based on stack addresses or something similar.
- Codes which double-buffer, or ping-pong between source and destination arrays that are input on odd iterations and output on even iterations (for example, the snippet shown below), can be accommodated by capturing two graphs: one per even iteration, one per odd iteration.

```
int* src;
int* dest;
while (err > tolerance) {
  for (int i = 0; i < N; i++) {
    dest[i] = foo(src[i]);
  }
  err = bar(dest);
  int* tmp = dest;
  dest = src;
  src = tmp;
}
```

- Many CUDA library calls, like cublas, etc. can occur in a captured region. Setup for the library calls, such as creating handles, and computing and allocating workspace requirements, should be done before the capture region.
- Graph capturing is thread-safe with respect to each async queue. Host threads can independently capture graphs using different async queues.
- When `-gpu=tripcount:device` is used, loop trip counts can vary between runs of the same captured graph, as long as the trip count is updated on the device.

The OpenACC API follows the basic portion of the CUDA Graph API fairly closely. The major difference is OpenACC includes the `cudaGraphInstantiate()` call as part of the end capture function.

From Fortran, the graph type is defined in the OpenACC module:

```
type, bind(c) :: acc_graph_t
  type(c_ptr) :: graph
  type(c_ptr) :: graph_exec
end type acc_graph_t
```

These subroutines are available in the OpenACC runtime. Here, pGraph is type(acc_graph_t) and async is just the asynchronous queue value:

```
subroutine accx_async_begin_capture( async )
subroutine accx_async_end_capture( async, pGraph )
subroutine accx_graph_launch( pGraph, async )
subroutine accx_graph_delete( pGraph )
type(c_ptr) function accx_get_graph( pGraph )
type(c_ptr) function accx_get_graph_exec( pGraph )
```

From C, the graph type is defined in OpenACC.h:

```
typedef struct { void *graph; void *graph_exec; } acc_graph_t;
```

These void functions are available in the OpenACC runtime:

```
extern void accx_async_begin_capture(long async);
extern void accx_async_end_capture(long async, acc_graph_t *pgraph);
extern void accx_graph_launch(acc_graph_t *pgraph, long async);
extern void accx_graph_delete(acc_graph_t *pgraph);
extern void *accx_get_graph(acc_graph_t *pgraph);
extern void *accx_get_graph_exec(acc_graph_t *pgraph);
```

We will use a simple Fortran example code which demonstrates some of the modifications needed to use CUDA Graphs from OpenACC. The original serial code for a conjugate gradient iterative solver:

```
subroutine RunCG(N, A, b, x, tol, max_iter)
  implicit none
  integer, intent(in) :: N, max_iter
  real(WP), intent(in) :: A(N, N), b(N), tol
  real(WP), intent(inout) :: x(N)

  real(WP) :: alpha, rr0, rr
  real(WP), allocatable :: Ax(:), r(:), p(:)
  integer :: it, i

  allocate(Ax(N), r(N), p(N))

  call symmatvec(N, N, A, x, Ax)
  do i = 1, N
    r(i) = b(i) - Ax(i)
    p(i) = r(i)
  enddo
  rr0 = dot(N, r, r)

  do it = 1, max_iter
    call symmatvec(N, N, A, p, Ax)
    alpha = rr0 / dot(N, p, Ax)

    do i = 1, N
      x(i) = x(i) + alpha * p(i)
      r(i) = r(i) - alpha * Ax(i)
    enddo

    rr = dot(N, r, r)

    print*, "Iteration ", it, " residual: ", sqrt(rr)
    if (sqrt(rr) <= tol) then
      deallocate(Ax, r, p)
      return
```

(continues on next page)

(continued from previous page)

```
    endif
    do i = 1, N
      p(i) = r(i) + (rr / rr0) * p(i)
    enddo
    rr0 = rr
  enddo

  deallocate(Ax, r, p)

end subroutine RunCG
```

For this exercise we wish to put the `do it = 1,max_iter` work for each iteration into a CUDA graph. Step one is to port the code to OpenACC, keeping in mind that we want to use asynchronous queues. We annotate the dot function with OpenACC directives like this:

```
function dot(N, x, y) result(r)
  integer, intent(in) :: N
  real(WP), intent(in) :: x(N), y(N)
  integer :: i
  real(WP) :: r

  r = 0.d0
  !$acc parallel loop present(x, y) reduction(+:r) async(1)
  do i = 1, N
    r = r + x(i) * y(i)
  enddo
  !$acc wait(1)
end function dot
```

We write the symmetric matrix multiply like this:

```
subroutine symmatvec(M, N, AT, x, Ax)
  implicit none
  integer, intent(in) :: M, N
  real(WP), intent(in) :: AT(N, M), x(N)
  real(WP), intent(out) :: Ax(M)

  integer :: i, j
  real(WP) :: s

  ! Note: Since A is symmetric, we can use the "transpose"
  ! for better memory access here
  !$acc parallel loop gang present(AT, x, Ax) async(1)
  do i = 1, M
    s = 0.d0
    !$acc loop vector reduction(+:s)
    do j = 1, N
      s = s + AT(j,i) * x(j)
    end do
    Ax(i) = s
  end do
end subroutine
```

And now our main loop of the conjugate gradient solver looks like this:

```
do it = 1, max_iter
  call symmatvec(N, N, A, p, Ax)
```

(continues on next page)

(continued from previous page)

```
  alpha = rr0 / dot(N, p, Ax)

  !$acc parallel loop gang vector async(1)
  do i = 1, N
    x(i) = x(i) + alpha * p(i)
    r(i) = r(i) - alpha * Ax(i)
  enddo

  rr = dot(N, r, r)

  print*, "Iteration ", it, " residual: ", sqrt(rr)
  if (sqrt(rr) <= tol) exit

  !$acc parallel loop gang vector async(1)
  do i = 1, N
    p(i) = r(i) + (rr / rr0) * p(i)
  enddo
  rr0 = rr
enddo
```

Step 2 is to prepare the code for running under CUDA Graphs. There is a lot of host code executing in the main loop. While the `dot()` function runs on the GPU, the rest of the statement `alpha = rr0 / dot(...)` runs on the host. Similarly, the 2nd `dot()` call returns its value to the host. The print statement occurs on the host, as does the residual check. Finally, this iteration's value for rr is moved to rr0 in the last statement of the loop, on the host.

The dot product is tricky. We wish to compute the dot product on the GPU, and leave the result on the GPU, so the reduction variable must be present on the GPU. Here, we change the function call to a subroutine, and remove the initialization which is outside of the parallel region:

```
subroutine dot(N, x, y, r)
  implicit none
  integer, intent(in) :: N
  real(WP), intent(in) :: x(N), y(N)
  integer :: i
  real(WP) :: r

  !$acc parallel loop present(x, y, r) reduction(+:r) async(1)
  do i = 1, N
    r = r + x(i) * y(i)
  enddo
end subroutine dot
```

We add one serial kernel to do some of the swapping between rr0 and rr, as well as zeroing out the scalar that will hold the dot product reduction, and move the print and check outside of the GPU capture region, replaced by a update host operation. The finished loop, complete with graph control, looks like this:

```
do it = 1, max_iter
  if (it .eq. 1) then  ! First time capture
    call accx_async_begin_capture(1)

    call symmatvec(N, N, A, p, Ax)
    call dot(N, p, Ax, rden)

    !$acc serial async(1)
```

(continues on next page)

(continued from previous page)

```
    rr0 = rr
    alpha = rr0 / rden
    rden = 0.0d0
    rr = 0.0d0
    !$acc end serial

    !$acc parallel loop gang vector async(1)
    do i = 1, N
      x(i) = x(i) + alpha * p(i)
      r(i) = r(i) - alpha * Ax(i)
    enddo

    call dot(N, r, r, rr)

    !$acc update host(rr) async(1)

    !$acc parallel loop gang vector async(1)
    do i = 1, N
      p(i) = r(i) + (rr / rr0) * p(i)
    enddo
    call accx_async_end_capture(1, graph)
  endif
  ! Always launch, then wait
  call accx_graph_launch(graph, 1)
  !$acc wait(1)

  rra(it) = rr
  if (sqrt(rr) <= tol) exit
enddo
```

Step 3 is to compile, run, and profile the result. No special compiler options are needed besides -acc=gpu. When running, you may be advised to set the `NVCOMPILER_ACC_USE_GRAPH` environment variable. This is currently necessary to properly set the OpenACC runtime for graph capture. Failure to abide by the guidelines above may result in wrong answers, which can be hard to debug. See the following sections on how to use environment variables to help. A common issue is that the pointers passed to the device kernels during graph playback will be the same every time. Make sure that is the case between iterations in the code without graph capture.

The Nsight Systems tool has very good support for profiling CUDA graphs. The timeline view will provide information on whether you have reduced the launch overhead gaps between the GPU kernels. Figure 1 shows a timeline of the iterations of the original OpenACC loop:

Figure 2 shows a timeline of the iterations when using CUDA Graphs. When the size N is less than a few thousand, launch latency becomes a major contributor to the overall time and here we can see about a 2x speedup:

You can see a more-detailed trace of the CUDA Graph components by adding the `--cuda-graph-trace=node` option to the nsys profile command.

The above loop demonstrates several of the guidelines outlined at the top of this section, namely, capturing compute regions, whether at the top level or in subprogram units, capturing data movement, and restructuring code regions to minimize or eliminate the host code within a capture region. And the minimal API to begin capture, end capture, then launch the captured graph.

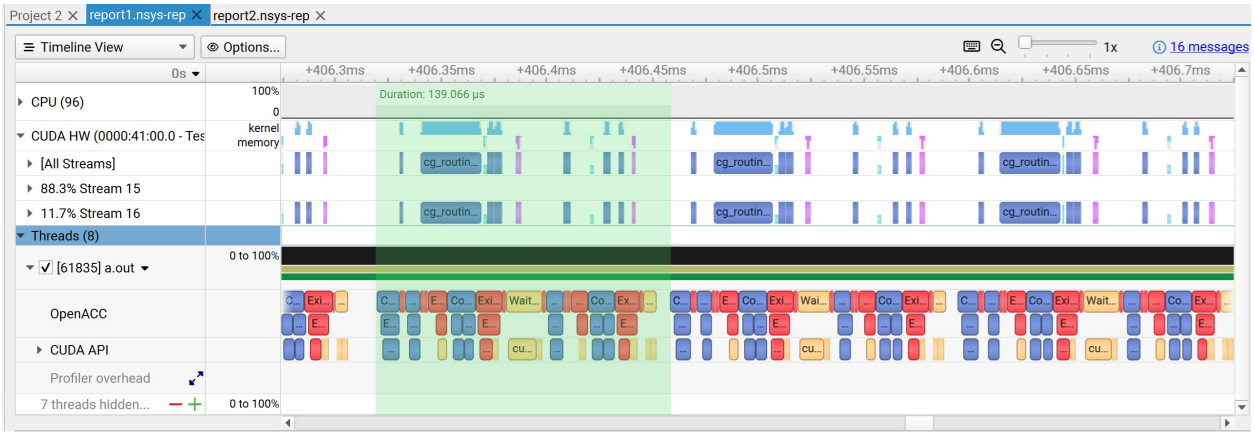


Figure 1: Nsight Systems Report1 Timeline

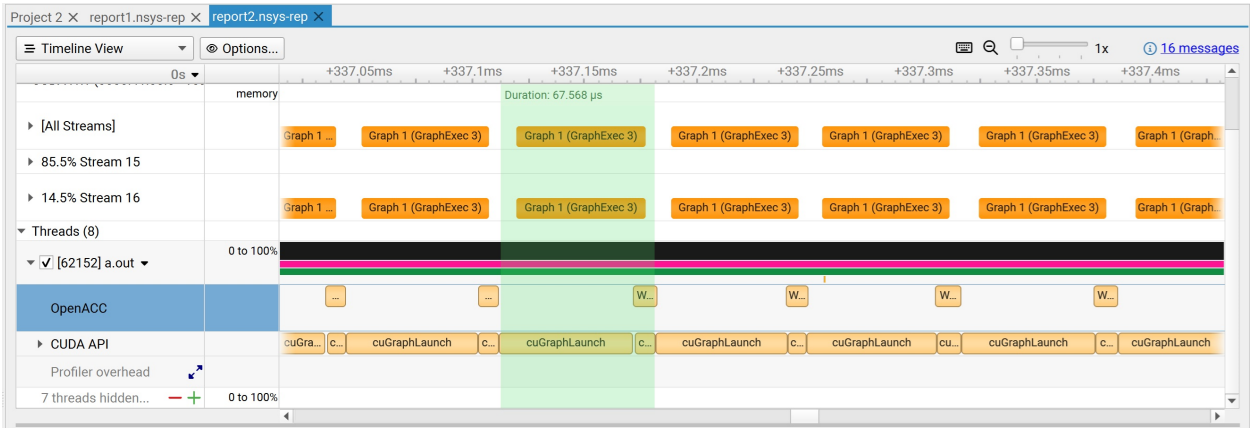


Figure 2: Nsight Systems Report2 Timeline

13.8. Host and Device Trip Count Options

The `-gpu=tripcount` option controls whether the trip counts for loops in a compute construct, such as `acc parallel loop`, are calculated on the host or on the device. The default behavior of the NVHPC compilers is to use the values calculated on the host, though the OpenACC specification states that trip count values should be calculated on the device. We have chosen to maintain, as-is, the default behavior so as not to interfere with existing applications that currently depend on it for correctness. To ensure compliance with the specification, please use the `-gpu=tripcount:device` option. To maintain the default behavior, please use `-gpu=tripcount:host` or do not specify a `-gpu=tripcount` option.

To emit a warning at compile-time that an OpenACC program may be using host values for trip counts, use `-gpu=tripcount:warn`, or use `-gpu=tripcount:nowarn` to disable these warnings.

To check at runtime whether the host and device values for trip counts are the same, use `-gpu=tripcount:check`. Set the environment variable `NVCOMPILER_ACC_CHECK_TRIPCOUNT` to enable reporting of any differences discovered. To disable these checks, use `-gpu=tripcount:nocheck`.

13.8.1. When to Use `-gpu=tripcount:device` or `-gpu=tripcount:host`

Consider the following example code snippet:

```
real :: array(1000, 10)
integer :: i, j, n, m

!$acc data create(n, m) copy(array)

!$acc kernels
n = 1000
m = 10
!$acc end kernels

!$acc parallel loop defualt(none) collapse(2)
do j=1,m
  do i=1,n
    array(i, j) = i+j
  end do
end do
```

The trip count variables `n` and `m` are created on the device, and then their values are set on the device in the `acc kernels` construct. Their values are not set on the host. Therefore, when the parallel loop is run on the device, if the host values for `n` and `m` are used, the loop will not run for the correct number of iterations. In this and similar cases, to ensure the correctness of the program, `-gpu=tripcount:device` should be used.

In cases where the values of `n` and `m` are set on the host, it is sufficient to rely on the default behavior or to specify `-gpu=tripcount:host`. There are two ways to verify whether or not the program's correctness may be affected by the use of `-gpu=tripcount:device` versus `-gpu=tripcount:host`. The `-gpu=tripcount:check` option can be used to detect discrepancies between host and device values for trip counts at runtime, and the `-gpu=tripcount:warn` option can be used to issue compile-time warnings that host values for trip counts may be used.

Note: For CUDA Graphs, `-gpu=tripcount:device` allows trip counts to vary between runs for captured graphs on the device, as long as the trip count is updated on the device. This behavior can affect the correctness of CUDA Graphs, and some applications may require this option to use CUDA Graphs correctly.

13.9. 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 23: Supported Environment Variables

Use this environment variable...	To do this...
NVCOMPILER_ACC_CHECK_TRIPCOUNT	Enable output for checking differences between host and device trip counts when `-gpu=tripcount:check` is used.
NVCOMPILER_ACC_CUDA_PROFSTOP	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_DEVICE_NUM	Sets the default device number to use. NVCOMPILER_ACC_DEVICE_NUM. Specifies 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.
ACC_DEVICE_NUM	Legacy name. Superseded by NVCOMPILER_ACC_DEVICE_NUM.
NVCOMPILER_ACC_DEVICE_TYPE	Sets the default device type to use for OpenACC regions. NVCOMPILER_ACC_DEVICE_TYPE. Specifies which accelerator device to use when executing accelerator regions when 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 OpenACC implementation may be the strings NVIDIA, MULTICORE or HOST
ACC_DEVICE_TYPE	Legacy name. Superseded by NVCOMPILER_ACC_DEVICE_TYPE.
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	With no argument, a debug message will be written to stderr for each kernel launch and/or data transfer. When set to an integer value, the value is used as a bit mask to print information about: 1: kernel launches 2: data transfers 4: region entry/exit 8: wait operations or synchronizations with the device 16: device memory allocates and deallocates
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.

13.10. 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.

Note: 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 concurently.

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.

13.11. 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.

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

13.11.1. 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.

13.11.2. Runtime Library Routines

Table 24 lists and briefly describes the runtime library routines supported by the NVIDIA HPC Compilers in addition to the standard OpenACC runtine API routines.

Table 24: Accelerator Runtime Library Routines

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

Chapter 14. Using OpenMP

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. For general information about using OpenMP and to obtain a copy of the OpenMP specification, refer to the OpenMP organization's website.

The NVFORTRAN, NVC++, and NVC compilers support a subset of the OpenMP Application Program Interface for CPUs and GPUs. In defining this subset, we have focused on OpenMP 5.0 features that will enable CPU and GPU targeting for OpenMP applications with a goal of encouraging programming practices that are portable and scalable. For features that are to be avoided, wherever possible, the directives and API calls related to those features are parsed and ignored to maximize portability. Where ignoring such features is not possible, or could result in ambiguous or incorrect execution, the compilers emit appropriate error messages at compile- or run-time.

OpenMP applications properly structured for GPUs, meaning they expose massive parallelism and have relatively little or no synchronization in GPU-side code segments, should compile and execute with performance on par with or close to equivalent OpenACC. Codes that are not well-structured for GPUs may perform poorly but should execute correctly.

Use the `-mp` compiler switch to enable processing of OpenMP directives and pragmas. The most important sub-options to `-mp` are the following:

- `gpu`: OpenMP directives are compiled for GPU execution plus multicore CPU fallback; this feature is supported on NVIDIA V100 or later GPUs.
- `multicore`: OpenMP directives are compiled for multicore CPU execution only; this sub-option is the default.

Predefined Macros

The following macros corresponding to the offload target compiled for are added implicitly:

- `__NVCOMPILER_OPENMP_GPU` when OpenMP target directives are compiled for GPU.
- `__NVCOMPILER_OPENMP_MULTICORE` when OpenMP target directives are compiled for multicore CPU.

14.1. Environment Variables

The OpenMP specification includes many environment variables related to program execution.

Thread affinity

One important environment variable is `OMP_PROC_BIND`. It controls the OpenMP CPU thread affinity policy. When thread affinity is disabled, the operating system is free to move threads between the available CPU cores. When thread affinity is enabled, each thread is bound to a subset of the available CPU cores. The environment variable `OMP_PLACES` can be used to specify how a subset of the available CPU cores is determined for each thread. When set to a valid value, this environment variable will enable thread affinity and override the default thread affinity policy.

Binding threads to certain CPU cores is often beneficial for application performance, because that can improve the CPU cache hit rate and limit memory transactions between different NUMA nodes. Therefore, it is important to consider enabling thread affinity for your application.

The default value of `OMP_PROC_BIND` is `false`. Thus, thread affinity is disabled by default. This is a conservative setting that allows certain classes of applications (such as OpenMP + MPI) to create multiple processes without taking special care of the thread affinity policy to avoid binding threads in different processes to the same CPU cores.

The following table explains the simplest possible values of `OMP_PROC_BIND`. For the comprehensive explanation of `OMP_PROC_BIND` and `OMP_PLACES`, please refer to the OpenMP specification.

Value	Behavior
`OMP_PROC_BIND=false`	Thread affinity is disabled unless `OMP_PLACES` is set to a valid value. When thread affinity is disabled, the operating system is free to assign threads to any available CPU core at any time of the application execution. This is the default value.
`OMP_PROC_BIND=true`	Thread affinity is enabled. Unless `OMP_PLACES` is set, the implementation attempts to assign threads optimally to CPU cores to maximize the cache hit rate and minimize the number of memory transactions between NUMA nodes.

Delaying Thread Affinity

Some applications need to initialize non-OpenMP threads or third-party multithreaded libraries before entering OpenMP logic. In such cases, if thread affinity is enabled, the application may want to prevent the restricted main thread affinity from being inherited by non-OpenMP threads (which is the standard behavior in the Linux OS).

