The AMD Optimizing C/C++ and Fortran Compiler (AOCC) is highly optimized for x86 targets, especially for AMD Zen architecture-based processors.

AOCC leverages LLVM, a project that includes a collection of modular and reusable compiler and toolchain technologies. With LLVM, converting a source code to a machine code (object code) for a specific processor involves three phases:

Frontend phase
: In this phase, the frontend parses, validates, and diagnoses errors in the source code, and converts the parsed code into the LLVM language-independent intermediate representation (LLVM IR)
One option that can be used as a C/C++ language compiler frontend is Clang, which is a project under the LLVM umbrella. Similarly, Flang is a Fortran language frontend integrated with LLVM.

Optimizer phase
: The IR is transformed into an optimized IR by going through a series of optimization passes (e.g., inlining, unrolling, dead code elimination, and so on.)

Backend phase
: The optimized IR is transformed into a target-specific machine code.

AOCC is a proprietary fork of LLVM+Clang, with changes to improve performance for processors based on the AMD Zen microarchitectures. For example, AOCC version 4.2 is based on LLVM version 16.0.3.

AOCC 4.2 supports the following standards:

- **C:** C17 (default), C11, C99.
- **C++:** C++20, C++17 (default), C++14, C++11, C++98.
- **Fortran:** Fortran-95, Fortran-2003, partial support of Fortran-2008.
- **OpenMP:**
  - **For C and C++:** OpenMP 5.0 and earlier standards.
  - **For Fortran:** OpenMP 4.5.

AOCC Usage

Multiple versions of AOCC are installed on NAS systems. We recommend using the latest version.

Load an AOCC Modulefile

After one of the modules is loaded, the environment variables PATH, LIBRARY_PATH, LD_LIBRARY_PATH, C_INCLUDE_PATH, and CPLUS_INCLUDE_PATH are modified.

You can load an AOCC modulefile as follows:

```
% module use -a /nasa/modulefiles/testing
% module avail comp-aocc
--- /nasa/modulefiles/toss4 ---
comp-aocc/3.2.0
--- /nasa/modulefiles/testing ---
comp-aocc/4.0.0  comp-aocc/4.1.0  comp-aocc/4.2.0
% module load comp-aocc/4.2.0
```

Compile and Link using the Clang, Clang++, or Flang Compiler Driver

Use one of the following command lines:

```
clang   [command-line options] program.c
clang++ [command-line options] program.cpp
flang   [command-line options] program.f90
```

A few command-line options are provided here:

- **-O3:** provides more optimization than the -O3 in the base LLVM version.
  Note: -O2 is the default in AOCC 4.2.
- **-Ofast: -**O3 plus other aggressive optimizations that may violate strict compliance with language standards.
  Note: -Ofast in AOCC has more optimization than -Ofast in the base LLVM version.
- **-march=znver3:** generates instructions that target AMD Zen 3 processor (such as Milan).
- **-ffast-math:** enables faster but less-precise math operations.
- **-ffp-model={precise,fast}:** is only applicable to Flang; "precise" disables optimizations that are not value-safe on floating-point data.
- **-fopenmp:** parses OpenMP programs/directives and generates parallel code.
- **-flto:** provides whole program analysis with Link Time Optimization (LTO).
- **-fuse-ld=lld:** uses the LLD linker (default).
- **-fgen-aor:** generates compiler-time AOCC optimization report (AOR); for link-time AOR generation, add -flto -fuse-ld=lld.
  Note: The generation of AOR requires python3. With AOCC 4.2, the script run_opt-viewer that is used to generate the report requires python3.8.

You can find a list of command-line options on the Clang help page (clang --help) or in Chapter 5: Command-line Options in the AMD AOCC User Guide. Additional options can be found in Compiler Options Quick Reference Guide for Zen 3 and Zen 4 processors.

Link to an AMD AOCL Library

Some applications might perform better on the AMD Zen processors when they use AOCL, rather than using other scientific and math libraries from other sources. For more information, see AMD Optimizing CPU Libraries (AOCL).

Link to an MPI Library

HPE MPT

HPE MPT provides the wrapper scripts mpicc, mpicxx, and mpif90 for compiling and linking MPI applications. Each of the scripts detects which compilers are available in your environment, and uses the first one it finds when checking against a list of compilers included in the script.

With the module mpi-hpe/mpt.2.2825Apr23_rhe187, the list of precedence for the mpif90 script is {ifort openf90 