To support this use case, the NVIDIA HPC OpenMP Runtime provides the `NVCOMPILER_OMP_LAZY_PROC_BIND` environment variable. This variable controls whether OpenMP CPU thread affinity binding should be delayed until an OpenMP parallel region is created, or whether it should be set as early as possible. The table below explains the behavior for each setting:

Value	Behavior
NVCOMPILER_OMP_LAZY_PROC_BIND=false	The main thread affinity is set during the initialization of the NVIDIA HPC OpenMP Runtime. This occurs at an unspecified point in the application's lifecycle, before any OpenMP directive or most OpenMP API functions are executed. This is the default setting.
NVCOMPILER_OMP_LAZY_PROC_BIND=true	The OpenMP CPU thread affinity binding is delayed until the first parallel region is created.

Note: When thread affinity is disabled, the NVCOMPILER_OMP_LAZY_PROC_BIND environment variable has no effect.

Number of threads

By default, the NVIDIA HPC OpenMP Runtime limits the number of threads that can be created for a parallel region to 4 times the number of logical CPU processors.

If an application attempts to exceed this limit, a warning message is emitted to the standard error output. You can suppress this warning by setting the NVCOMPILER_OMP_DISABLE_WARNINGS environment variable to the `true` value.

To override the default limit on the maximum number of threads, use the NVCOMPILER_CPU_HARD_THREAD_LIMIT environment variable:

Value	Behavior
NVCOMPILER_CPU_HARD_THREAD_LIMIT=<positive_integer>	Maximum number of threads in parallel regions

Handling Stack Overflows

OpenMP threads created by the NVIDIA HPC OpenMP Runtime have a default stack size determined by the implementation and the operating system. This can be adjusted by setting the OMP_STACKSIZE environment variable. When the stack size is too small, the application may crash due to a stack overflow.

To help diagnose this situation, the NVIDIA HPC OpenMP Runtime attempts to recognize stack overflows and can provide a meaningful error message along with a backtrace when a stack overflow occurs. You can control this behavior by setting the NVCOMPILER_OMP_STACK_GUARD environment variable. The table below explains the behavior for each setting:

Value	Behavior
NVCOMPILER_OMP_STACK_GUARD=true	The runtime will handle stack overflows by showing a meaningful error message along with a backtrace. This is the default setting.
NVCOMPILER_OMP_STACK_GUARD=false	The runtime will not handle stack overflows. Applications may crash with a memory access violation error such as segmentation fault.

Handling stack overflows is only supported for OpenMP threads created by the NVIDIA HPC OpenMP Runtime.

The implementation of the stack overflow handling relies on the use of system calls such as `sigaction` and `sigaltstack` to set up a signal handler for stack overflows. This may cause issues with third-party libraries that rely on the default signal handler behavior. To work around this, you can set the `NVCOMPILER_OMP_STACK_GUARD` environment variable to `false` to disable stack overflow handling.

Device offload

Another important environment variable to understand is `OMP_TARGET_OFFLOAD`. Use this environment variable to affect the behavior of execution on host and device including host fallback. The following table explains the behavior determined by each of the values to which you can set this environment variable.

Value	Behavior
`OMP_TARGET_OFFLOAD=DEFAULT`	Try to execute on a GPU; if a supported GPU is not available, fallback to the host
`OMP_TARGET_OFFLOAD=DISABLED`	Do not execute on the GPU even if one is available; execute on the host
`OMP_TARGET_OFFLOAD=MANDATORY`	Execute on a GPU or terminate the program

Number of teams on device

When an application offloads an `omp target teams` construct to the GPU, the number of teams is calculated automatically unless the construct has a `num_teams` clause. The automatic setting of the number of teams can be limited to a maximum value provided by the `OMP_NUM_TEAMS` environment variable. The same maximum value can also be set by the application at run time with the function `omp_set_num_teams`.

Value	Behavior
`OMP_NUM_TEAMS=<positive_integer>`	Maximum number of teams on device

For the comprehensive explanation of `OMP_NUM_TEAMS`, please refer to the OpenMP specification.

Number of threads in teams

An `omp target teams` construct offloaded to the GPU creates a league of teams each consisting of a certain number of threads. The number of threads is the same for all teams in the league, and is calculated automatically unless the construct has a `thread_limit` clause.

The environment variable `OMP_TEAMS_THREAD_LIMIT` can be used to limit the maximum number of threads in teams. The same maximum value can be set by the application with the runtime function `omp_set_teams_thread_limit`.

For NVIDIA GPUs, we recommend using values that are multiples of 32 (which is the size of the GPU thread warp). That equally applies to the `OMP_TEAMS_THREAD_LIMIT` environment variable, the `omp_set_teams_thread_limit` function and the `thread_limit` clause. For any other value, the actual limit on the number of threads per team will likely be rounded down to the nearest multiple of 32. The same guidance applies to the `num_threads` clause as well.

Value	Behavior
`OMP_TEAMS_THREAD_LIMIT=<positive_integer>`	Maximum number of threads in teams

For the comprehensive explanation of `OMP_TEAMS_THREAD_LIMIT`, please refer to the OpenMP specification.

Forcing the number of device teams and threads

In certain situations, for instance for debugging or performance tuning, it may be desirable to specify an exact number of teams and threads on the GPU. While OpenMP offers a number of convenient ways to control that, e.g. the `num_teams` and `thread_limit` clauses, as well as the environment variables described above, they do not guarantee an exact teams and threads configuration.

The NVIDIA HPC OpenMP Runtime supports the `NVCOMPILER_OMP_CUDA_GRID` environment variable. When set, it requests the runtime to use the exact number of teams and threads per team when running OpenMP compute constructs on the GPU. Essentially, its effect is to use a specific CUDA grid configuration for any kernel, bypassing runtime and compiler guidance.

Value	Behavior
`NVCOMPILER_OMP_CUDA_GRID=<num_blocks>,<num_threads>`	The `<num_blocks>` and `<num_threads>` must be positive integers. They are used to form a CUDA grid when running GPU kernels associated with `omp target` compute constructs.

However, even with an exact CUDA grid specified, the runtime may still use a corrected configuration if that is necessary for a successful kernel launch.

Please refer to the CUDA C++ Programming Guide for the detailed explanation of how the CUDA kernel execution configurations work.

Runtime warnings

The NVIDIA HPC OpenMP Runtime may issue warnings when it encounters behavior that is non-standard or not explicitly defined by the OpenMP specification. These warnings are printed to the application's standard error output. Warnings do not interrupt execution; the application continues running normally.

To suppress all runtime warning messages, set the `NVCOMPILER_OMP_DISABLE_WARNINGS` environment variable to the `true` value.

Value	Behavior
`NVCOMPILER_OMP_DISABLE_WARNINGS=false`	The runtime can generate warning messages to the standard error output. This is the default setting.
`NVCOMPILER_OMP_DISABLE_WARNINGS=true`	No warning messages will be generated.

14.2. Fallback Mode

The HPC compilers support host fallback of OpenMP `target` regions when no GPU is present or `OMP_TARGET_OFFLOAD` is set to `DISABLED`. Execution should always be correct but the performance of the target region may not always be optimal when run on the host. OpenMP target regions prescriptively structured for optimal execution on GPUs may not perform well when run on the dissimilar architecture of the CPU. To provide performance portability between host and device, we recommend use of the `loop` construct.

firstprivates with nowait not supported for host execution

There is currently a limitation on the use of the `nowait` clause on target regions intended for execution on the host (-mp or -mp=gpu with `OMP_TARGET_OFFLOAD=DISABLED`). If the target region references variables having the `firstprivate` data-sharing attribute, their concurrent updates are not guaranteed to be safe. To work around this limitation, when running on the host, we recommend avoiding the `nowait` clause on such target regions or equivalently using the `taskwait` construct immediately following the region.

14.3. Loop

The HPC compilers support the `loop` construct with an extension to the default binding thread set mechanism specified by OpenMP in order to allow the compilers the freedom to analyze loops and dependencies to generate highly parallel code for CPU and GPU targets. In other words, the compilers map `loop` to either teams or to threads, as the compiler chooses, unless the user explicitly specifies otherwise. The mapping selected is specific to each target architecture even within the same executable (i.e., GPU offload and host fallback) thereby facilitating performance portability.

The shape of the parallelism offered by NVIDIA's GPUs, consisting of thread blocks and three dimensions of threads therein, differs from the multi-threaded vector parallelism of modern CPUs. The following table summarizes the OpenMP mapping to NVIDIA GPUs and multicore CPUs:

Construct	CPU	GPU
`!$omp target`		starts offload
`!$omp teams`	single team	CUDA thread blocks in grid
`!$omp parallel`	CPU threads	CUDA threads within thread block
`!$omp simd`	hint for vector instructions	simdlen(1)

HPC programs need to leverage all available parallelism to achieve performance. The programmer can attempt to become an expert in the intricacies of each target architecture and use that knowledge to structure programs accordingly. This prescriptive model can be successful but tends to increase source code complexity and often requires restructuring for each new target architecture. Here's an example where a programmer explicitly requests the steps the compiler should take to map parallelism to two targets:

```
#ifdef TARGET_GPU
   #pragma omp target teams distribute reduction(max:error)
#else
   #pragma omp parallel for reduction(max:error)
```

(continues on next page)

(continued from previous page)

```
#endif
for( int j = 1; j < n-1; j++) {
#ifdef TARGET_GPU
   #pragma omp parallel for reduction(max:error)
#endif
  for( int i = 1; i < m-1; i++ ) {
      Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                           + A[j-1][i] + A[j+1][i]);
      error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
  }
}
```

In this example, the GPU target has nested parallelism in the `target teams distribute` construct. In such cases, the compiler may need help from the programmer to allocate the optimal number of threads for the nested parallel region. We recommend considering the use of the `thread_limit` clause on the entire `target` construct that has nested parallelism.

An alternative is for the programmer to focus on exposing parallelism in a program and allowing a compiler to do the mapping onto the target architectures. The HPC compilers' implementation of `loop` supports this descriptive model. In this example, the programmer specifies the loop regions to be parallelized by the compiler and the compilers parallelize `loop` across teams and threads:

```
#pragma omp target teams loop reduction(max:error)
for( int j = 1; j < n-1; j++) {
  #pragma omp loop reduction(max:error)
  for( int i = 1; i < m-1; i++ ) {
      Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                           + A[j-1][i] + A[j+1][i]);
      error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
  }
}
```

The programmer's tuning tool with `loop` is the `bind` clause. The following table extends the previous mapping example:

Construct	CPU	GPU
`!$omp loop bind(teams)`	threads	CUDA thread blocks and threads
`!$omp loop bind(parallel)`	threads	CUDA threads
`!$omp loop bind(thread)`	single thread (useful for vector instructions)	single thread

Orphaned `loop` constructs within a single file are supported; a binding region of either `parallel` or `thread` must be specified with such loops via the `bind` clause. The compilers support `loop` regions containing procedure calls as long as the callee does not contain OpenMP directives.

Here are a few additional examples using `loop`. We also show examples of the type of information the compiler would provide when using the `-Minfo` compiler option.

Use of `loop` in Fortran:

```
!$omp target teams loop
do n1loc_blk = 1, n1loc_blksize
  do igp = 1, ngpown
    do ig_blk = 1, ig_blksize
      do ig = ig_blk, ncouls, ig_blksize
        do n1_loc = n1loc_blk, ntband_dist, n1loc_blksize
          !expensive computation codes
        enddo
      enddo
    enddo
  enddo
enddo

$ nvfortran test.f90 -mp=gpu -Minfo=mp
42, !$omp target teams loop
  42, Generating "nvkernel_MAIN__F1L42_1" GPU kernel
      Generating Tesla code
    43, Loop parallelized across teams ! blockidx%x
    44, Loop run sequentially
    45, Loop run sequentially
    46, Loop run sequentially
    47, Loop parallelized across threads(128) ! threadidx%x
  42, Generating Multicore code
    43, Loop parallelized across threads
```

Use of `loop`, `collapse`, and `bind`:

```
!$omp target teams loop collapse(3)
do n1loc_blk = 1, n1loc_blksize
  do igp = 1, ngpown
    do ig_blk = 1, ig_blksize
      !$omp loop bind(parallel) collapse(2)
      do ig = ig_blk, ncouls, ig_blksize
        do n1_loc = n1loc_blk, ntband_dist, n1loc_blksize
          !expensive computation codes
        enddo
      enddo
    enddo
  enddo
enddo

$ nvfortran test.f90 -mp=gpu -Minfo=mp

42, !$omp target teams loop
  42, Generating "nvkernel_MAIN__F1L42_1" GPU kernel
      Generating Tesla code
    43, Loop parallelized across teams collapse(3) ! blockidx%x
    44,   ! blockidx%x collapsed
    45,   ! blockidx%x collapsed
    47, Loop parallelized across threads(128) collapse(2) ! threadidx%x
    48,   ! threadidx%x collapsed
  42, Generating Multicore code
    43, Loop parallelized across threads
```

Use of `loop`, `collapse`, and `bind(thread)`:

```
!$omp target teams loop collapse(3)
```

(continues on next page)

(continued from previous page)

```
do n1loc_blk = 1, n1loc_blksize
  do igp = 1, ngpown
    do ig_blk = 1, ig_blksize
      !$omp loop bind(thread) collapse(2)
      do ig = ig_blk, ncouls, ig_blksize
        do n1_loc = n1loc_blk, ntband_dist, n1loc_blksize
          ! expensive computation codes
        enddo
      enddo
    enddo
  enddo
enddo

$ nvfortran test.f90 -mp=gpu -Minfo=mp

42, !$omp target teams loop
  42, Generating "nvkernel_MAIN__F1L42_1" GPU kernel
      Generating Tesla code
    43, Loop parallelized across teams, threads(128) collapse(3) ! blockidx%x
↪threadidx%x
    44,   ! blockidx%x threadidx%x collapsed
    45,   ! blockidx%x threadidx%x collapsed
    47, Loop run sequentially
    48,   collapsed
  42, Generating Multicore code
    43, Loop parallelized across threads
```

14.4. OpenMP Subset

This section contains the subset of OpenMP 5.0 features that the HPC compilers support. We have attempted to define this subset of features to be those that enable, where possible, OpenMP-for-GPU application performance that closely mirrors the success NVIDIA has seen with OpenACC. Almost every feature supported on NVIDIA GPUs is also supported on multicore CPUs, although the reverse is not true. Most constructs from OpenMP 3.1 and OpenMP 4.5 that apply to multicore CPUs are supported for CPU targets, and some features from OpenMP 5.0 are supported as well.

OpenMP target offload to NVIDIA GPUs is supported on NVIDIA V100 or later GPUs.

The section numbers below correspond to the section numbers in the OpenMP Application Programming Interface Version 5.0 November 2018 document.

2. Directives

2.3 Variant Directives

2.3.4 Metadirectives

The `target_device`/`device` context selector is supported with the `kind`(`host`|`nohost`|`cpu`|`gpu`) and `arch`(`nvptx`|`nvptx64`) trait selectors. The `arch` trait property `nvptx` is an alias for `nvptx64`; any other `arch` trait properties are treated as not matching or are ignored. The `isa` selector is treated as not matching or is ignored; no support is provided to select a context based on NVIDIA GPU compute capability.

The `implementation` context selector is supported with the `vendor(nvidia)` trait selector.

The `user` context selector is supported with the `condition(expression)` trait selector including dynamic `user` traits.

The syntax `begin/end metadirective` is not supported.

2.3.5 Declare Variant Directive

The `device` context selector is supported with the `kind(host|nohost|cpu|gpu)` and `arch(nvptx|nvptx64)` trait selectors. The `arch` trait property `nvptx` is an alias for `nvptx64`; any other `arch` trait properties are treated as not matching or are ignored. The `isa` selector is also treated as not matching or is ignored; no support is provided to select a context based on NVIDIA GPU compute capability.

The `implementation` context selector is supported with the `vendor(nvidia)` trait selector; all other implementation trait selectors are treated as not matching.

The syntax `begin/end declare variant` is supported for C/C++.

2.4 Requires Directive

The `requires` directive has limited support. The requirement clauses `unified_address` and `unified_shared_memory` are accepted but have no effect. To activate OpenMP unified shared memory programming a command-line option needs to be passed in (refer to *OpenMP with CUDA Unified Memory* for more details).

2.5 Internal Control Variables

ICV support is as follows.

- `dyn-var`, `nthread-var`, `thread-limit-var`, `max-active-levels-var`, `active-levels-var`, `levels-var`, `run-sched-var`, `dyn-sched-var`, and `stacksize-var` are supported
- `place-partition-var`, `bind-var`, `wait-policy-var`, `display-affinity-var`, `default-device-var`, and `target-offload-var` are supported only on the CPU
- `affinity-format-var` is supported only on the CPU; its value is immutable
- `max-task-priority-var`, `def-allocator-var` are not supported
- `cancel-var` is not supported; it always returns false

2.6 Parallel Construct

Support for `parallel` construct clauses is as follows.

- The `num_threads`, `default`, `private`, `firstprivate`, and `shared` clauses are supported
- The `reduction` clause is supported as described in 2.19.5
- The `if` and `copyin` clauses are supported only for CPU targets; the compiler emits an error for GPU targets
- The `proc_bind` clause is supported only for CPU targets; it is ignored for GPU targets
- The `allocate` clause is ignored

2.7 Teams Construct

The `teams` construct is supported only when nested within a `target` construct that contains no statements, declarations, or directives outside the `teams` construct, or as a combined `targetteams` construct. The `teams` construct is supported for GPU targets. If the `target` construct falls back to CPU mode, the number of teams is one. Support for `teams` construct clauses is as follows.

- The `num_teams`, `thread_limit`, `default`, `private`, and `firstprivate` clauses are supported

- The `reduction` clause is supported as described in 2.19.5
- The `shared` clause is supported for CPU targets and is supported for GPU targets in unified-memory mode
- The `allocate` clause is ignored

2.8 Worksharing Constructs

2.8.1 Sections Construct

The `sections` construct is supported only for CPU targets; the compiler emits an error for GPU targets. Support for `sections` construct clauses is as follows.

- The `private` and `firstprivate` clauses are supported
- The `reduction` clause is supported as described in 2.19.5
- The `lastprivate` clause is supported; the optional `lastprivate` modifier is not supported
- The `allocate` clause is ignored

2.8.2 Single Construct

Support for `single` construct clauses is as follows.

- The `private`, `firstprivate`, and `nowait` clauses are supported
- The `copyprivate` clause is supported only for CPU targets; the compiler emits an error for GPU targets
- The `allocate` clause is ignored

2.8.3 Workshare Construct

The `workshare` construct is supported in Fortran only for CPU targets; the compiler emits an error for GPU targets.

2.9 Loop-Related Constructs

2.9.2 Worksharing-Loop Construct (for/do)

Support for worksharing `for` and `do` construct clauses is as follows.

- The `private`, `firstprivate`, and `collapse` clauses are supported
- The `reduction` clause is supported as described in 2.19.5
- The `schedule` clause is supported; the optional modifiers are not supported
- The `lastprivate` clause is supported; the optional `lastprivate` modifier is not supported
- The `ordered` clause is supported only for CPU targets; `ordered(n)` clause is not supported
- The `linear` clause is not supported
- The `order(concurrent)` clause is ignored
- The `allocate` clause is ignored

2.9.3 SIMD Directives

The `simd` construct can be used to provide tuning hints for CPU targets; the `simd` construct is ignored for GPU targets. Support for `simd` construct clauses is as follows.

- The `reduction` clause is supported as described in 2.19.5
- The `lastprivate` clause is supported; the optional `lastprivate` modifier is not supported
- The `if`, `simdlen`, and `linear` clauses are not supported

- The `safelen`, `aligned`, `nontemporal`, and `order(concurrent)` clauses are ignored

The composite `forsimd` and `dosimd` constructs are supported for CPU targets; they are treated as `for` and `do` directives for GPU targets. Supported `simd` clauses are supported on the composite constructs for the CPU. Any `simd` clauses are ignored for GPU targets.

The `declaresimd` directive is ignored.

2.9.4 Distribute Directives

The `distribute` construct is supported within a `teams` construct. Support for `distribute` construct clauses is as follows:

- The `private`, `firstprivate`, `collapse`, and `dist_schedule(static [ ,chunksize])` clauses are supported
- The `lastprivate` clause is not supported
- The `allocate` clause is ignored

The `distribute simd` construct is treated as a `distribute` construct and is supported for GPU targets; valid supported `distribute` clauses are accepted; `simd` clauses are ignored. The `distributesimd` construct is not supported for CPU targets.

The `distributeparallelfor` or `distributeparalleldo` constructs are supported for GPU targets. Valid supported `distribute` and `parallel` and `for` or `do` clauses are accepted. The `distributeparallelfor` or `distributeparalleldo` constructs are not supported for CPU targets.

The `distribute parallel `for simd` or `distribute parallel do simd` constructs are treated as `distribute parallel for` or `distribute parallel do` constructs and are supported for GPU targets. These are not supported for CPU targets.

2.9.5 Loop Construct

Support for `loop` construct clauses is as follows.

- The `private`, `bind`, and `collapse` clauses are supported
- The `reduction` clause is supported as described in 2.19.5
- The `order(concurrent)` clause is assumed
- The `lastprivate` clause is not supported

2.10 Tasking Constructs

2.10.1 Task Construct

The `task` construct is supported for CPU targets. The compiler emits an error when it encounters `task` within a `target` construct. Support for `task` construct clauses is as follows:

- The `if`, `final`, `default` , `private`, `firstprivate`, and `shared` clauses are supported
- The `depend([dependmodifier,] dependtype : list)` clause is supported as described in 2.17.11

2.10.4 Taskyield Construct

The `taskyield` construct is supported for CPU targets; it is ignored for GPU targets.

2.11 Memory Management Directives

The memory management allocators, memory management API routines, and memory management directives are not supported

2.12 Device Directives

2.12.1 Device Initialization

Depending on how the program is compiled and linked, device initialization may occur at the first `target` construct or API routine call, or may occur implicitly at program startup.

2.12.2 Target Data Construct

The `target data` construct is supported for GPU targets. Support for `target data` construct clauses is as follows.

- The `if`, `device`, `use_device_ptr`, and `use_device_addr` clauses are supported
- The `map` clause is supported as described in 2.19.7

2.12.3 Target Enter Data Construct

The `target enter data` construct is supported for GPU targets. Support for `enter data` construct clauses is as follows.

- The `if`, `device`, and `nowait` clauses are supported
- The `map` clause is supported as described in 2.19.7.
- The `depend([dependmodifier,] dependtype : list)` clause is supported as described in 2.17.11

2.12.4 Target Exit Data Construct

The `target exit data` construct is supported for GPU targets. Support for `exit data` construct clauses is as follows.

- The `if`, `device`, and `nowait` clauses are supported
- The `map` clause is supported as described in 2.19.7.
- The `depend([dependmodifier,] dependtype : list)` clause is supported as described in 2.17.11

2.12.5 Target Construct

The `target` construct is supported for GPU targets. If there is no GPU or GPU offload is otherwise disabled, execution falls back to CPU mode. Support for `target` construct clauses is as follows:

- The `if`, `private`, `firstprivate`, `is_device_ptr`, and `nowait` clauses are supported
- The `device` clause is supported without the device-modifier `ancestor` keyword
- The `map` clause is supported as described in 2.19.7
- The `defaultmap` clause is supported using OpenMP 5.0 semantics
- The `depend([dependmodifier,] dependtype : list)` clause is supported as described in 2.17.11
- The `allocate` and `uses_allocate` clauses are ignored

2.12.6 Target Update Construct

The `target update` construct is supported for GPU targets. Support for `target update` construct clauses is as follows.

- The `if`, `device`, and `nowait` clauses are supported.
- The `to` and `from` clauses are supported without `mapper` or `mapid`
- The `depend([dependmodifier,] dependtype : list)` clause is supported as described in 2.17.11

Array sections are supported in `to` and `from` clauses, including noncontiguous array sections. Array section strides are not supported. If the array section is noncontiguous, the OpenMP runtime may have to use multiple host-to-device or device-to-host data transfer operations, which increases the overhead. If the host data is in host-pinned memory, then `update` data transfers with the `nowait` clause are asynchronous. This means the data transfer for a `target update to nowait` may not occur immediately or synchronously with the program thread, and any changes to the data may affect the transfer, until a synchronizing operation is reached. Similarly, a `target update from nowait` may not occur immediately or synchronously with the program thread, and the downloaded data may not be available until a synchronizing operation is reached. If the host data is not in host-pinned memory, then `update` data transfers with the `nowait` clause may require that the data transfer operation use an intermediate pinned buffer managed by the OpenMP runtime library, and that a memory copy operation on the host between the program memory and the pinned buffer may be needed before starting or before finishing the transfer operation, which affects overhead and performance. To learn more about the pinned buffer, please refer to *Staging Memory Buffer <acc-mem-pinned-buffer>*.

2.12.7 Declare Target Construct

The `declare target` construct is supported for GPU targets.

- `declare target ... end declare target` is supported
- `declare target(list)` is supported
- The `to(list)` clause is supported
- The `device_type` clause is supported for C/C++

A function or procedure that is referenced in a function or procedure that appears in a `declare target to` clause (explicitly or implicitly) is treated as if its name had implicitly appeared in a `declare target to` clause.

2.13 Combined Constructs

Combined constructs are supported to the extent that the component constructs are themselves supported.

2.14 Clauses on Combined and Composite Constructs

Clauses on combined constructs are supported to the extent that the clauses are supported on the component constructs.

2.16 Master Construct

The `master` construct is supported for CPU and GPU targets.

2.17 Synchronization Constructs and Clauses

2.17.1 Critical Construct

The `critical` construct is supported only for CPU targets; the compiler emits an error for GPU targets.

2.17.2 Barrier Construct

The `barrier` construct is supported.

2.17.3 Implicit Barriers

Implicit barriers are implemented.

2.17.4 Implementation-Specific Barriers

There may be implementation-specific barriers, and they may be different for CPU targets than for GPU targets.

2.17.5 Taskwait Construct

The `taskwait` construct is supported only for CPU targets; it is ignored for GPU targets.

- The `depend([dependmodifier,] dependtype : list)` clause is supported as described in 2.17.11

2.17.6 Taskgroup Construct

The `taskgroup` construct is supported only for CPU targets. It is ignored for GPU targets.

2.17.7 Atomic Construct

Support for `atomic` construct clauses is as follows.

- The `read`, `write`, `update`, and `capture` clauses are supported.
- The memory order clauses `seq_cst`, `acq_rel`, `release`, `acquire`, `relaxed` are not supported
- The `hint` clause is ignored

2.17.8 Flush Construct

The `flush` construct is supported only for CPU targets.

2.17.9 Ordered Construct and Ordered Directive

The `ordered` block construct is supported only for CPU targets.

2.17.11 Depend Clause

The `depend` clause is supported on CPU targets. It is not supported on GPU targets. The dependence types `in`, `out`, and `inout` are supported. The dependence types `mutexinoutset` and `depobj`, dependence modifier `iterator(iters)`, `depend(source)`, and `depend(sink:vector)` are not supported.

2.19 Data Environment

2.19.2 Threadprivate Directive

The `threadprivate` directive is supported only for CPU targets. It is not supported for GPU targets; references to `threadprivate` variables in device code are not supported.

2.19.5 Reduction Clauses and Directives

The `reduction` clause is supported. The optional modifier is not supported.

2.19.6 Data Copying Clauses

The data copying `copyin` and `copyprivate` clauses are supported only for CPU targets; the compiler emits a compile-time error for GPU targets.

2.19.7 Data Mapping Attribute Rules, Clauses, and Directives

- The `map([[mapmod[,]...] maptype:] datalist)` clause is supported. Of the map-type-modifiers, `always` is supported, `close` is ignored, and `mapper(mapid)` is not supported.
- The `defaultmap` clause is supported using OpenMP 5.0 semantics.

2.20 Nesting of Regions

For constructs supported in this subset, restrictions on nesting of regions is observed. Additionally, nested parallel regions on CPU are not supported and nested teams or parallel regions in a target region are not supported.

Runtime Library Routines

3.2 Execution Environment Routines

The following execution environment runtime API routines are supported.

- `omp_set_num_threads`, `omp_get_num_threads`, `omp_get_max_threads`, `omp_get_thread_num`, `omp_get_thread_limit`, `omp_get_supported_active_levels`, `omp_set_max_active_levels`, `omp_get_max_active_levels`, `omp_get_level`, `omp_get_ancestor_thread_num`, `omp_get_team_size`, `omp_get_num_teams`, `omp_get_team_num`, `omp_is_initial_device`

The following execution environment runtime API routines are supported only on the CPU.

- `omp_get_num_procs`, `omp_set_dynamic`, `omp_get_dynamic`, `omp_set_schedule`, `omp_get_schedule`, `omp_in_final`, `omp_get_proc_bind`, `omp_get_num_places`, `omp_get_affinity_format`, `omp_set_default_device`, `omp_get_default_device`, `omp_get_num_devices`, `omp_get_device_num`, `omp_get_initial_device`

The following execution environment runtime API routines have limited support.

- `omp_get_cancellation`, `omp_get_nested`; supported only on the CPU; the value returned is always false
- `omp_display_affinity`, `omp_capture_affinity`; supported only on the CPU; the format specifier is ignored
- `omp_set_nested`; supported only on the CPU, the value is ignored

The following execution environment runtime API routines are not supported.

- `omp_get_place_num_procs`, `omp_get_place_proc_ids`, `omp_get_place_num`, `omp_get_partition_num_places`, `omp_get_partition_place_nums`, `omp_set_affinity_format`, `omp_get_max_task_priority`, `omp_pause_resource`, `omp_pause_resource_all`

3.3 Lock Routines

Lock runtime API routines are not supported on the GPU. The following lock runtime API routines are supported on the CPU.

- `omp_init_lock`, `omp_init_nest_lock`, `omp_destroy_lock`, `omp_destroy_nest_lock`, `omp_set_lock`, `omp_set_nest_lock`, `omp_unset_lock`, `omp_unset_nest_lock`, `omp_test_lock`, `omp_test_nest_lock`

The following lock runtime API routines are not supported.

- `omp_init_lock_with_hint`, `omp_init_nest_lock_with_hint`

3.4 Timing Routines

The following timing runtime API routines are supported.

- `omp_get_wtime`, `omp_get_wtick`

3.6 Device Memory Routines

The following device memory routines are supported only on the CPU.

- `omp_target_is_present`, `omp_target_associate_ptr`, `omp_target_disassociate_ptr`
- `omp_target_memcpy` and `omp_target_memcpy_rect` are only supported when copying to and from the same device.

The following device memory routines are supported on the CPU; we extend OpenMP to support these in target regions on a GPU, but only allocation and deallocation on the same device is supported.

- `omp_target_alloc`, `omp_target_free`

3.7 Memory Management Routines

The following memory management routines are supported.

- `omp_alloc`, `omp_free`

The following memory management routines are not supported.

- `omp_init_allocator`, `omp_destroy_allocator`, `omp_set_default_allocator`, `omp_get_default_allocator`

6 Environment Variables

The following environment variables have limited support.

- `OMP_SCHEDULE`, `OMP_NUM_THREADS`, `OMP_NUM_TEAMS`, `OMP_DYNAMIC`, `OMP_PROC_BIND`, `OMP_PLACES`, `OMP_STACKSIZE`, `OMP_WAIT_POLICY`, `OMP_MAX_ACTIVE_LEVELS`, `OMP_NESTED`, `OMP_THREAD_LIMIT`, `OMP_TEAMS_THREAD_LIMIT`, `OMP_DISPLAY_ENV`, `OMP_DISPLAY_AFFINITY`, `OMP_DEFAULT_DEVICE`, and `OMP_TARGET_OFFLOAD` are supported on CPU.
- `OMP_CANCELLATION` and `OMP_MAX_TASK_PRIORITY` are ignored.
- `OMP_AFFINITY_FORMAT`, `OMP_TOOL`, `OMP_TOOL_LIBRARIES`, `OMP_DEBUG`, and `OMP_ALLOCATOR` are not supported

14.5. Using metadirective

This section contains limitations affecting `metadirective` along with a few guidelines for its use.

The Fortran compiler does not support variants leading to an OpenMP directive for which a corresponding `end` directive is required.

Nesting `user` conditions, while legal, may create situations that the HPC Compilers do not handle gracefully. To avoid potential problems, use `device` traits inside `user` conditions instead. The following example illustrates this best practice.

Avoid nesting dynamic `user` conditions like this:

```
#pragma omp metadirective \
  when( user={condition(use_offload)} : target teams distribute) \
  default( parallel for schedule(static) )
  for (i = 0; i < N; i++) {
    ...
#pragma omp metadirective \
  when( user={condition(use_offload)} : parallel for)
    for (j = 0; j < N; j++) {
      ...
    }
    ...
  }
```

Instead, use `target_device` and `device` traits within dynamic `user` conditions like this:

```
#pragma omp metadirective \
  when( target_device={kind(gpu)}, user={condition(use_offload)} : target teams
↪distribute) \
  default( parallel for schedule(static) )
```

(continues on next page)

(continued from previous page)

```
  for (i = 0; i < N; i++) {
    ...
#pragma omp metadirective \
  when( device={kind(gpu)} : parallel for)
    for (j = 0; j < N; j++) {
      ...
    }
    ...
  }
```

The HPC compilers do not support nesting `metadirective` inside a `target` construct applying to a syntactic block leading to a `teams` variant. Some examples:

The compilers will emit an error given the following code:

```
#pragma omp target map(to:v1,v2) map(from:v3)
{
#pragma omp metadirective \
when( device={arch("nvptx")} : teams distribute parallel for) \
default( parallel for)
  for (int i = 0; i < N; i++) {
    v3[i] = v1[i] * v2[i];
  }
}
```

The compilers will always match `device={arch("nvptx")}` given the following code:

```
#pragma omp target map(to:v1,v2) map(from:v3)
#pragma omp metadirective \
when( device={arch("nvptx")} : teams distribute parallel for) \
default( parallel for)
  for (int i = 0; i < N; i++) {
    v3[i] = v1[i] * v2[i];
  }
```

The compilers match `device={"arch"}` for GPU code, and `default` for host fallback, given the following code:

```
#pragma omp target teams distribute map(to:v1,v2) map(from:v3)
for (...)
{
#pragma omp metadirective \
when( device={arch("nvptx")} : parallel for) \
default( simd )
  for (int i = 0; i < N; i++) {
    v3[i] = v1[i] * v2[i];
  }
}
```

14.6. Mapping target constructs to CUDA streams

An OpenMP target task generating construct is executed on the GPU in a CUDA stream. The following are target task generating constructs:

- `target enter data`
- `target exit data`
- `target update`
- `target`

This section explains how these target constructs are mapped to CUDA streams. The relationship with the OpenACC queues is also explained below.

Keep in mind that the `target data` construct does not generate a task and is not necessarily executed in a CUDA stream. It also cannot have the `depend` and `nowait` clauses, thus its behavior cannot be directly controlled by the user application. The rest of this section does not cover the behavior of the `target data` construct.

Any task-generating target construct can have `depend` and `nowait` clauses. The NVIDIA OpenMP Runtime takes these clauses as a guidance for how to map the construct to a specific CUDA stream. Below is a breakdown of how the clauses affect the mapping decisions.

'target' without 'depend', without 'nowait'

For these constructs, the per-thread default CUDA stream is normally used. The stream is unique for each host thread, so target regions created by different host threads will execute independently in different streams according to the CUDA rules described in CUDA Runtime API; see the rules in the "Per-thread default stream" section.

The OpenACC queue `acc_async_sync` is initially associated with the same per-thread default CUDA stream. The user is allowed to change the association by calling `acc_set_cuda_stream(acc_async_sync, stream)`. This will change accordingly the stream used for `target` without `nowait`.

The CUDA stream handle can be directly obtained via the `ompx_get_cuda_stream(int device, int nowait)` function, with the `nowait` parameter set to 0. The per-thread default stream can be obtained with the CUDA handle `CU_STREAM_PER_THREAD` or `cudaStreamPerThread`.

Here is an example of how a custom CUDA stream can be used to substitute the default stream:

```
extern __global__ void kernel(int *data);

  CUstream stream;
  cuStreamCreate(&stream, CU_STREAM_DEFAULT);
  acc_set_cuda_stream(acc_async_sync, stream);
#pragma omp target enter data map(to:data[:N])
#pragma omp target data use_device_ptr(data)
  kernel<<<N/32, 32, 0, stream>>>(data);
#pragma omp target teams distribute parallel for
  for (int i = 0; i < N; i++) {
    data[i]++;
  }
#pragma omp target exit data map(from:data[:N])
```

Note there is no explicit stream synchronization after the CUDA `kernel` is launched. The stream is synchronized automatically at the `target` constructs that follow.

'target' with 'depend', without 'nowait'

For this construct, the runtime will block the current thread until all dependencies listed in the `depend` clause are resolved. Then, the `target` construct will be executed in the default per-thread CUDA stream as described in the previous section (that is, as if there is no `depend` clause).

'target' with 'nowait', without 'depend'

By default, the runtime will select a CUDA stream for each new `target nowait` construct. The selected stream may be the same that was used for a prior `target nowait` construct. That is, there is no guarantee of uniqueness of the selected stream.

This is different from the OpenACC model that uses the same CUDA stream associated with the `acc_async_noval` queue for any asynchronous construct with the `async` clause without an argument. To change this behavior, the user can call the `ompx_set_cuda_stream_auto(int enable)` function with the `enable` parameter set to 0. In this case, the CUDA stream associated with the `acc_async_noval` OpenACC queue will be used for all OpenMP `target nowait` constructs. Another way to enable this behavior is to set the environment variable `NVCOMPILER_OMP_AUTO_STREAMS` to `FALSE`.

To access the stream used for the next `target nowait` construct, the user can call the `ompx_get_cuda_stream(int device, int nowait)` function, with the `nowait` parameter set to 1.

'target' with both 'depend' and 'nowait'

The decision on which CUDA stream to use in this case relies on previously scheduled target and host tasks sharing a subset of the dependencies listed in the `depend` clause:

- If the target construct has only one dependency, which is of the type `inout` or `out`, and that dependency maps to a previously scheduled `target depend(...) nowait` construct, and the same device is used for both target constructs, then the CUDA stream which the previous target task was scheduled to will be used.
- Otherwise, a CUDA stream will be selected for this target construct according to the stream selection policy.

Note that target constructs with a single `in` dependency can be scheduled on a newly selected CUDA stream. This is to allow parallel execution of multiple `target nowait` constructs that depend on data produced by another previously scheduled `target nowait` construct.

Here is a simplified example of how a `target` construct, a CUDA library function and a CUDA kernel can be executed on the GPU in the same stream asynchronously with respect to the host thread:

```
extern __global__ void kernel(int *data);

cudaStream_t stream = (cudaStream_t)ompx_get_cuda_stream(omp_get_default_device(), 1);
cufftSetStream(cufft_plan, stream);

#pragma omp target enter data map(to:data[:N]) depend(inout:stream) nowait
#pragma omp target data use_device_ptr(data)
  {
    kernel<<<N/32, 32, 0, stream>>>(data);
    cufftExecC2C(cufft_plan, data, data, CUFFT_FORWARD);
  }
#pragma omp target teams distribute parallel for depend(inout:stream) nowait
  for (int i = 0; i < N; i++) {
```

(continues on next page)

(continued from previous page)

```
    data[i]++;
  }
#pragma omp target exit data map(from:data[:N]) depend(inout:stream) nowait
```

Note that the `stream` variable holds the CUDA stream handle and also serves as the dependency for the `target` constructs. This dependency enforces the order of execution and also guarantees the target constructs are on the same stream that was returned from the `ompx_get_cuda_stream` function call.

NVIDIA OpenMP API to access and control CUDA streams

NVIDIA OpenMP Runtime provides the following API to access CUDA streams and to control their use.

```
void *ompx_get_cuda_stream(int device, int nowait);
```

This function returns the handle of the CUDA stream that will be used for the next `target` construct:

- If the `nowait` parameter is set to 0, it returns the CUDA stream associated with the OpenACC queue `acc_async_sync`, which is initially mapped to the default per-thread CUDA stream;
- Otherwise, it returns a CUDA stream which will be used for the next `target nowait` construct that cannot be mapped to an existing stream according to the rules for the `depend` clause.

```
void ompx_set_cuda_stream_auto(int enable);
```

This function sets the policy for how CUDA streams are selected for `target nowait` constructs:

- If the `enable` parameter is set to a non-zero value, an internally selected CUDA stream will be used for each `target nowait` construct that follows. This is the default behavior;
- Otherwise, the CUDA stream associated with the OpenACC queue `acc_async_noval` will be used for all `target nowait` constructs that follow. This becomes the default behavior if the environment variable `NVCOMPILER_OMP_AUTO_STREAMS` is set to `FALSE`.

The setting is done only for the host thread which calls this function.

14.7. Noncontiguous Array Sections

Array sections can be used in `to` and `from` clauses, including noncontiguous array sections. The noncontiguous array section must be specified in a single `map` clause; it cannot be split between multiple directives. Although this feature may become a part of a future OpenMP specification, at this time it is an NVIDIA HPC compilers extension.

14.8. OpenMP with CUDA Unified Memory

This section will focus on OpenMP unified shared memory programming, and assume users are familiar with Separate, Managed, and Unified Memory Modes explained in the *Memory Model* and *Managed and Unified Memory Modes* sections. OpenMP unified shared memory corresponds to Unified Memory Mode in NVHPC Compilers and it can be enabled with `-gpu=mem:unified` flag. Source code with `requires unified_shared_memory` directive is accepted but requires `-gpu=mem:unified` flag to activate Unified Memory Mode.

In Unified Memory Mode, `map` clauses on `target` constructs are optional. Additionally, `declare target` directives are optional for variables with static storage duration accessed inside functions to which such directive is applied. The OpenMP unified shared memory eases accelerator programming on the GPUs removing the need for data management and only requiring to express the parallelism in the compute regions.

In Unified Memory Mode, all data is managed by the CUDA runtime. Explicit data `map` clauses which manage the data movement across the host and devices become optional. All variables are accessible from the OpenMP offload compute regions executing on the GPU. The `map` clause with `alloc`, `to`, `from`, and `tofrom` type will not result in any device allocation or data transfer. The OpenMP runtime, however, may leverage such clauses to communicate preferable data placement to the CUDA runtime by means of memory hint APIs as elaborated in the following blog post on the NVIDIA website: Simplifying GPU Application Development with Heterogeneous Memory Management. Device memory can be allocated or deallocated in OpenMP programs in Unified Memory Mode by using the `omp_target_alloc` and `omp_target_free` API calls. Please, note that the memory allocated through `omp_target_alloc` cannot be accessed by the host.

Understanding Data Movement

When the compiler encounters a compute construct without visible `target data` directives or `map` clauses, it attempts to determine what data is required for correct execution of the region on the GPU. When the compiler is unable to determine the size and shape of data needing to be accessible on the device, it behaves as follows:

- In Separate Memory Mode, the compiler may not be able to alert you to the need for an explicit data clause specifying size and/or shape of data being copied to/from the GPU. In this case, the default length of one may be used. This may cause illegal memory access errors at runtime on the GPU devices.
- In Managed Memory Mode (`-gpu=mem:managed`), the compiler assumes the data is allocated in managed memory and thus is accessible from the device; if this assumption is wrong, for example, if the data was defined globally or is located on the CPU stack, the program may fail at runtime.
- In Unified Memory Mode (`-gpu=mem:unified`), all data is accessible from the device making information about size and shape unnecessary.

Take the following example in C:

```
#pragma omp declare target
void set(int* ptr, int i, int j, int dim){
  int idx = i * dim + j;
  return ptr[idx] = someval(i, j);
}
#pragma omp end declare target

void fill2d(int* ptr, int dim){
#pragma omp target teams distribute parallel for
  for (int i = 0; i < dim; i++)
    for (int j = 0; j < dim; j++)
      set(ptr, i, j, dim);
}
```

In Separate Memory Mode, the only way to guarantee correctness for this example is to specify an array section in the `target` construct as follows:

```
#pragma omp target teams distribute parallel for map(from: ptr[0:dim*dim])
```

This change explicitly instructs the OpenMP implementation about the precise data segment used within the target for loop.

In Unified Memory Mode, the `map` clause is not required.

The next example, in Fortran, illustrates how a global variable can be accessed in an OpenMP routine without requiring any explicit annotation.

```
module m
integer :: globmin = 1234
contains
subroutine findmin(a)
!$omp declare target
  integer, intent(in)  :: a(:)
  integer :: i
  do i = 1, size(a)
    if (a(i) .lt. globmin) then
      globmin = a(i)
    endif
  end do
end subroutine
end module m
```

Compile the example above for Unified Memory Mode:

```
nvfortran -mp=gpu -gpu=mem:unified example.f90
```

The source does not need any OpenMP directives to access module variable `globmin`, to either read or update its value, in the routine invoked from CPU and GPU. Moreover, any access to `globmin` will be made to the same exact instance of the variable from CPU and GPU; its value is synchronized automatically. In Separate or Managed Memory Modes, such behavior can only be achieved with a combination of OpenMP `declare target` and `target update` directives in the source code.

Migrating existing OpenMP applications written for Separate Memory Mode should, in most cases, be a seamless process requiring no source changes. Some data access patterns, however, may lead to different results produced during application execution in Unified Memory Mode. Applications which rely on having separate data copies in GPU memory to conduct temporary computations on the GPU – without maintaining data synchronization with the CPU – pose a challenge for migration to unified memory. For the following Fortran example, the value of variable `c` after the last loop will differ depending on whether the example is compiled with or without `-gpu=mem:unified`.

```
b(:) = ...
c = 0

!$omp target data map(to: b) map(from: a)
!$omp target distribute teams parallel for
do i = 1, N
  b(i) = b(i) * i
end do
!$omp target distribute teams parallel for
do i = 1, N
  a(i) =  b(i) + i
end do
!$omp end target data

do i = 1, N
  c = c + a(i) + b(i)
end do
```

Without Unified Memory, array `b` is copied into the GPU memory at the beginning of the OpenMP `target data` region. It is then updated in the GPU memory and used to compute elements of array `a`. As instructed by the data clause `map(to:b)`, `b` is not copied back to the CPU memory at the end of the `target data` region and therefore its initial value is used in the computation of `c`. With `-mp=gpu -gpu=mem:unified`, the updated value of `b` in the first loop is automatically visible in the last loop leading to a different value of `c` at its end.

Additional complications may arise from the asynchronous execution as the use of unified shared memory may require extra synchronizations to avoid data races.

14.9. Multiple Device Support

A program can use multiple devices on a single node.

This functionality is supported using the `omp_set_default_device` API call and the `device()` clause on the `target` constructs. Our experience is that most programs use MPI parallelism with each MPI rank selecting a single GPU to which to offload. Some programs assign multiple MPI ranks to each GPU, in order to keep the GPU fully occupied, though the fixed memory size of the GPU limits how effective this strategy can be. Similarly, other programs use OpenMP thread parallelism on the CPU, with each thread selecting a single GPU to which to offload.

14.10. Interoperability with CUDA

The HPC Compilers support interoperability of OpenMP and CUDA to the same extent they support CUDA interoperability with OpenACC.

If OpenMP and CUDA code coexist in the same program, the OpenMP runtime and the CUDA runtime use the same CUDA context on each GPU. To enable this coexistence, use the compilation and linking option `-cuda`. CUDA-allocated data is available for use inside OpenMP target regions with the OpenMP analog `is_device_ptr` to OpenACC's `deviceptr()` clause.

OpenMP-allocated data is available for use inside CUDA kernels directly if the data was allocated with the `omp_target_alloc()` API call; if the OpenMP data was created with a `target data map` clause, it can be made available for use inside CUDA kernels using the `target data use_device_addr()` clause. Calling a CUDA device function inside an OpenMP target region is supported, as long as the CUDA function is a scalar function, that is, does not use CUDA shared memory or any inter-thread synchronization. Calling an OpenMP `declare target` function inside a CUDA kernel is supported as long as the `declare target` function has no OpenMP constructs or API calls.

14.11. Interoperability with Other OpenMP Compilers

OpenMP CPU-parallel object files compiled with NVIDIA's HPC compilers are interoperable with OpenMP CPU-parallel object files compiled by other compilers using the KMPC OpenMP runtime interface. Compilers supporting KMPC OpenMP include Intel and CLANG. The HPC compilers support

a GNU OpenMP interface layer as well which provides OpenMP CPU-parallel interoperability with the GNU compilers.

For OpenMP GPU computation, there is no similar formal or informal standard library interface for launching GPU compute constructs or managing GPU memory. There is also no standard way to manage the device context in such a way as to interoperate between multiple offload libraries. The HPC compilers therefore do not support interoperability of device compute offload operations and similar operations generated with another compiler.

14.12. GNU STL

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.

Chapter 15. Using Stdpar

This chapter describes the NVIDIA HPC Compiler support for standard language parallelism, also known as Stdpar:

- ISO C++ standard library parallel algorithms with `nvc++`
- ISO Fortran `do concurrent` loop construct with `nvfortran`

Use the `-stdpar` compiler option to enable parallel execution with standard parallelism. The sub-options to `-stdpar` are the following:

- `gpu`: compile for parallel execution on GPU; this sub-option is the default. This feature is supported on the NVIDIA Pascal architecture and newer.
- `multicore`: compile for multicore CPU execution.

By default, NVC++ auto-detects and generates GPU code for the type of GPU that is installed on the system on which the compiler is running. To generate code for a specific GPU architecture, which may be necessary when the application is compiled and run on different systems, add the `-gpu=ccXX` command-line option. More details can be found in *Compute Capability*.

Predefined Macros

The following macros corresponding to the parallel execution target compiled for are added implicitly:

- `__NVCOMPILER_STDPAR_GPU` for parallel execution on GPU.
- `__NVCOMPILER_STDPAR_MULTICORE` for parallel execution on multicore CPU.

15.1. GPU Memory Modes

When compiling for GPU execution, Stdpar utilizes *Managed and Unified Memory Modes* for managing data accessed from the sequential code running on CPU and from the parallel code on GPU.

The compiler detects the memory capability of the system on which the compiler is running and uses that information to enable the correct memory mode as follows:

- When compiled on the platform with full CUDA Unified Memory capability, `-stdpar` implies `-gpu=mem:unified`.
- When compiled on the platform with CUDA Managed Memory capability only, `-stdpar` implies `-gpu=mem:managed`.

To compile code for a specific Memory Mode regardless of the memory capability of the system on which you are compiling, add the desired `-gpu=mem:unified` or `-gpu=mem:managed` option.

Stdpar with Separate Memory Mode can only be supported when the data are fully managed through features of other programming models e.g. OpenACC.

All restrictions on variables used on the GPU in standard language parallel code in Managed Memory Mode have been removed when using Unified Memory Mode.

If the compiler utilises CUDA Managed Memory automatically, the interception of deallocations is enabled implicitly at runtime. This is to prevent deallocating the data with unmatching API which may lead to undefined behavior. The interception incurs some runtime overhead and may be unnecessary if allocatations and deallocations for all data in the application are performed using the matching APIs. The interception can be disabled using dedicated command-line options detailed in *Interception of Deallocations*. More details about the memory modes supported by the NVIDIA HPC Compilers and dedicated command-line options can be found in *Memory Model*.

15.2. Stdpar C++

The NVIDIA HPC C++ compiler, NVC++, supports C++ Standard Language Parallelism (Stdpar) for execution on NVIDIA GPUs and multicore CPUs. As mentioned previously, use the NVC++ command-line option `-stdpar` to enable GPU accelerated C++ Parallel Algorithms. The following sections go into more detail about the NVC++ support for the ISO C++ Standard Library Parallel Algorithms.

15.2.1. Introduction to Stdpar C++

The C++17 Standard introduced higher-level parallelism features that allow users to request parallelization of Standard Library algorithms.

This higher-level parallelism is expressed by adding an execution policy as the first parameter to any algorithm that supports execution policies. Most of the existing Standard C++ algorithms were enhanced to support execution policies. C++17 defined several new parallel algorithms, including the useful std::reduce and std::transform_reduce.

C++17 defines three execution policies:

- `std::execution::seq`: Sequential execution. No parallelism is allowed.
- `std::execution::par`: Parallel execution on one or more threads.
- `std::execution::par_unseq`: Parallel execution on one or more threads, with each thread possibly vectorized.

When you use an execution policy other than `std::execution::seq`, you are communicating two important things to the compiler:

- You prefer but do not require that the algorithm be run in parallel. A conforming C++17 implementation may ignore the hint and run the algorithm sequentially, but a performance-oriented implementation takes the hint and executes in parallel when possible and prudent.
- The algorithm is safe to run in parallel. For the `std::execution::par` and `std::execution::par_unseq` policies, any user-provided code—such as iterators, lambdas, or function objects passed into the algorithm—must not introduce data races if run

concurrently on separate threads. For the `std::execution::par_unseq` policy, any user-provided code must not introduce data races or deadlocks if multiple calls are interleaved on the same thread, which is what happens when a loop is vectorized. For more information about potential deadlocks, see the forward progress guarantees provided by the parallel policies or watch CppCon 2018: Bryce Adelstein Lelbach "The C++ Execution Model".

The C++ Standard grants compilers great freedom to choose if, when, and how to execute algorithms in parallel as long as the forward progress guarantees the user requests are honored. For example, `std::execution::par_unseq` may be implemented with vectorization and `std::execution::par` may be implemented with a CPU thread pool. It is also possible to execute parallel algorithms on a GPU, which is a good choice for invocations with sufficient parallelism to take advantage of the processing power and memory bandwidth of NVIDIA GPU processors.

15.2.2. NVC++ Compiler Parallel Algorithms Support

NVC++ supports C++ Standard Language Parallelism with the parallel execution policies `std::execution::par` or `std::execution::par_unseq` for execution on GPUs or multicore CPUs.

Lambdas, including generic lambdas, are fully supported in parallel algorithm invocations. No language extensions or non-standard libraries are required to enable GPU acceleration. All data movement between host memory and GPU device memory is performed implicitly and automatically under the control of *Managed and Unified Memory Modes*.

It's straightforward to automatically GPU accelerate C++ Parallel Algorithms with NVC++. However, there are some restrictions and limitations you need to be aware of as explained below.

15.2.2.1 Enabling Parallel Algorithms with the -stdpar Option

GPU acceleration of C++ Parallel Algorithms is enabled with the `-stdpar=gpu` command-line option to NVC++. If `-stdpar=gpu` is specified (or `-stdpar` without an argument), almost all algorithms that use a parallel execution policy are compiled for offloading to run in parallel on an NVIDIA GPU:

```
nvc++ -stdpar=gpu program.cpp -o program
```

```
nvc++ -stdpar program.cpp -o program
```

In addition, the GPU acceleration sub-option can be further specialized using `-stdpar=gpu:acc`. This option directs the compiler to use its OpenACC implementation to GPU-accelerate a subset of algorithm with a parallel execution policy:

```
nvc++ -stdpar=gpu:acc program.cpp -o program
```

More details about the OpenACC support of Stdpar C++ is provided in OpenACC Implementation of Parallel Algorithms.

Acceleration of C++ Parallel Algorithms with multicore CPUs is enabled with the `-stdpar=multicore` command-line option to NVC++. If `-stdpar=multicore` specified, almost all algorithms that use a parallel execution policy are compiled to run on a multicore CPU:

```
nvc++ -stdpar=multicore program.cpp -o program
```

When either `-stdpar=gpu,multicore` or `-stdpar=gpu:acc,multicore` command-line options are specified to NVC++, the parallel algorithms code is compiled for both GPU and multicore CPU. When

the execution platform has any GPU the binary executes on the GPU and otherwise on the multicore CPU.

```
nvc++ -stdpar=gpu,multicore program.cpp -o program
```

```
nvc++ -stdpar=gpu:acc,multicore program.cpp -o program
```

15.2.3. Stdpar C++ Simple Example

Here are a few simple examples to get a feel for how the C++ Parallel Algorithms work.

From the early days of C++, sorting items stored in an appropriate container has been relatively easy using a single call like the following:

```
std::sort(employees.begin(), employees.end(),
          CompareByLastName());
```

Assuming the comparison class `CompareByLastName` is thread-safe, which is true for most comparison functions, parallelizing this sort is simple with C++ Parallel Algorithms. Include `<execution>` and add an execution policy to the function call:

```
std:sort(std::execution::par,
         employees.begin(), employees.end(),
         CompareByLastName());
```

Calculating the sum of all the elements in a container is also simple with the `std::accumulate` algorithm. Prior to C++17, transforming the data in some way while taking the sum was somewhat awkward. For example, to compute the average age of your employees, you might write the following code:

```
int ave_age =
  std::accumulate(employees.begin(), employees.end(), 0,
                  [](int sum, const Employee& emp){
                      return sum + emp.age();
                  })
  / employees.size();
```

The `std::transform_reduce` algorithm introduced in C++17 makes it simple to parallelize this code. It also results in cleaner code by separating the reduction operation, in this case `std::plus`, from the transformation operation, in this case `emp.age()`:

```
int ave_age =
  std::transform_reduce(std::execution::par_unseq,
                        employees.begin(), employees.end(),
                        0, std::plus<int>(),
                        [](const Employee& emp){
                            return emp.age();
                        })
  / employees.size();
```

15.2.4. OpenACC Implementation of Parallel Algorithms

NVC++ has an experimental GPU support for a subset of algorithms with parallel execution policies `std::par` and `std::par_unseq` accelerated through the OpenACC implementation. This feature, enabled with the `-stdpar=gpu:acc` option, may result in better application performance on the GPU and faster compilation speed.

The following subset of algorithms have OpenACC implementation support:

- `std::adjacent_find`
- `std::all_of`, `std::any_of`, `std::none_of`
- `std::copy`, `std::copy_n`
- `std::count`, `std::count_if`
- `std::equal`
- `std::fill`, `std::fill_n`
- `std::find`, `std::find_if`
- `std::for_each`, `std::for_each_n`
- `std::max_element`, `std::min_element`, `std::minmax_element`
- `std::merge`
- `std::mismatch`
- `std::reduce`
- `std::replace`, `std::replace_if`, `std::replace_copy`, `std::replace_copy_if`
- `std::reverse`, `std::reverse_copy`
- `std::rotate_copy`
- `std::search`
- `std::swap_ranges`
- `std::transform`
- `std::transform_reduce`

The remainder of the parallel algorithms are parallelized using the default GPU implementation as if `-stdpar=gpu` was specified.

When the code is compiled for GPU with the OpenACC acceleration `__NVCOMPILER_STDPAR_OPENACC_GPU` macro is defined implicitly.

15.2.5. Coding Guidelines for GPU-accelerating Parallel Algorithms

GPUs are not simply CPUs with more threads. To effectively take advantage of the massive parallelism and memory bandwidth available on GPUs, it is typical for GPU programming models to put some limitations on code executed on the GPU. The NVC++ implementation of C++ Parallel Algorithms is no exception in this regard. The sections which follow detail the limitations that apply in the current release.

15.2.5.1 Parallel Algorithms and Device Function Annotations

Functions to be executed on the GPU within parallel algorithms do not need any `__device__` annotations or other special markings to be compiled for GPU execution. The NVC++ compiler walks the call graph for each source file and automatically infers which functions must be compiled for GPU execution.

However, this only works when the compiler can see the function definition in the same source file where the function is called. This is true for most inline functions and template functions but may fail when functions are defined in a different source file or linked in from an external library. You need to be aware of this when formulating parallel algorithms invocations that you expect to be offloaded and accelerated on NVIDIA GPUs.

When calling an externally defined function from within a parallel algorithm region, such functions require some form of device annotations from other GPU programming models e.g. OpenACC routine directive (refer to *External Device Function Annotations* for more information).

15.2.5.2 Data Management in Parallel Algorithms

When offloading parallel algorithms to a GPU, it's essential to consider how data is accessed from the parallel region. Some GPUs may not access certain segments of the CPU's address space. Developers targeting platforms without unified shared memory or those seeking to optimize performance must be aware of these memory distinctions, as they may affect the folowing types of data accessed in parallel algorithm regions:

- Pointer data passed into lambda functions within the parallel algorithm.
- Data captured by reference in lambda functions or pointer data captured by value.
- Variables with static storage duration referenced inside the parallel algorithm.

To avoid memory access violations, developers must ensure that all of the above data is accessible to the GPU before the parallel algorithm is executed.

Stdpar C++ only supports *Managed and Unified Memory Modes* which allow data being accessed from CPU and GPU. Through support in both the CUDA device driver and the NVIDIA GPU hardware, the CUDA Unified Memory manager automatically moves some types of data based on usage.

Stdpar with Separate Memory Mode can only be supported when the data are fully managed through the OpenACC data directives, refer to *Interoperability with OpenACC*.

Since object-oriented design is fundamental to C++, special consideration must be given to composite data types with pointer or reference members. The data referenced or pointed to may not be stored contiguously within the composite data type. Moreover, such data might not even be allocated in the same memory segment as the composite type itself. As a result, when accessing both the composite data type and its referenced or pointed-to data from parallel algorithms, the developer must ensure that the member data is also made accessible to the GPU. These considerations should also be taken into account when standard library containers are used in the parallel algorithms as the containers frequently contain member pointers to their elements.

The discussion in this section assumes familiarity with the Managed and Unified Memory Modes covered in *Memory Model* and *Managed and Unified Memory Modes*. The code executing within the parallel algorithm is referred to as the accelerator subprogram. In contrast to the code executing outside of the parallel algorithm which is referred to as the host subprogram.

Managed Memory Mode

When Stdpar code is compiled with Managed Memory Mode (as default mode or by passing `-gpu=mem:managed`) only data dynamically allocated on the heap in CPU code can be managed automatically. CPU and GPU automatic storage (stack memory) and static storage (global or static data) cannot be automatically managed. Likewise, data that is dynamically allocated in program units not compiled by `nvc++` with the `-stdpar` option is not automatically managed by CUDA Unified Memory even though it is on the CPU heap. The compiler utilizes CUDA Managed Memory for dynamic allocations to make data accessible from CPU and GPU. As managed memory allocation calls can incur higher runtime overhead than standard allocator calls, the implementation uses memory pools for performance reasons by default as detailed in *Memory Pool Allocator <gpu-mem-poolallocator>*.

The Managed Memory Mode is intended for binaries run on targets with CUDA Managed Memory capability only. Any pointer that is dereferenced and any C++ object that is referenced within a parallel algorithm invocation must refer to data on the CPU heap that is allocated in a program unit compiled by `nvc++` with `-stdpar`. Dereferencing a pointer to a CPU stack or a global object will result in a memory violation in GPU code.

Unified Memory Mode

When Unified Memory is the default memory mode or is selected explicitly on the command line by passing `-gpu=mem:unified`, there are no restrictions on variables accessed in the parallel algorithms. Therefore, all CPU data (either residing on stack, heap, or globally) are simply accessible in the parallel algorithm functions. Note that memory dynamically allocated in GPU code is only visible from GPU code and can never be accessed by the CPU regardless of the CUDA Unified Memory capability.

When compiling a binary for platforms with full CUDA Unified Memory capability, only those source files using features from the standard parallel algorithms library must be compiled by `nvc++` with the `-stdpar` option. There is no requirement that the code dynamically allocating memory accessed on GPU is also compiled in such a way.

Unified Memory Mode may utilize CUDA Managed Memory for dynamic allocation, more details can be found in *Managed and Unified Memory Modes*.

Summary

The following table provides a key summary of important command-line options selecting memory modes and the impact of memory modes on the Stdpar features.

Table 25: Stdpar C++ Feature Differences for Memory Modes

Command-line options	Dynamically allocated variables outside of parallel algorithm region	Automatic or static storage variables outside of parallel algorithm region	Dynamic allocator
No memory-specific flags passed, compiling on target with CUDA Managed Memory only	Can be accessed within parallel region code	Cannot be accessed within parallel algorithm code	cudaMallocManaged
No memory-specific flags passed, compiling on target with full CUDA Unified Memory	Can be accessed within parallel region code	Can be accessed within parallel algorithm code	cudaMallocManaged or system allocators: new/malloc (compiler picks the most suitable allocator)
`-gpu=mem:managed`	Can be accessed within parallel region code	Cannot be accessed within parallel algorithm code	cudaMallocManaged
`-gpu=mem:unified`	Can be accessed within parallel region code	Can be accessed within parallel algorithm code	cudaMallocManaged or system allocators: new/malloc (compiler picks the most suitable allocator)
`-gpu=mem:unified:managedalloc`	Can be accessed within parallel region code	Can be accessed within parallel algorithm code	cudaMallocManaged
`-gpu=mem:unified:nomanagedalloc`	Can be accessed within parallel region code	Can be accessed within parallel algorithm code	System allocators: new/malloc

Examples

For example, `std::vector` uses dynamically allocated memory, which is accessible from the GPU when using Stdpar. Iterating over the contents of a `std::vector` in a parallel algorithm works as expected when compiling with either `-gpu=mem:managed` or `-gpu=mem:unified`:

```
std::vector<int> v = ...;
std::sort(std::execution::par,
          v.begin(), v.end()); // Okay, accesses heap memory.
```

On the other hand, `std::array` performs no dynamic allocations. Its contents are stored within the `std::array` object itself, which is often on a CPU stack. Iterating over the contents of a `std::array` will not work on systems with only CUDA Managed Memory support unless the `std::array` itself is allocated on the heap and the code is compiled with `-gpu=mem:managed`:

```
std::array<int, 1024> a = ...;
std::sort(std::execution::par,
           a.begin(), a.end()); // Fails on targets with CUDA Managed
                                // Memory capability only, array is on
                                // a CPU stack inaccessible from GPU.
                                // Works correctly on targets whith full
                                // CUDA Unified Memory support.
```

The above example works as expected when run on a target supporting full CUDA Unified Memory

capability.

When executing on targets with CUDA Managed Memory capability only, pay particular attention to lambda captures, especially capturing data objects by reference, which may contain non-obvious pointer dereferences:

```
void saxpy(float* x, float* y, int N, float a) {
   std::transform(std::execution::par_unseq, x, x + N, y, y,
                  [&](float xi, float yi){ return a * xi + yi; });
}
```

In the earlier example, the containing function parameter a is captured by reference. The code within the body of the lambda, which is running on the GPU, tries to access a, which is in the CPU stack memory. This attempt results in a memory violation and undefined behavior. In this case, the problem can easily be fixed by changing the lambda to capture by value:

```
void saxpy(float* x, float* y, int N, float a) {
   std::transform(std::execution::par_unseq, x, x + N, y, y,
                  [=](float xi, float yi){ return a * xi + yi; });
}
```

With this one-character change, the lambda makes a copy of a, which is then copied to the GPU, and there are no attempts to reference CPU stack memory from GPU code. Such code will run correctly without requiring modifications on targets with full CUDA Unified Memory capability.

If `std::vector` is accessed through a subscript operator from the device this would require such a vector object to be accessible from the parallel code executing on the GPU. This means that the `std::vector` needs to be allocated dynamically in order to make it accessible from the GPU when compiled for the systems with only CUDA Managed Memory support.

```
std::vector<int> v = ...;
std::for_each(std::execution::par,
              idx.begin(), idx.end(), [&](auto i)
              {v[i] = 1;}); // Fails on targets with CUDA Managed
                            // Memory capability only, vector object is on
                            // a CPU stack inaccessible from GPU.
                            // Works correctly on targets with full
                            // CUDA Unified Memory support.
```

An alternative approach to managing the content of the `std::vector` on systems with CUDA Managed Memory support only would be to obtain a pointer to its elements data region using `data()` member.

```
std::vector<int> v = ...;
int* vdataptr = v.data();
std::for_each(std::execution::par,
              idx.begin(), idx.end(), [&](auto i)
              {vdataptr[i] = 1;}); // Works, vector elements are in heap
                                   // memory
```

Whether `-gpu=mem:unified` is enabled by default or passed explicitly on the command line, parallel algorithms can access global variables and accesses to global variables from CPU and GPU are kept in sync. Extra care should be taken when accessing global variables within parallel algorithms, as simultaneous updates in different iterations running on the GPU can lead to data races. The following example illustrates the safe update of a global variable in the parallel algorithm since the update only occurs in one iteration.

```
int globvar = 123;
void foo() {
  auto r = std::views::iota(0, N);
  std::for_each(std::execution::par_unseq, r.begin(), r.end(),
                [](auto i) {
                       if (i == N - 1)
                         globvar += 345;
          });
  // globvar is equal to 468.
}
```

15.2.5.3 Parallel Algorithms and Function Pointers

Functions compiled to run on either the CPU or the GPU must be compiled into two different versions, one with the CPU machine instructions and one with the GPU machine instructions.

In the current implementation, a function pointer either points to the CPU or the GPU version of the functions. This causes problems if you attempt to pass function pointers between CPU and GPU code. You might inadvertently pass a pointer to the CPU version of the function to GPU code. In the future, it may be possible to automatically and seamlessly support the use of function pointers across CPU and GPU code boundaries, but it is not supported in the current implementation.

Function pointers can't be passed to Parallel Algorithms to be run on the GPU, and functions may not be called through a function pointer within GPU code. For example, the following code example won't work correctly:

```
void square(int& x) { x = x * x; }
void square_all(std::vector<int>& v) {
  std::for_each(std::execution::par_unseq,
                v.begin(), v.end(), &square);
}
```

It passes a pointer to the CPU version of the function square to a parallel `for_each` algorithm invocation. When the algorithm is parallelized and offloaded to the GPU, the program fails to resolve the function pointer to the GPU version of `square`.

You can often solve this issue by using a function object, which is an object with a function call operator. The function object's call operator is resolved at compile time to the GPU version of the function, instead of being resolved at run time to the incorrect CPU version of the function as in the previous example. For example, the following code example works:

```
struct squared {
  void operator()(int& x) const { x = x * x; }
};
void square_all(std::vector<int>& v) {
  std::for_each(std::execution::par_unseq,
                v.begin(), v.end(), squared{});
}
```

Another possible workaround is to change the function to a lambda, because a lambda is implemented as a nameless function object:

```
void square_all(std::vector<int>& v) {
  std::for_each(std::execution::par_unseq, v.begin(), v.end(),
                [](int& x) { x = x * x; });
}
```

If the function in question is too big to be converted to a function object or a lambda, then it should be possible to wrap the call to the function in a lambda:

```
void compute(int& x) {
  // Assume lots and lots of code here.
}
void compute_all(std::vector<int>& v) {
  std::for_each(std::execution::par_unseq, v.begin(), v.end(),
                [](int& x) { compute(x); });
}
```

No function pointers are used in this example.

The restriction on calling a function through a function pointer unfortunately means passing polymorphic objects from CPU code to GPU-accelerated Parallel Algorithms is not currently supported, as virtual tables are implemented using function pointers.

15.2.5.4 Random Access Iterators

The C++ Standard requires that the iterators passed to most C++ Parallel Algorithms be forward iterators. However, C++ Parallel Algorithms on GPUs only works with random access iterators. Passing a forward iterator or a bidirectional iterator to a GPU/CPU-accelerated Parallel Algorithm results in a compilation error. Passing raw pointers or Standard Library random access iterators to the algorithms has the best performance, but most other random-access iterators work correctly.

15.2.5.5 Interoperability with the C++ Standard Library

Large parts of the C++ Standard Library can be used with stdpar on GPUs.

- `std::atomic<T>` objects within GPU code work provided that T is a four-byte or eight-byte integer type.
- Math functions that operate on floating-point types—such as `sin`, `cos`, `log`, and most of the other functions declared in `<cmath>` —can be used in GPU code and resolve to the same implementations that are used in CUDA C++ programs.
- `std::complex`, `std::tuple`, `std::pair`, `std::optional`, `std::variant`, and `<type_traits>`, are supported and work as expected in GPU code.

The parts of the C++ Standard Library that aren't supported in GPU code include I/O functions and in general any function that accesses the CPU operating system. As a special case, basic `printf` calls can be used within GPU code and leverage the same implementation that is used in NVIDIA CUDA C++.

15.2.5.6 No Exceptions in GPU Code

As with most other GPU programming models, throwing and catching C++ exceptions is not supported within Parallel Algorithm invocations that are offloaded to the GPU.

Unlike some other GPU programming models where try/catch blocks and throw expressions are compilation errors, exception code does compile but with non-standard behavior. Catch clauses are ignored, and throw expressions abort the GPU kernel if actually executed. Exceptions in CPU code work without restrictions.

15.2.6. Stdpar C++ Standard Library Extensions

In addition to the C++ standard parallel algorithms, NVC++ supports a set of C++ standard library features that complement and extend the Stdpar programming model. These features provide multidimensional data abstractions and linear algebra operations that integrate with parallel execution, enabling expressive and portable high-performance code. The `-stdpar` compiler option enables GPU or CPU multicore acceleration of supported operations in these libraries.

Table 26 lists all standard library extensions available and the minimum language version required to use them.

Table 26: C++ standard library features information

Feature	Recommended	Limited support	Standard proposal	Other notes
Multi-dimensional spans (mdspan)	C++23	C++17	P0009	https://github.com/NVIDIA/libcudacxx
Slices of multi-dimensional spans (submdspan)	C++23	C++17	P2630	https://github.com/NVIDIA/libcudacxx
Linear algebra	C++26	C++17	P1673	https://github.com/kokkos/stdblas

15.2.6.1 Multi-dimensional Spans

Multi-dimensional spans (`std::mdspan`) enable customizable multi-dimensional access to data. This feature was added to C++23 (see P0009 and follow-on papers) along with the closely related multi-dimensional arrays (`std::mdarray`, see P1684) and sub-spans (`std::submdspan`, see P2630). A Gentle Introduction to mdspan gives a tutorial. The reference mdspan implementation https://github.com/kokkos/mdspan also has many useful examples.

nvc++ provides an implementation of `std::mdspan` available via the `<mdspan>` header that works with C++17 or newer. It enables applications that are not targeting the C++23 version of the standard to use mdspan.

In addition, nvc++ back-ports the multidimensional subscript operator (`[i, j]` syntax) to C++17 or newer, as shown in the example below.

The following example (godbolt):

```
#include <mdspan>
#include <iostream>


int main() {
 std::array d{
   0, 5, 1,
   3, 8, 4,
   2, 7, 6,
 };


 std::mdspan m{d.data(), std::extents{3, 3}};
 static_assert(m.rank()==2, "Rank is two");
```

(continues on next page)

(continued from previous page)

```
 for (std::size_t i = 0; i < m.extent(0); ++i)
   for (std::size_t j = 0; j < m.extent(1); ++j)
     std::cout << "m[" << i << ", " << j << "] == " << m[i, j] << "\n";


 return 0;
}
```

is compiled as follows

```
nvc++ -stdpar -std=c++17 -o example example.cpp
```

and outputs

```
m[0, 0] == 0
m[0, 1] == 5
m[0, 2] == 1
m[1, 0] == 3
m[1, 1] == 8
m[1, 2] == 4
m[2, 0] == 2
m[2, 1] == 7
m[2, 2] == 6
```

15.2.6.2 Linear Algebra

P1673 - A free function linear algebra interface based on the BLAS proposes standardizing an idiomatic C++ interface based on std::mdspan for a subset of the Basic Linear Algebra Subroutines (BLAS) standard. For an introduction to this feature, see P1673 (C++ linear algebra library) background & motivation. There are many useful examples available in $HPCSDK_HOME/examples/stdpar/stdblas and in the repository of the reference implementation. A detailed documentation is available at $HPCSDK_HOME/compilers/include/__linalg/__p1673_bits/README.md. nvc++ provides an implementation of the C++ linear algebra library (`std::linalg`) available via the `<linalg>` header that works in C++17 or newer. To use the linear algebra library facilities, a suitable linear algebra library must be linked: cuBLAS for GPU execution via the `-cudalib=cublas` flag, and a CPU BLAS library for CPU execution. The HPC SDK bundles OpenBLAS which may be linked using the `-lblas` linker flag. For some functions, LAPACK APIs are used. Therefore, it is recommended to add the `-llapack` linker flag as well.

Execution	BLAS library	Architectures	Compiler flags
Multicore	OpenBLAS	x86_64, aarch64	`-stdpar=multicore -lblas -llapack`
GPU	cuBLAS	All	`-stdpar=gpu -cudalib=cublas`

The following example (godbolt):

```
#include <mdspan>
#include <linalg>
#include <vector>
#include <array>
```

(continues on next page)

(continued from previous page)

```
int main()
{
 constexpr size_t N = 4;
 constexpr size_t M = 2;


 std::vector<double> A_vec(N*M);
 std::vector<double> x_vec(M);
 std::array<double, N> y_vec(N);


 std::mdspan A(A_vec.data(), N, M);
 std::mdspan x(x_vec.data(), M);
 std::mdspan y(y_vec.data(), N);


 for(int i = 0; i < A.extent(0); ++i)
   for(int j = 0; j < A.extent(1); ++j)
     A[i,j] = 100.0 * i + j;


 for(int j = 0; j < x.extent(0); ++j) x[j] = 1.0 * j;
 for(int i = 0; i < y.extent(0); ++i) y[i] = -1.0 * i;


 std::linalg::matrix_vector_product(A, x, y); // y = A * x


 // y = 0.5 * y + 2 * A * x
 std::linalg::matrix_vector_product(std::execution::par,
                                    std::linalg::scaled(2.0, A), x,
                                    std::linalg::scaled(0.5, y), y
                                   );


 // Print the results:
 for (int i = 0; i < N; ++i) std::printf("y[%d] = %f\n", i, y[i]);
 return 0;
}
```

is compiled as follows for GPU execution:

```
nvc++ -std=c++17 -stdpar=gpu -cudalib=cublas -o example example.cpp
```

And as follows for CPU execution:

```
nvc++ -std=c++17 -stdpar=multicore -o example example.cpp -lblas
```

and produces the same outputs in both cases:

```
y[0] = 2.500000
y[1] = 252.500000
y[2] = 502.500000
y[3] = 752.500000
```

15.2.7. NVC++ Experimental Features

nvc++ experimental features are enabled with the –experimental-stdpar compiler flag. Experimental feature headers are exposed via the `<experimental/...>` namespaces and limited support for these features is available in older C++ versions. Table 27 lists all experimental features available and the minimum language version required to use them.

Table 27: Experimental features information

Feature	Recommended	Limited support	Standard proposal	Other notes
Multi-dimensional arrays (mdarray)	C++23	C++17	P1684	https://github.com/kokkos/mdspan
Senders and receivers	C++23	C++20	P2300	https://github.com/NVIDIA/stdexec

15.2.7.1 Senders and Receivers

P2300 - std::execution proposes a model of asynchronous programming for adoption into the C++26 Standard. For an introduction to this feature, see Design - user side section of the proposal. The NVIDIA implementation of Senders and receivers is open source and its repository contains many useful examples. nvc++ provides access to the NVIDIA implementation which works in C++20 or newer. Since the proposal is still evolving, our implementation is not stable. It is experimental in nature and will change to follow the proposal closely without any warning. The NVIDIA implementation is structured as follows:

Includes	Namespace	Description
<stdexec/...>	::stdexec	Approved for C++ standard
<sexec/...>	::exec	Generic additions and extensions
<nvexec/...>	::nvexec	NVIDIA-specific extensions and customizations

The following example (godbolt) builds a task graph in which two different vectors, v0 and v1, are concurrently modified in bulk, using a CPU thread pool and a GPU stream context, respectively. This graph then transfers execution to the CPU thread pool, and adds both vectors into v2 on the CPU, returning the sum of all elements:

```
int main()
{
  // Declare a pool of 8 worker CPU threads:
  exec::static_thread_pool pool(8);


  // Declare a GPU stream context:
  nvexec::stream_context stream_ctx{};


  // Get a handle to the thread pool:
  auto cpu_sched = pool.get_scheduler();
  auto gpu_sched = stream_ctx.get_scheduler();
```

(continues on next page)

(continued from previous page)

```
  // Declare three dynamic array with N elements
  std::size_t N = 5;
  std::vector<int> v0 {1, 1, 1, 1, 1};
  std::vector<int> v1 {2, 2, 2, 2, 2};
  std::vector<int> v2 {0, 0, 0, 0, 0};


  // Describe some work:
  auto work = stdexec::when_all(
    // Double v0 on the CPU
    stdexec::just()
      | exec::on(cpu_sched,
                 stdexec::bulk(N, [v0 = v0.data()](std::size_t i) {
                   v0[i] *= 2;
      })),
    // Triple v1 on the GPU
    stdexec::just()
      | exec::on(gpu_sched,
                 stdexec::bulk(N, [v1 = v1.data()](std::size_t i) {
                   v1[i] *= 3;
    }))
  )
  | stdexec::transfer(cpu_sched)
  // Add the two vectors into the output vector v2 = v0 + v1:
  | stdexec::bulk(N, [&](std::size_t i) { v2[i] = v0[i] + v1[i]; })
  | stdexec::then([&] {
    int r = 0;
    for (std::size_t i = 0; i < N; ++i) r += v2[i];
      return r;
  });
  auto [sum] = stdexec::sync_wait(work).value();
  // Print the results:
  std::printf("sum = %d\n", sum);
  for (int i = 0; i < N; ++i) {
    std::printf("v0[%d] = %d, v1[%d] = %d, v2[%d] = %d\n",
                i, v0[i], i, v1[i], i, v2[i]);
  }
  return 0;
}
```

is compiled as follows:

```
nvc++ --stdpar=gpu --experimental-stdpar -std=c++20 -o example example.cpp
```

and outputs:

```
sum = 40
v0[0] = 2, v1[0] = 6, v2[0] = 8
v0[1] = 2, v1[1] = 6, v2[1] = 8
v0[2] = 2, v1[2] = 6, v2[2] = 8
v0[3] = 2, v1[3] = 6, v2[3] = 8
v0[4] = 2, v1[4] = 6, v2[4] = 8
```

15.2.8. Stdpar C++ Larger Example: LULESH

The LULESH hydrodynamics mini-app was developed at Lawrence Livermore National Laboratory to stress test compilers and model performance of hydrodynamics applications. It is about 9,000 lines of C++ code, of which 2,800 lines are the core computation that should be parallelized.

We ported LULESH to C++ Parallel Algorithms and made the port available on LULESH's GitHub repository. To compile it, install the NVIDIA HPC SDK, check out the 2.0.2-dev branch of the LULESH repository, go to the correct directory, and `run make`.

```
git clone --branch 2.0.2-dev https://github.com/LLNL/LULESH.git
cd LULESH/stdpar/build
make run
```

While LULESH is too large to show the entire source code here, there are some key code sequences that demonstrate the use of stdpar.

The LULESH code has many loops with large bodies and no loop-carried dependencies, making them good candidates for parallelization. Most of these were easily converted into calls to `std::for_each_n` with the `std::execution::par` policy, where the body of the lambda passed to `std::for_each_n` is identical to the original loop body.

The function `CalcMonotonicQRegionForElems` is an example of this. The loop header written for OpenMP looks as follows:

```
#pragma omp parallel for firstprivate(qlc_monoq, qqc_monoq, \
                monoq_limiter_mult, monoq_max_slope, ptiny)
for ( Index_t i = 0 ; i < domain.regElemSize(r); ++i ) {
```

This loop header in the C++ Parallel Algorithms version becomes the following:

```
std::for_each_n(
  std::execution::par, counting_iterator(0), domain.regElemSize(r),
  [=, &domain](Index_t i) {
```

The loop body, which in this case is almost 200 lines long, becomes the body of the lambda but is otherwise unchanged from the OpenMP version.

In a number of places, an explicit `for` loop was changed to use C++ Parallel Algorithms that better express the intent of the code, such as the function `CalcPressureForElems`:

```
#pragma omp parallel for firstprivate(length)
for (Index_t i = 0; i < length ; ++i) {
  Real_t c1s = Real_t(2.0)/Real_t(3.0) ;
  bvc[i] = c1s * (compression[i] + Real_t(1.));
  pbvc[i] = c1s;
}
```

This function was rewritten as as follows:

```
constexpr Real_t cls = Real_t(2.0) / Real_t(3.0);
std::transform(std::execution::par,
  compression, compression + length, bvc,
  [=](Real_t compression_i) {
    return cls * (compression_i + Real_t(1.0));
  });
std::fill(std::execution::par, pbvc, pbvc + length, cls);
```

15.2.9. Interoperability with OpenACC

A subset of OpenACC features can be used when compiling Stdpar code for GPUs. Such a subset is documented in this section. To activate OpenACC directives recognition with Stdpar code add -acc command line flag to nvc++.

```
nvc++ -stdpar -acc example.cpp
```

OpenACC functionality is detailed in the OpenACC specification and the NVHPC compiler specific differences are detailed in *Using OpenACC* of this guide.

Combining OpenACC features with Stdpar offers greater flexibility in how code is written. For instance, it allows external functions to be called from within parallel algorithms. Additionally, it provides opportunities for performance tuning, such as through explicit data management.

15.2.9.1 Data Management Directives

C++ parallel algorithms can be offloaded to the GPU when the data accessed in such algorithms is managed through the OpenACC directives. With data fully managed through the OpenACC directives, Stdpar code can run with all GPU Memory Modes including Separate Memory Mode (compiled with `-gpu=mem:separate`).

The following data directives are supported:

- OpenACC structured data construct directive
- OpenACC unstructured enter/exit data directives
- OpenACC host_data directive
- OpenACC update directive

Only the data that are captured by reference or pointer-like data captured by values as well as pointer-like data passed as arguments in the parallel algorithm lambdas can be managed through OpenACC. Any non-pointer variables that are captured by value in the parallel algorithm lambda or non-pointer data passed in as lambda arguments are managed by the C++ implementation. A copy of such data is automatically created in the memory accessible from the GPU. For additional details refer to *Data Management in Parallel Algorithms*.

OpenACC data management can serve two main purposes:

- **Explicit Data Management:** This is necessary for data that cannot be managed implicitly, such as on platforms without full CUDA Unified Memory support and when data is not allocated in the CUDA Managed Memory segment.
- **Performance Tuning:** Even when data is located in the GPU-accessible memory, performance can be optimized via OpenACC features. Many OpenACC data directives and clauses provide hints to the CUDA device driver, which can improve implicit data management.

Data management strategies may differ depending on the specific goals being pursued. These differences are outlined where applicable.

General Rules

All directives, except `host_data`, can be used for data management tasks such as allocating memory in the GPU and copying data between the CPU and the GPU. These directives can be used to ensure that the data is present on the device during the execution of parallel algorithms. The `host_data` construct, on the other hand, is used for address translation between CPU and GPU address spaces when data is accessed in parallel algorithms.

```
int n = get_n();
T* in  = new T[nelem];
T* out = new T[nelem];
// Data captured by the lambda are managed explicitly with OpenACC
#pragma acc enter data copyin(n, in[0:nelem]) create(out[0:nelem])
#pragma acc host_data use_device(n, in, out)
{
  std::for_each(std::execution::par_unseq, r.begin(), r.end(),
                [&,in,out](auto i) {
                  out[i] = in[i] * n;
                });
}
#pragma acc exit data copyout(out[0:nelem])
```

In the above example all data accessed from `std::for_each` through the lambda capture are managed explicitly through the OpenACC data directives. Since the data inside the parallel algorithms are either captured by reference or capturing a pointer, the application code must ensure that such data is accessible from the GPU. To make non-GPU resident data accessible in the parallel region, such a region must be enclosed into the `host_data` construct region with all variables that are managed explicitly via OpenACC runtime listed in the `use_device` clause. The data need to be present (copied or created) at the time the `host_data` directive is encountered/executed at runtime and the data must also be present for the duration of parallel algorithm execution. The implications of the above are such that lambdas accessing variables enclosed in `use_device` regions can not be additionally invoked from the host code (from outside the parallel region executing on the GPU) because the variable addresses from the GPU obtained through `host_data` may not be accessible on the CPU.

Note: If the iterator in the above example would be a pointer type it would require explicit data management in addition to the data captured by the lambda.

If the example below is compiled for Separate Memory Mode (-gpu=mem:separate) calling `fn` from within a parallel `std::for_each` works fine but not from outside of any parallel algorithm function since the data resident on GPU would need to be accessed from the CPU.

```
int n = get_n();
T* in  = new T[nelem];
T* out = new T[nelem];
#pragma acc enter data copyin(n, in[0:nelem]) create(out[0:nelem])
#pragma acc host_data use_device(n, in, out)
{
  auto fn = [&,in,out](auto i) { out[i] = in[i] * n;};
  std::for_each(std::execution::par_unseq, r.begin(), r.end(), fn);
  // The following line would not be legal, fn accesses variables in GPU memory
  //std::for_each(r.begin(), r.end(), fn);
}
#pragma acc exit data copyout(out[0:nelem])
```

Note: The behavior of using `use_device` with non-pointer data type is such that all occurrences of non-pointer variables inside the `host_data` region are converted to using the addresses of the variable in the GPU address space before accessing that variable. This is essentially equivalent to translating original occurrences of such variable `var` into `dvar = *acc_device(&var)`.

Composite Data Types

Composite data types with pointer members can also be managed explicitly but require explicit deep

copy to work correctly including pointer attach/detach.

```
struct S {
  float *ptr;
}

int idx[N] = {/*...*/};
float arr[N];
S s{arr};
// Deep copying ptr member with OpenACC
#pragma acc enter data copyin(s.ptr[0:N])
#pragma acc enter data copyin(s, idx)
#pragma acc data attach(s.ptr)
#pragma acc host_data use_device(s, idx)
{
  std::for_each_n(std::execution::par, idx, N,
                  [&](int i) { s.ptr[i] += 5.0; });
}
#pragma acc exit data copyout(s.ptr[0:N])
#pragma acc exit data copyout(s)
```

When variable of struct S type in the above example is copied to the device, a deep copy is performed with the content pointed by `S.ptr` copied separately. The pointer attachment is used to ensure the address of the pointer is changed to the device memory equivalent before it is accessed from the GPU. Depending on the order of the copies, the pointer `attach` clause may not be required.

Note: In the above example the pointer-like iterator `idx` is managed through the OpenACC directives in addition to the data captured by the lambda.

Standard Containers

If the standard containers with non-contiguous storage must be used in host code with explicit data management to GPU memory, the only viable option is to access the raw data directly using the raw pointer to data (e.g. obtained via `data()` member of `std::vector`) unless the iterator over the data can be used.

```
std::vector<T> in(nelem);
std::vector<T> out(nelem);
T *inptr=in.data(),*outptr=out.data();
#pragma acc data copyin(inptr[0:nelem]) copyout(outptr[0:nelem])
#pragma acc host_data use_device(inptr,outptr)
{
  std::for_each(std::execution::par_unseq, r.begin(), r.end(),
                [=](auto i) {
                  outptr[i] = inptr[i];
                });
}
```

In the above example vector elements are accessed through raw pointers to their elements obtained through `vector::data()` member, they are explicitly management through the OpenACC data clauses.

Static Storage Data

Global or static variables can be made accessible in the parallel algorithms using OpenACC data directives similarly to other variables.

```
int glob_arr[N] = {/*...*/};
void foo(){
#pragma acc data copy(glob_arr)
#pragma acc host_data use_device(glob_arr)
  {
    std::for_each_n(std::execution::par, glob_arr, N,
                    [](int &e) { e += 1; });
  }
}
```

In the above example the global array `glob_arr` is updated on the GPU with help of OpenACC data directives.

Member Functions

When the data members are managed inside the member functions the implicit object pointer `this` needs to be explicitly managed for correctness as accessing members is always done through the dereference of the object itself.

```
struct S {
  float *ptr;

  void update_member() {
#pragma acc data copy(ptr[0:N], this)
#pragma acc host_data use_device(ptr, this)
  {
      std::for_each(std::execution::par, ptr, ptr + N,
                    [=](float &e) { ptr[&e - ptr] += 5.0; });
   }
  }
};
```

GPU Memory Mode Related Differences

In Separate Memory Mode all data must be managed explicitly via extra device allocations and `memcpy` between the host and device and the address translations. This also applies to variables with automatic or static storage duration in Managed Memory Mode.

In Unified Memory Mode all data is automatically managed by the CUDA device driver. Additionally in Managed Memory Mode all dynamic allocations are managed by the CUDA device driver. Use of data clauses and directives can only propagate memory usage hints to the CUDA device driver which are used to improve the data management performance. More details can be found in Memory Model and *OpenACC with CUDA Unified Memory* .

All the data managed by the CUDA device driver can benefit from the simplified uses of the OpenACC features, particularly:

- Use of `host_data` directive is not required since the host and device address of data in unified shared memory is identical.
- Use of pointer attach or detach is not required since the host and device pointers in unified shared memory are identical.

The following example illustrates simplified data managment with only OpenACC data construct enclosing the `std::for_each` with Unified Memory Mode.

```
int n = get_n();
T* in = new T[nelem];
T* out = new T[nelem];
```

(continues on next page)

(continued from previous page)

```
#pragma acc data copyin(in[0:nelem]) copyout(out[0:nelem])
{
  std::for_each(std::execution::par_unseq, r.begin(), r.end(),
                [&](auto i) {
                  out[i] = in[i] * n;
                });
}
```

In the above example we leverage OpenACC explicit data management construct to indicate how data is used on GPU for the computation executed in `std::for_each`:

- `in` is moved into the GPU memory;
- `out` is moved from the GPU memory.

Both `in` and `out` are captured by reference and therefore their host address is used in the lambda of `std::for_each`. The scalar variable `n` is not managed. The use of `host_data` construct is not required.

When standard containers are used in data directives and clauses, the underlying data collection can be managed too. For example, in order to indicate that elements of the `std::vector` are accessed from the GPU the application code must first retrieve the pointer to the array elements using its `data()` member. Then such pointers can be used in the regular data directives.

```
std::vector<T> in(nelem);
std::vector<T> out(nelem);
T *inptr=in.data(), *outptr=out.data();
#pragma acc data copyin(inptr[0:nelem]) copyout(outptr[0:nelem])
{
  std::for_each(std::execution::par_unseq, r.begin(), r.end(),
                [&](auto i) {
                  out[i] = in[i];
                });
}
```

The above example demonstrates the use of OpenACC data directives with a raw pointer to elements of `std::vector` which can improve memory performance for data in unified memory and the full deep copy of vector content using attach/detach is not required.

```
int n = get_n();
T* in = new T[nelem];
T* out = new T[nelem];
#pragma acc enter data copyin(n)
#pragma acc host_data use_device(n)
{
  std::for_each(std::execution::par_unseq, r.begin(), r.end(),
                [&, in, out](auto i) {
                  out[i] = in[i] * n;
                });
}
#pragma acc enter data delete(n)
```

In the above example, `in` and `out` are dynamically allocated and managed by CUDA device driver with Managed Memory Mode, `n` is on the stack and therefore managed explicitly via OpenACC directives.

15.2.9.2 External Device Function Annotations

Using OpenACC routine directive annotations allows calling external device functions.

```
// In file1.cpp
extern int foo();

void bar()
{
  std::for_each(std::execution::par_unseq, r.begin(), r.end(),
                [=](auto i) {
                  ou[i] = foo();
                });
}
// In file2.cpp
#pragma acc routine
int foo(){
  return 4;
}
```

The above code can be compiled/linked as follows:

```
nvc++ -stdpar file1.cpp
nvc++ -acc file2.cpp
nvc++ -stdpar -acc file1.o file2.o
```

15.2.10. Getting Started with Parallel Algorithms for GPUs

To get started, download and install the NVIDIA HPC SDK on your x86-64 or Arm CPU-based system running a supported version of Linux.

The NVIDIA HPC SDK is freely downloadable and includes a perpetual use license for all NVIDIA Registered Developers, including access to future release updates as they are issued. After you have the NVIDIA HPC SDK installed on your system, the nvc++ compiler is available under the `/opt/nvidia/hpc_sdk` directory structure.

- To use the compilers including nvc++ on a Linux/x86-64 system, add the directory `/opt/nvidia/hpc_sdk/Linux_x86_64/26.11/compilers/bin` to your path.
- On an Arm CPU-based system, replace `Linux_x86_64` with `Linux_aarch64`.

15.2.10.1 Supported NVIDIA GPUs

The NVC++ compiler can automatically offload C++ Parallel Algorithms to NVIDIA GPUs based on the Volta architecture or newer. These architectures include features – such as independent thread scheduling and hardware optimizations for CUDA Unified Memory – that were specifically designed to support high-performance, general-purpose parallel programming models like the C++ Parallel Algorithms.

The NVC++ compiler provides limited support for C++ Parallel Algorithms on the Pascal architecture, which does not have the independent thread scheduling necessary to properly support the `std::execution::par` policy. When compiling for the Pascal architecture (-gpu=cc60), NVC++ compiles algorithms with the `std::execution::par` policy for serial execution on the CPU. Only algorithms with the `std::execution::par_unseq` policy will be scheduled to run on Pascal GPUs.

15.2.10.2 Supported CUDA Versions

The NVC++ compiler is built on CUDA libraries and technologies and uses CUDA to accelerate C++ Parallel Algorithms on NVIDIA GPUs. A GPU-accelerated system on which NVC++-compiled applications are to be run must have a CUDA 11.2 or newer device driver installed.

The NVIDIA HPC SDK compilers ship with an integrated CUDA toolchain, header files, and libraries to use during compilation, so it is not necessary to have a CUDA Toolkit installed on the system.

When `-stdpar` is specified, NVC++ compiles using the CUDA toolchain version that best matches the CUDA driver installed on the system on which compilation is performed. To compile using a different version of the CUDA toolchain, use the `-gpu=cudaX.Y` option. For example, use the `-gpu=cuda12.9` option to specify that your program should be compiled for a CUDA 12.9 system using the CUDA 12.9 toolchain.

15.3. Stdpar Fortran

Fortran 2008 introduced the `do concurrent` (DC) loop construct signaling that loop iterations have no interdependencies. With `-stdpar` such loop iterations will be executed in parallel on the GPU when `-stdpar` (or `-stdpar=gpu`) is passed to `nvfortran` or using CPU threads when `-stdpar=multicore` is passed to `nvfortran`. More details can be found in the following blog post on the NVIDIA website: Accelerating Fortran DO CONCURRENT with GPUs and the NVIDIA HPC SDK.

15.3.1. Calling Routines in DO CONCURRENT on the GPU

When compiling for the GPU, calling routines in the body of `do concurrent` loop can be constrained. PURE routines can generally be called inside the `do concurrent` loop body. The compiler detects that such routines are to be compiled for the GPU target. External routines, however, can't be called from within the DC loop unless they are explicitly annotated with the OpenACC routine directive (refer to *Interoperability with OpenACC <stdpar-fortran-interop-openacc>*) or CUDA device attribute (refer to *Interoperability with CUDA Fortran*).

The following example will compile successfully.

```
module m
contains
pure subroutine foo()
return
end subroutine
end module m

program dc
use m
implicit none
integer :: i

do concurrent (i=1:10)
  call foo()
enddo
end program
```

The following example, however, doesn't compile unless `foo` is either

- annotated with `!$acc routine`,
- or attributed with `attributes(device)` and compiled as Stdpar and CUDA Fortran.

```
program dc
implicit none
interface
  pure subroutine foo()
  end subroutine foo
end interface
integer :: i

do concurrent (i=1:10)
  call foo()
enddo
end program
```

15.3.2. GPU Data Management

If `-gpu=mem:managed` is enabled by default or is explicitly passed on the command line, some data accesses in `do concurrent` loops are invalid. For example, accessing global variables in the routines called from the `do concurrent` loop does not perform expected value updates in the CPU code.

Additionally, there are rare instances where the compiler cannot accurately determine variable sizes for implicit data movements between CPU and GPU. As demonstrated in the following example, `a` is an assumed-size array, and its access region inside the DC construct cannot be determined at compile time because the element index positions are taken from another array `b` initialized outside of the routine. Such code does not update `a` as expected and may result in a memory violation and undefined behavior.

```
subroutine r(a, b)
  integer :: a(*)
  integer :: b(:)
  do concurrent (i = 1 : size(b))
    a(b(i)) = i
  enddo
end subroutine
```

There are no limitations on the variable accessed in `do concurrent` loops described above when the code is compiled with `-gpu=mem:unified`, whether this option is enabled by default or explicitly via an option on the command line.

15.3.3. Interoperability with OpenACC

OpenACC features can be used when compiling Stdpar code for GPUs. To activate OpenACC directives recognition with Stdpar code add `-acc` command line flag to `nvfortran`.

```
nvfortran -stdpar -acc example.f90
```

OpenACC functionality and interoperability with DO-CONCURRENT loop is detailed in the OpenACC specification and the NVIDIA HPC compiler specific differences are detailed in *Using OpenACC* of this guide.

Using OpenACC features can enhance functionality of DC-loop for example with the following:

- Explicit data management to improve performance of CPU-GPU implicit data movements or even leverage separate memory compiling on the GPU when compiling with `-gpu=mem:separate` passed in.
- Tuning DC-loop execution on the GPU e.g. GPU kernels launch configuration.
- Executing DC-loops asynchronously.
- Calling external routines from within DC-loops.
- Atomic operations in DC-loops.

Examples

Some examples of using OpenACC directives with DC-loops are provided below.

The following example demonstrates how the data accessed inside the DC-loop are fully managed in the OpenACC data construct.

```
!$acc data copyin(b) copyout(a)
do concurrent (j=1:N)
  do i=1,K
    a(j,i) = b(j,i)
  end do
end do
!$acc end data
```

While in the above example the data construct is used for GPU data management, the same effect can be achieved with the use of data clauses on the compute construct enclosing DC-loop.

The following example shows how the scheduling of DC loop on the GPU is controlled through the clauses on the compute construct.

```
!$acc parallel loop num_gangs(50000) vector_length(32)
do concurrent (i=1:K,j=1:N)
  a(j,i) = real(j)
end do
```

Use of OpenACC async clause on the compute constructs can be utilised to perform computations in DC-loop asynchronously.

```
!$acc parallel loop async
do concurrent (j=1:N)
  a(j) = j
end do

b = foo()

#pragma acc wait

c = sum(a) + b
```

In the previous example, array a is filled in with values asynchronously in DC-loop.

15.3.4. Interoperability with CUDA Fortran

CUDA Fortran features can also be used when compiling Stdpar code for GPUs. To recognize CUDA Fortran features in your source code, compile with the `-cuda` command line flag using `nvfortran`.

```
nvfortran -stdpar -cuda example.f90
```

Using CUDA Fortran extensions can enhance the functionality of a do concurrent (DC) loop and Stdpar program, for several cases:

- Explicit data locality, accessing CUDA Fortran attributed arrays or other data with the device, managed, unified, or constant attributes from within DC-loops.
- Tuning DC-loop execution on the GPU e.g. controlling the GPU kernels launch configuration.
- Executing DC-loops asynchronously using a specific CUDA stream.
- Calling external, user-defined CUDA device routines from within DC-loops.
- Using CUDA Atomic operations in DC-loops, or other CUDA-specific device-side runtime library calls.
- Inserting CUDA Runtime API calls for memory tuning hints outside of DC-loops.

Examples

Some examples of using CUDA Fortran features with DC-loops are provided below. The following example demonstrates how a DC-loop can access CUDA Fortran device data, run on a specific CUDA stream, call the CUDA Runtime API for creating a stream, and hide non-standard features behind the CUF sentinel for code portability.

```
!@cuf use cudafor
!@cuf integer(kind=cuda_stream_kind) :: istrm
      real, allocatable :: a(:,:), b(:,:)
!@cuf attributes(device) :: a  ! A is device array only, not unified/managed
      . . .
!@cuf istat = cudaStreamCreate(istrm)
      . . .
      a(:,:) = 0.0
      . . .
!$cuf kernel do(1) <<< *, *, stream=istrm>>>
      do concurrent (j=1:N)
        do i=1,K
          a(j,i) = a(j,i) + 2.0 * b(j,i)
        end do
      end do
```

This program demonstrates how to call low-level CUDA device functions from within a DC-loop. The function can be written in either CUDA Fortran or CUDA C++, depending on the interface. The CUDA C function must be compiled for relocatable device code. This can be used for accessing features in CUDA and NVIDIA GPUs not readily available in directive-based models or standard languages.

```
module mcuda
  contains
    attributes(host,device) pure integer function std_dbg(itype)
    integer, value :: itype
    if (itype.eq.1) then
      std_dbg = threadIdx%x
    else if (itype.eq.2) then
```

(continues on next page)

(continued from previous page)

```
      std_dbg = blockIdx%x
    else
      std_dbg = (blockIdx%x-1)*blockDim%x + threadIdx%x
    end if
    end function
end module

program test
use mcuda
integer, parameter :: N = 2000
integer, allocatable :: a(:), b(:), c(:)
allocate(a(N),b(N),c(N))

do concurrent (j=1:N)
  a(j) = std_dbg(1)
  b(j) = std_dbg(2)
  c(j) = std_dbg(3)
end do

print *,a(1),a(N/2),a(N)
print *,b(1),b(N/2),b(N)
print *,c(1),c(N/2),c(N)
end
```

Many functions from the CUDA Fortran `cudadevice` module are available within do concurrent loops, not just atomics. This code snippet shows two uses:

```
real :: tmp(4), x, y
...
block; use cudadevice
do concurrent (i=1:K,j=1:N)
  x = real(j) + a(i,j)
  y = atomicAdd(b(1,j), x)
end do

do concurrent (j=1:N)
  x = real(j)
  tmp(1:4) = __ldca(a(1:4,j))
  tmp(1:4) = tmp(1:4) + x
  call __stwt(b(1:4,j), tmp)
end do
end block
```

Chapter 16. 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.

16.1. 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 28: 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

16.2. 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

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

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;
    }
```

(continues on next page)

(continued from previous page)

```
    for (t = 0; t < 5; t++) {
    for(i = 0; i < size; i++) {
      a2[i] += a1[i];
    }
    pcast_compare(a2, "float", size, "a2", "example.c", "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.

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

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.

16.3. 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 correponding 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)
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
    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 = 0.00000000000000000e+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
```

16.4. 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 28. Currently there is no support for structured or derived types.
- The `-gpu=mem:managed` or `-gpu=mem:unified` options are incompatible with autocompare 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.

16.5. 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 29: PCAST_COMPARE Options

Option	Description
`abs=n`	Compare absolute difference; tolerate differences up to 10^(-n), only applicable to floating point types. Default value is 0
`create`	Specifies that this is the run that will produce the reference file (`pcast_compare` only)
`compare`	Specifies that the current run will be compared with a reference file (`pcast_compare` only)
`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)
`output-file="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)
`verboseauto-compare`	Outputs verbose reporting of what and where the host is comparing (autocompare only)

Chapter 17. Using MPI

MPI (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.

17.1. 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_64/26.11/examples/MPI/samples/mpihello .
$ cd mpihello
$ export PATH=/opt/nvidia/hpc_sdk/Linux_x86_64/26.11/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.

17.2. Using MPI Compiler Wrappers

When you use MPI compiler wrappers to build with the `-fpic` or `-mcmodel=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 `-mcmodel=medium`, use this command:

```
$ mpifort -mcmodel=medium -fortranlibs hello.f
```

17.3. Testing and Benchmarking

The /opt/nvidia/hpc_sdk/Linux_x86_64/26.11/examples/MPI directory contains various benchmarks and tests. Copy this directory into a local working directory by issuing the following command:

```
text
```

```
% cp -r /opt/nvidia/hpc_sdk/Linux_x86_64/26.11/examples/MPI .
```

There are several example programs available in this directory.

Chapter 18. 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.

Note: This section does not duplicate material related to using libraries for inlining which are described in *Creating an Inline Library*.

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

18.1. 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:

abs	acosf	asinf	atan	atan2	atan2f
atanf	cos	cosf	exp	expf	fabs
fabsf	fmax	fmaxf	fmin	fminf	log
log10	log10f	logf	pow	powf	sin
sinf	sqrt	sqrtf	tan	tanf	

18.2. Using System Library Routines

Release 26.11 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	

18.3. 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.

18.3.1. 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
```

Note: 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 myprof.f tobeshared.so -R/home/myusername/bin
```

Note: 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.

18.3.2. 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`.

18.4. 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.

18.5. 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
```

18.6. 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.

If the `-Mnvpl` switch is also specified in addition to `-Mscalapack`, then the NVPL ScaLAPACK library will be used.

18.7. 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.

18.8. NVIDIA Performance Libraries (NVPL)

The NVIDIA Performance Libraries (NVPL) are a suite of high performance mathematical libraries optimized for the NVIDIA Grace Arm architecture. These CPU-only libraries have no dependencies on CUDA or CTK, and are drop in replacements for standard C and Fortran mathematical APIs allowing HPC applications to achieve maximum performance on the Grace platform. They are available for Arm CPUs only. The NVPL includes the following math libraries: BLAS, FFT, LAPACK, RAND, ScaLAPACK, Sparse, and Tensor. Refer to the NVPL documentation for more information about these math libraries. The following section explains how to use them with the NVHPC compilers.

To use the NVPL libraries, use the `-Mnvpl` option when linking your main program:

```
% nvfortran myprog.f -Mnvpl
```

You can link only the NVPL libraries your application needs using the sub-options to `-Mnvpl`. For example, if you only want the BLAS and FFT libraries from the NVPL, link as follows:

```
% nvfortran myprog.f -Mnvpl=blas,fft
```

Refer to the NVIDIA HPC Compilers Reference Guide for a complete list of supported options for the `-Mnvpl` flag.

ScaLAPACK

Similar to other ScaLAPACK libraries, the NVPL version is designed to be used with MPI. A straightforward way to access the NVPL ScaLAPACK library is to use an MPI wrapper (i.e., `mpicc`, `mpic++`, `mpifort`) and link with both `-Mnvpl` and `-Mscalapack`. For example:

```
% mpic++ myprog.cpp -Mscalapack -Mnvpl
```

If you choose not to use an MPI wrapper, you can satisfy ScaLAPACK's dependency on libmpi.so by explicitly providing this library at link time.

The NVPL ScaLAPACK interfaces are available for the following MPI variants: MPICH, Open MPI 3.x, Open MPI 4.x (including HPC-X), and Open MPI 5.x. The HPC SDK contains builds of Open MPI 3, Open MPI 4, and HPC-X; to take advantage of the NVPL's ScaLAPACK interfaces for MPICH or Open MPI 5.x, you must supply your own build of these MPI libraries.

18.9. Linking with the nvmalloc Library

The NVIDIA HPC SDK installation includes a custom host (system) memory allocation library based on the jemalloc memory allocator. This library, nvmalloc, replaces the system malloc(), free(), and other related functions used by the nvc, nvc++, and nvfortran runtime for dynamic heap allocations. Using nvmalloc can improve the performance of your application as the underlying jemalloc avoids memory fragmentation and provides scalable concurrency. You can link with this library by specifying -nvmalloc on any of the compiler command lines used for linking. For example:

```
% nvc main.c -nvmalloc
```

The -nvmalloc option is turned on by default on Arm systems when building executables. The option -nonvmalloc can be used to turn off nvmalloc.

18.9.1. Using nvmalloc with Shared Libraries

The -nvmalloc option is not available when building shared libraries due to limitations inherent with function interception. Instead, use -nvmalloc when building the main executables to handle memory allocations and frees from all the linked shared libraries.

18.9.2. Performance Considerations

The environment variable `MALLOC_CONF` can be used to control the behavior of nvmalloc. The parameter `dirty_decay_ms` controls how nvmalloc returns unused memory pages back to the OS via a time-smoothed decay curve. While tuning is possible, users should first aim to avoid frequent allocation and deallocation patterns through buffer reuse or pooling whenever possible. For applications with unavoidable, bursty allocate/deallocate pattern, setting `MALLOC_CONF="dirty_decay_ms:-1"` may improve performance by reusing warm memory pages. However, this will result in an increase in the Resident Set Size (RSS) of the application. This configuration is not advisable when operating near the system's total memory capacity or in shared-use environments.

Chapter 19. 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; these environment variables are described in the OpenMP Specification available online.
- 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.

19.1. 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.

19.1.1. 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_x86-64/26.11/compilers/man
$ set path = ($NVHPCSDK/Linux_x86_64/26.11/compilers/bin $path)
```

For bash, sh, zsh, or ksh, use these commands:

```
$ NVHPCSDK=/opt/nvidia/hpc_sdk; export NVHPCSDK
$ MANPATH=$MANPATH:$NVHPCSDK/Linux_x86_64/26.11/compilers/man; export MANPATH
$ PATH=$NVHPCSDK/Linux_x86_64/26.11/compilers/bin:$PATH; export PATH
```

On a Linux/Arm Server system replace `Linux_x86_64` with `Linux_aarch64`.

19.2. HPC Compiler Related Environment Variables - General

The following table provides a listing of environment variables that affect the behavior of the NVIDIA HPC Compilers and the executables they generate.

Table 30: 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.
FORT_CONVERT	Allows the user to change the default byte ordering (endianness) of unformatted I/O files.
FORT_CONVERT_UNIT	Allows the user to change the default byte ordering (endianness) of specific unformatted I/O files.
FORT_FMT_RECL	Allows the user to change the default Fortran stdout (unit 6) line length before a line break occurs. Default: 80 bytes.
FORT_NO_EMPTY_ALLOCATION	Allows the user to make the ALLOCATE statement to produce an error if the allocated array size is negative or zero.
GMON_OUT_PREFIX	Specifies the name of the profiler 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.
NO_STOP_MESSAGE	If specified, the execution of a plain STOP statement does not produce the message `FORTRAN STOP`.
PATH	A colon separated list of directories use to search for executables.
NVCOMPILER_FPU_STATE	Manages the initial state of the processor's floating point control and status register at program startup.
NVCOMPILER_TERM	Controls the stack traceback and just-in-time debugging functionality.
NVCOMPILER_TERM_DEBUG	Specifies an alternate debugger to use when `NVCOMPILER_TERM` is set to `DEBUG`.
PWD	Absolute path of the current working directory.
STATIC_RANDOM_SEED	Forces the seed returned by `RANDOM_SEED` to be constant.
TMP	Sets the directory to use for temporary files created during execution of the HPC compilers and tools; interchangeable with `TMPDIR`.
TMPDIR	Sets the directory to use for temporary files created during execution of the HPC compilers and tools.

19.2.1. HPC Compilers Environment Variables - General

Use the environment variables listed in Table 30 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.

19.2.1.1 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
```

19.2.1.2 FORT_FMT_RECL

The FORT_FMT_RECL environment variable specifies the maximum line in bytes for Fortran formatted output to standard out (unit 6) before a newline will be generated.

If the environment variable FORT_FMT_RECL is present, the Fortran runtime library will use the value specified as the number of bytes to output before a newline is generated.

The default value of FORT_FMT_RECL is 80.

- In csh:

  ```
  $ setenv FORT_FMT_RECL length-in-bytes
  ```

- In bash, sh, zsh, or ksh:

  ```
  $ FORT_FMT_RECL=length-in-bytes
  $ export FORT_FMT_RECL
  ```

19.2.1.3 FORT_CONVERT

The FORT_CONVERT environment variable specifies the byte ordering (endianness) of Fortran unformatted I/O files.

If the environment variable FORT_CONVERT is present, the Fortran runtime library will use the value specified as the default byte ordering.

The implicit default value of FORT_CONVERT is NATIVE.

```
FORT_CONVERT_UNIT=<OPT>
<OPT>            'HELP' | <MODE>
<MODE>           'little_endian' | 'big_endian' | 'native' | 'swap'
```

- In csh:

```
$ setenv FORT_CONVERT <MODE>
```

- In bash, sh, zsh, or ksh:

```
$ FORT_FMT_CONVERT=<MODE>
$ export FORT_CONVERT
```

19.2.1.4 FORT_CONVERT_UNIT

The FORT_CONVERT_UNIT environment variable specifies the byte ordering (endianness) of specific Fortran unformatted I/O file(s).

If the environment variable FORT_CONVERT_UNIT is present, the Fortran runtime library will use the value specified as the default byte ordering for a specific file(s).

There is no implicit default value of FORT_CONVERT_UNIT.

```
FORT_CONVERT_UNIT=<OPT>
<OPT>            'HELP' | <MODE> | <MODE> ';' | <EXCEPTIONS>
<MODE>           'little_endian' | 'big_endian' | 'native' | 'swap'
<EXCEPTIONS>     <EXCEPTION> | <EXCEPTIONS> ';' <EXCEPTION>
<EXCEPTION>      <MODE> ':' <UNIT-LIST> | <UNIT-LIST>
<UNIT-LIST>      <UNIT> | <UNIT> ',' <UNIT-LIST>
<UNIT>           <INTEGER> | <INTEGER> '-' <INTEGER>
<INTEGER>        [[:digit:]]+

HELP             Print options.
<MODE>           Global endianness for all unformatted files.
<MODE> ';'       Global endianness for all unformatted files followed by
                 one or more I/O unit number exceptions.
<EXCEPTION>      Conversion method for specific unit number(s).
                 big_endian:     Unit(s) formatted with big endianness.
                 little_endian:  Unit(s) formatted with little endianness.
                 native:         Unit(s) formatted with host's endianness.
                 swap:           Unit(s) formatted with ~host's endianness.
```

- In csh:

```
$ setenv FORT_CONVERT_UNIT <OPT>
```

- In bash, sh, zsh, or ksh:

```
$ FORT_FMT_CONVERT_UNIT=<OPT>
$ export FORT_CONVERT_UNIT
```

Examples:

```
FORT_CONVERT_UNIT=big_endian
```

Use big endianness for all unformatted files.

```
FORT_CONVERT_UNIT=big_endian:10,20-30
```

Exempt units 10 and 20 through 30 from default endianness and force big endianness.

```
FORT_CONVERT_UNIT=10,20-30
```

Shorthand for:

```
FORT_CONVERT_UNIT=big_endian:10,20-30
```

```
FORT_CONVERT_UNIT=big_endian;little_endian:11,12,13;swap:14,15,16:native:17,20-22
```

Default endianness is big endian. Units 11,12,13 use little endianness. Units 14,15,16 use big endianness if host is little endian, else use little endianness. Units 17 and 20 through 22 use host endianess.

Note:

A) **Unformatted file I/O endianness determined in order by:**

1) Host's byte odering.
2) "CONVERT=" OPEN statement connection specifier.
3) Compiler option "-byteswapio".
4) "FORT_CONVERT=" environment variable.
5) "FORT_CONVERT_UNIT=" environment variable.

B) Exceptions are parsed left to right, so when an exception is specified multiple times for the same unit or range of units, the rightmost exception is used. Example:

`FORT_CONVERT_UNIT=swap:14-18;native:16`

Unit 16's endianness will use the host's byte ordering ("native").

C) <MODE>s and "HELP" are case insensitive.

19.2.1.5 FORT_NO_EMPTY_ALLOCATION

The FORT_NO_EMPTY_ALLOCATION environment variable allows the user to make the ALLOCATE statement produce an error if the allocated array size is negative or zero.

If the environment variable FORT_NO_EMPTY_ALLOCATION is present and is set to a non-zero integer value, the Fortran runtime library will change the behavior of the ALLOCATE statement to produce an error if the allocated array size is negative or zero.

The default value of FORT_NO_EMPTY_ALLOCATION is 0, which corresponds to the standard behavior of the ALLOCATE statement to treat a zero or negative array size as a valid empty allocation.

- In csh:

```
$ setenv FORT_NO_EMPTY_ALLOCATION 1
```

Consider the following example program that tries to allocate a very large array:

```
program huge
implicit none
integer :: s
real, allocatable :: x(:)
s = 2**33
allocate(x(s))
x(1:1000) = 0
end program
```

The calculated array size 2**33 may overflow the default integer kind (e.g. 32-bit) and become a negative number. However, the standard behavior of the ALLOCATE statement is to treat the negative or zero size value as a valid empty allocation. The access to the allocated array will therefore produce undefined behavior, likely a memory access violation.

By setting the FORT_NO_EMPTY_ALLOCATION environment variable to a non-zero value, you can make the program produce an immediate error at the ALLOCATE statement, which will terminate the program:

```
$ nvfortran huge.F
$ setenv FORT_NO_EMPTY_ALLOCATION 1
$ ./a.out
0: ALLOCATE: 0 bytes requested
```

You can also pass a status variable to the ALLOCATE statement to check if the allocation was successful:

```
program huge
implicit none
integer :: s, status
real, allocatable :: x(:)
s = 2**33
allocate(x(s), stat=status)
if (status /= 0) then
  write(*,*) 'ALLOCATE failed with status: ', status
  stop
end if
x(1:1000) = 0
end program
```

```
$ nvfortran huge.F
$ setenv FORT_NO_EMPTY_ALLOCATION 1
$ ./a.out
 ALLOCATE failed with status:             1
FORTRAN STOP
```

19.2.1.6 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
```

19.2.1.7 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":"./"
```

19.2.1.8 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_aarch64 targets:

```
$ set path = (/opt/nvidia/hpc_sdk/Linux_x86_64/26.11/compilers/bin $path)
$ setenv MANPATH "$MANPATH":/opt/nvidia/hpc_sdk/Linux_x86_64/26.11/compilers/
↪man
```

19.2.1.9 NO_STOP_MESSAGE

If the NO_STOP_MESSAGE variable exists, the execution of a plain STOP statement does not produce the message FORTRAN STOP. The default behavior of the NVIDIA Fortran compiler is to issue this message.

19.2.1.10 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_aarch64 are done similarly:

```
$ set path = (/opt/nvidia/hpc_sdk/Linux_x86_64/26.11/compilers/bin $path)
```

19.2.1.11 NVCOMPILER_FPU_STATE

The NVCOMPILER_FPU_STATE environment variable manages the initial state of the processor's floating point control and status register. NVCOMPILER_FPU_STATE eliminates the need to compile the main entry point (c/c++/Fortran) of programs with -M[no]daz, -M[no]flushz, or -Ktrap= command line options, as those options can now be specified at runtime.

Note: Linux only

If the environment variable NVCOMPILER_FPU_STATE is present, all settings from the command line options -M[no]daz, -M[no]flushz, or -Ktrap= are ignored and the FPU is initialized according to the options specified. NVCOMPILER_FPU_STATE with no options resets the floating-point control and status register to the system defaults.

The value of NVCOMPILER_FPU_STATE is a comma-separated list of options. The commands for setting the environment variable follow.

- In csh:

```
$ setenv NVCOMPILER_FPU_STATE option[,option...]
```

- In bash, sh, zsh, or ksh:

```
$ NVCOMPILER_FPU_STATE=option[,option...]
$ export NVCOMPILER_FPU_STATE
```

Table 31 lists the supported values for option.

By default, these options are taken from the compiler command line options -M[no]daz, -M[no]flushz, and -Ktrap=.

Table 31: Supported NVCOMPILER_FPU_STATE options

`fp`	Shorthand for inv,divz,ovf
`inv`	Raise exception on floating-point invalid operation (infinity - infinity, infinity / infinity, 0 / 0, ...)
`invalid`	Alias for inv
`denorm`	Raise exception with floating-point denormalized operands (x86_64 only)
`divz`	Raise exception on floating-point divide-by-zero
`zero`	Alias for divz
`ovf`	Raise exception on floating-point overflow in result
`over-flow`	Alias for ovf
`unf`	Raise exception on floating-point underflow in result
`under-flow`	Alias for unf
`inexact`	Raise exception on floating-point inexact result
`daz`	Convert denormal source operands to zero
`nodaz`	Do not convert denormal source operands to zero
`ftz`	Flush underflow results to zero
`flushz`	Alias for ftz
`noftz`	Do not flush underflow results to zero
`noflushz`	Alias for noftz
`print`	Print to stderr the state of floating point control and status register before and after processing of environment variable `NVCOMPILER_FPU_STATE`
`debug`	Alias for print

19.2.1.12 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 32 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 32: 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]trace-fp`	Enables/disables stack traceback and printing of SIMD registers (ymm/zmm) on error (Linux x86_64 only)
`[no]signal`	Enables/disables establishment of signal handlers for common signals that cause program termination
`[no]abort`	Enables/disables calling the system termination routine abort()

[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]trace-fp

This enables/disables stack traceback and printing of SIMD registers (ymm/zmm) on error. (Linux x86_64 only)

[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 csh:

```
$ 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*.

19.2.1.13 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:

... code:: text

    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.

19.2.1.14 PWD

The PWD variable allows you to display the current directory.

19.2.1.15 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.

19.2.1.16 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.

19.2.1.17 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.

19.3. HPC Compiler Related Environment Variables - CUDA Selection

The following table provides a listing of environment variables that let you override the default location of CUDA components (toolkit, math libraries, communication librarires, etc.) to point the compilers to system-installed or custom builds of the same components, when you do not want to use those those included the HPC SDK.

Table 33: NVIDIA HPC CUDA Components Slection Environment Variable Summary

Environment Variable	Description
NVHPC_CUDA_HOME or NVCOMPILER_CUDA_HOME	Set the path to find a CUDA Toolkit installation.
NVCOMPILER_MATH_LIBS_HOME	Set the path to find CUDA math libraries (e.g. cuBlas, cuFFT, cuSPARSE, etc.).
NVCOMPILER_COMM_LIBS_HOME	Set the path to find communication-libraries (e.g. NCCL, NVSHMEM).
NVCOMPILER_NCCL_HOME	Set the path to find NCCL libraries.
NVCOMPILER_SHMEM_HOME	Set the path to find NVSHMEM libraries.
NVCOMPILER_CUPTI_LIBS_HOME	Set the path to find CUPTI libraries.
NVCOMPILER_NSIGHT_COMPUTE_HOME	Set the path to find NSIGHT Compute.
NVCOMPILER_NSIGHT_SYSTEMS_HOME	Set the path to find NSIGHT Systems.
NVCOMPILER_COMPUTE_SANITIZER_HOME	Set the path to find Compute Sanitizer.

19.3.1. HPC Compilers Environment Variables - CUDA Selection

Use the environment variables listed in Table 33 to override the default location of CUDA components (toolkit, math libraries, communication librarires, etc.). This section provides more detailed descriptions about the variables in this table. Once any of these variables are set, the compiler driver will use them during the compilation process and at runtime when the program executes.

19.3.1.1 NVHPC_CUDA_HOME or NVCOMPILER_CUDA_HOME

The NVHPC_CUDA_HOME and NVCOMPILER_CUDA_HOME environment variables have the same behavior and tell the compilers where to find a CUDA Toolkit installation when you do not want to use the one included with the HPC SDK.

If the environment variable NVHPC_CUDA_HOME or NVCOMPILER_CUDA_HOME is present, the NVHPC SDK compiler will look under the path specified in this variable to find the CUDA toolkit installation and use it during the compilation process and at runtime during program execution.

By default both variables are unset and the CUDA Toolkit bundled with the HPC SDK is used.

The following example tells the compiler to use the CUDA Toolkit installed under the location specified in the variable:

- In csh:

```
$ setenv NVHPC_CUDA_HOME /opt/nvidia/hpc_sdk/25.9/cuda/11.8
```

- In bash, sh, zsh, or ksh:

```
$ NVHPC_CUDA_HOME=/opt/nvidia/hpc_sdk/25.9/cuda/11.8
$ export NVHPC_CUDA_HOME
```

19.3.1.2 NVCOMPILER_MATH_LIBS_HOME

NVCOMPILER_MATH_LIBS_HOME tells the compilers where to find CUDA math libraries (e.g. cuBlas, cuFFT, cuSPARSE, etc.) when you do not want to use those included with the HPC SDK.

The following example tells the compiler to use the CUDA math libraries installed under the location specified in the variable:

- In csh:

```
$ setenv NVCOMPILER_MATH_LIBS_HOME /opt/nvidia/hpc_sdk/25.9/math_libs/11.8
```

- In bash, sh, zsh, or ksh:

```
$ NVCOMPILER_MATH_LIBS_HOME=/opt/nvidia/hpc_sdk/25.9/math_libs/11.8
$ export NVCOMPILER_MATH_LIBS_HOME
```

19.3.1.3 NVCOMPILER_COMM_LIBS_HOME

NVCOMPILER_COMM_LIBS_HOME tells the compilers where to find CUDA communication libraries (e.g. NCCL, NVSHMEM) when you do not want to use those included with the HPC SDK.

The following example tells the compiler to use the CUDA communication libraries installed under the location specified in the variable:

- In csh:

```
$ setenv NVCOMPILER_COMM_LIBS_HOME /opt/nvidia/hpc_sdk/25.9/comm_libs/11.8
```

- In bash, sh, zsh, or ksh:

```
$ NVCOMPILER_COMM_LIBS_HOME=/opt/nvidia/hpc_sdk/25.9/comm_libs/11.8
$ export NVCOMPILER_COMM_LIBS_HOME
```

19.3.1.4 NVCOMPILER_NCCL_HOME

NVCOMPILER_NCCL_HOME tells the compilers where to find the NCCL libraries when you do not want to use those included with the HPC SDK.

The following example tells the compiler to use the NCCL libraries installed under the location specified in the variable:

- In csh:

```
$ setenv NVCOMPILER_NCCL_HOME /opt/nvidia/hpc_sdk/25.9/comm_libs/11.8/nccl
```

- In bash, sh, zsh, or ksh:

```
$ NVCOMPILER_NCCL_HOME=/opt/nvidia/hpc_sdk/25.9/comm_libs/11.8/nccl
$ export NVCOMPILER_NCCL_HOME
```

19.3.1.5 NVCOMPILER_SHMEM_HOME

NVCOMPILER_SHMEM_HOME tells the compilers where to find the NVSHMEM libraries when you do not want use those included with the HPC SDK.

The following example tells the compiler to use the NVSHMEM libraries installed under the location specified in the variable:

- In csh:

```
$ setenv NVCOMPILER_SHMEM_HOME /opt/nvidia/hpc_sdk/25.9/comm_libs/11.8/nvshmem
```

- In bash, sh, zsh, or ksh:

```
$ NVCOMPILER_SHMEM_HOME=/opt/nvidia/hpc_sdk/25.9/comm_libs/11.8/nvshmem
$ export NVCOMPILER_SHMEM_HOME
```

19.3.1.6 NVCOMPILER_CUPTI_LIBS_HOME

NVCOMPILER_CUPTI_HOME tells the compilers where to find the CUPTI libraries when you do not want to use those included with the HPC SDK.

The following example tells the compiler to use the CUPTI libraries installed under the location specified in the variable:

- In csh:

```
$ setenv NVCOMPILER_CUPTI_HOME /opt/nvidia/hpc_sdk/25.9/cuda/11.8/extras/CUPTI
```

- In bash, sh, zsh, or ksh:

```
$ NVCOMPILER_CUPTI_HOME=/opt/nvidia/hpc_sdk/25.9/cuda/11.8/extras/CUPTI
$ export NVCOMPILER_CUPTI_HOME
```

19.3.1.7 NVCOMPILER_NSIGHT_COMPUTE_HOME

NVCOMPILER_NSIGHT_COMPUTE_HOME tells the compilers where to find the Nsight Compute tools when you do not want to use those included with the HPC SDK.

The following example tells the compiler to use the Nsight Compute tool installed under the location specified in the variable:

- In csh:

```
$ setenv NVCOMPILER_NSIGHT_COMPUTE_HOME /opt/nvidia/hpc_sdk/25.9/profilers/Nsight_
↪Compute
```

- In bash, sh, zsh, or ksh:

```
$ NVCOMPILER_NSIGHT_COMPUTE_HOME=/opt/nvidia/hpc_sdk/25.9/profilers/Nsight_Compute
$ export NVCOMPILER_NSIGHT_COMPUTE_HOME
```

19.3.1.8 NVCOMPILER_NSIGHT_SYSTEMS_HOME

NVCOMPILER_NSIGHT_SYSTEMS_HOME tells the compilers where to find the Nsight Systems tools when you do not want to use those included with the HPC SDK.

The following example tells the compiler to use the Nsight Systems tool installed under the location specified in the variable:

- In csh:

```
$ setenv NVCOMPILER_NSIGHT_SYSTEMS_HOME /opt/nvidia/hpc_sdk/25.9/profilers/Nsight_
↪Systems
```

- In bash, sh, zsh, or ksh:

```
$ NVCOMPILER_NSIGHT_SYSTEMS_HOME=/opt/nvidia/hpc_sdk/25.9/profilers/Nsight_Systems
$ export NVCOMPILER_NSIGHT_SYSTEMS_HOME
```

19.3.1.9 NVCOMPILER_COMPUTE_SANITIZER_HOME

NVCOMPILER_COMPUTE_SANITIZER_HOME tells the compilers where to find the compute-sanitizer tool when you do not want to use the one included with the HPC SDK.

The following example tells the compiler to use the compute-sanitizer tool installed under the location specified in the variable:

- In csh:

```
$ setenv NVCOMPILER_COMPUTE_SANITIZER_HOME /opt/nvidia/hpc_sdk/25.9/cuda/11.8/
↪compute-sanitizer
```

- In bash, sh, zsh, or ksh:

```
$ NVCOMPILER_COMPUTE_SANITIZER_HOME=/opt/nvidia/hpc_sdk/25.9/cuda/11.8/compute-
↪sanitizer
$ export NVCOMPILER_COMPUTE_SANITIZER_HOME
```

19.4. 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. The install script will generate environment module files for you as part of the set up process.

Assuming your installation base directory is `/opt/nvidia/hpc_sdk`, the environment modules will be installed under `/opt/nvidia/hpc_sdk/modulefiles`. There will be three sets of module files:

1. nvhpc

   Adds environment variable settings for the NVIDIA HPC Compilers, CUDA libraries, and additional libraries such as MPI, NCCL, and NVSHMEM.

2. nvhpc-nompi

   Adds environment variable settings for the NVIDIA HPC Compilers, CUDA libraries, and additional libraries such as NCCL and NVSHMEM. This will not include MPI, if you wish to use an alternate MPI implementation.

3. nvhpc-byo-compilers

   Adds environment variable settings for the CUDA libraries and additional libraries such as NCCL and NVSHMEM. This will not include the NVIDIA HPC Compilers nor MPI, if you wish to use alternate compilers and MPI.

You can load the nvhpc environment module for the 20.11 release as follows:

```
$ module load nvhpc/26.11
```

To see what versions of nvhpc are available on this system, use this command:

```
$ module avail nvhpc
```

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 nvc (nvhpc and nvhpc-nompi only)
CPATH	Prepends the math libraries include directory, the MPI include directory (nvhpc only), and NCCL and NVSHMEM include directories
CPP	C preprocessor, normally cpp (nvhpc and nvhpc-nompi only)
CXX	Path to nvc++ (nvhpc and nvhpc-nompi only)
FC	Full path to nvfortran (nvhpc and nvhpc-nompi only)
F90	Full path to nvfortran (nvhpc and nvhpc-nompi only)
F77	Full path to nvfortran (nvhpc and nvhpc-nompi only)
LD_LIBRARY_PATH	Prepends the CUDA library directory, the NVIDIA HPC Compilers library directory (nvhpc and nvhpc-nompi only), math libraries library directory, MPI library directory (nvhpc only), and NCCL and NVSHMEM library directories
MANPATH	Prepends the NVIDIA HPC Compilers man page directory (nvhpc and nvhpc-nompi only)
OPAL_PREFIX	Full path to the MPI directory (nvhpc only), e.g. /opt/nvidia/hpc_sdk/Linux_x86_64/26.11/comm_libs/mpi
PATH	Prepends the CUDA bin directory, the MPI bin directory (nvhpc only), and the NVIDIA HPC Compilers bin directory (nvhpc and nvhpc-nompi only)

Note: NVIDIA does not provide support for the Environment Modules package. For more information about the package, go to: http://modules.sourceforge.net.

19.5. 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 20. 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.

20.1. 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

20.1.1. 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.

Note: 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
```

20.1.2. 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.

Note: If your application is linked dynamically using shared objects, then the shared object versions of the NVIDIA HPC Compilers runtime are required.

20.1.3. 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.

20.1.4. 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.

20.1.5. 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 21. 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. Additional interoperability with C is available with Fortran 2018 and the `ISO_Fortran_binding.h` C header file. nvfortran supports both the `iso_c_binding` and the `ISO_Fortan_Binding.h` header file. In the absence of these mechanisms, 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`

21.1. 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.

21.2. 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.

21.3. 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 34, Table 35, 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.

21.4. 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 `bind(c)` in the Fortran program.
- If you call a Fortran function from C/C++, you should append an underscore to the Fortran function name in the calling program.

21.5. Compatible Data Types

Table 34 shows compatible data types between Fortran and C/C++. Table 35 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 34: 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
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 35: 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;	8
	float complex x;	8
complex*8 x	struct {float r,i;} x;	8
	float complex x;	8
double complex x	struct {double dr,di;} x;	16
	double complex x;	16
complex *16 x	struct {double dr,di;} x;	16
	double complex x;	16

Note: For C/C++, the `complex` type implies C99 or later.

21.5.1. 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.

21.6. 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 Linux systems, the compiler places the length argument(s) at the end of the parameter list, following the other formal arguments.

The length argument is passed by value, not by reference.

21.6.1. 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))
```

21.6.2. 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.

Note: The value of the character function is not automatically NULL-terminated.

21.6.3. 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);
```

21.7. 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 arrqays 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.

21.8. Examples

This section contains examples that illustrate inter-language calling.

21.8.1. Example – Fortran Calling C

Note: 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
```

21.8.2. Example - C Calling Fortran

Note: There are other solutions to calling Fortran from C than the one presented in this section. For example, you can use the `ISO_Fortran_binding.h` C header file which NVIDIA does support. For more information on this header file and for examples of how to use it, search the web using the keyword ISO_Fortran_binding.

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 numdoub1;
 short numshor1;
 extern void c2f_func_();
 c2f_sub_(&bool1,&letter1,&numint1,&numint2,&numfloat1,&numdoub1,&numshor1, 1);
 printf(" %s %c %d %d %3.1f %.0f %d\n",
 bool1?"TRUE":"FALSE", letter1, numint1, numint2,
 numfloat1, numdoub1, numshor1);
}
```

Fortran Subroutine c2f_sub.f

```
subroutine c2f_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"
```

(continues on next page)

(continued from previous page)

```
    numint1 = 11
    numint2 = -44
    numdoub1 = 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
```

21.8.3. 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
```

21.8.4. 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
```

Note: You cannot use the extern “C” form of declaration for an object’s member functions.

21.8.5. 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 numdoub1
  integer*2 numshor1
external f2cpfunc
call f2cp_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 f2cp_func.C

```
#define TRUE 0xff
#define FALSE 0
extern "C"
{
extern void f2cp_func_ (
 char *bool1, *letter1,
 int *numint1, *numint2,
 float *numfloat1,
 double *numdoub1,
 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
```

21.8.6. 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
```

Note: You must explicitly link in the NVFORTRAN runtime support libraries when linking nvfortran-compiled program units into C++ or C main programs.

Chapter 22. Programming Considerations for 64-Bit Environments

NVIDIA provides 64-bit compilers for 64-bit Linux operating systems running on x86-64 (Linux_x86_64) 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.

Note: 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.

- `-fPIC`
- `-mcmodel=medium`
- `-Mlarge_arrays`
- `-i8`
- `-Mlargeaddressaware`

22.1. 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.

22.1.1. 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.

22.1.2. 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`.

22.2. 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. 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.

Linux_aarch64 does not support -mcmodel=medium. If the medium model is specified on the command-line, the compiler driver will automatically select the large model.

22.3. 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.

22.4. 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.

Note: 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`.

22.5. 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 36: 64-bit Compiler Options

Option	Purpose	Considerations
`-mcmodel=medium`	Allow for data declarations larger than 2GB.	Linux_aarch64 does not support -mcmodel=medium. If the medium model is specified on the command-line, the compiler driver will automatically select the large model.
`-Mlarge_arrays`	Perform all array-location-to-address calculations using 64-bit integer arithmetic.	Slightly slower execution. Is implicit with `-mcmodel=medium`. Can be used with option `-mcmodel=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 37: 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 `-mcmodel=small`	64	32	2	2	2
-fpic *incompatible with* `-mcmodel=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.

22.6. 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 38: 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:
	`limit stacksize new_size ! in csh`
	`ulimit -s new_size ! in bash`

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`. **Note:** Without the 64-bit offset support in the object file format, large arrays cannot be declared statically, or locally on the stack.

22.7. 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
```

Note: 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
```

22.8. 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
```

22.9. 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
  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 23. C++ and C Inline Assembly and Intrinsics

The examples in this section are shown using x86-64 assembly instructions. Inline assembly is supported on Arm Server platforms as well, but is not documented in detail in this section.

23.1. 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 it 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");
```

23.2. 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.

23.2.1. 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:
..EN1:
## 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
```

(continues on next page)

(continued from previous page)

```
..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.

23.2.2. 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.

23.2.3. 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 noinline
int asmDivideConquer(int n)
{
 int ax = 0;
 int bx = 1;
 asm (
 "LogLoop:n"
 "cmp %2, %1n"
 "jnle Donen"
 "inc %0n"
 "add %1,%1n"
 "jmp LogLoopn"
 "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 noinline 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.

23.2.4. Additional Constraints

Operand constraints can be divided into four main categories:

- Simple Constraints
- Machine Constraints
- Multiple Alternative Constraints
- Constraint Modifiers

23.2.5. 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 39 describes these constraints.

Table 39: 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”.
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
```

(continues on next page)

(continued from previous page)

```
 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().

23.2.6. 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 40.

Table 40: 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 0to 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;"
```

(continues on next page)

(continued from previous page)

```
 :"=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.

23.2.7. 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 41: 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.

23.2.8. 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 42: 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 *Multiple Alternative Constraints*, 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.

23.3. 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.

23.4. 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 43 summarizes these modifiers, known as *Assembly String Modifiers*.

Table 43: 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, %S0 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	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.

23.5. 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.

23.6. 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 44: Intrinsic Header File Organization

Instructions	Header File		Instructions	Header File
ABM	intrin.h		SSE2	emmintrin.h
AVX	immintrin.h		SSE3	pmmintrin.h
MMX	mmintrin.h		SSSE3	tmmintrin.h
SSE	xmmintrin.h		SSE4a	ammintrin.h

The following is a simple example program that calls XMM intrinsics.

```
#include <xmmintrin.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;
 }
```

Notices

Notice and Disclaimers

All information provided in this document is provided as-is, for your informational purposes only and is subject to change at any time without notice. Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing. To obtain the latest information, please contact your NVIDIA representative. Product or service performance varies by use, configuration and other factors. Your costs and results may vary. No product or component is absolutely secure. TO THE FULLEST EXTENT PERMITTED BY APPLICABLE LAW, NVIDIA DISCLAIMS ALL WARRANTIES AND REPRESENTATIONS OF ANY KIND, WHETHER EXPRESS, IMPLIED OR STATUTORY, RELATING TO OR ARISING UNDER THIS DOCUMENT, INCLUDING, WITHOUT LIMITATION, THE WARRANTIES OF TITLE, NONINFRINGEMENT, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, USAGE OF TRADE AND COURSE OF DEALING. NVIDIA products are not intended or authorized for use as critical components in a system or application where the use of or failure of such system or application developed with products, technology, software or services provided by NVIDIA could result in injury, death or catastrophic damage.

Except for your permitted use of the information contained in this document, no license or right is granted by implication, estoppel or otherwise. If this document directly includes or links to third-party websites, products, services or information, please consult other sources to evaluate if and how to use that information since NVIDIA does not support, endorse or assume any responsibility for any third party offerings or its accuracy or usefulness.

TO THE FULLEST EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY (I) INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, OR (II) DAMAGES FOR THE (A) COST OF PROCURING SUBSTITUTE GOODS OR (B) LOSS OF PROFITS, REVENUES, USE, DATA OR GOODWILL ARISING OUT OF OR RELATED TO THIS DOCUMENT, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY, OR OTHERWISE, AND EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES AND EVEN IF A PARTY'S REMEDIES FAIL THEIR ESSENTIAL PURPOSE. ADDITIONALLY, TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, NVIDIA'S TOTAL CUMULATIVE AGGREGATE LIABILITY FOR ANY AND ALL LIABILITIES, OBLIGATIONS OR CLAIMS ARISING OUT OF OR RELATED TO THIS DOCUMENT WILL NOT EXCEED FIVE U.S. DOLLARS (US$5).

Statements in this document that refer to future plans or expectations are forward-looking statements. These statements are based on currently available information, beliefs, assumptions and involve many risks and uncertainties that could cause actual results to differ materially from those expressed or implied in these statements. For more information on the factors that could cause actual results to differ materially, see our most recent earnings release and SEC filings at NVIDIA Corporation SEC Filings.

© NVIDIA Corporation. All rights reserved. NVIDIA, the NVIDIA logo, and other NVIDIA marks are trademarks of NVIDIA Corporation or its affiliates. Other names and brands may be claimed as the property of others.

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

©2013-2026, NVIDIA Corporation & affiliates. All rights reserved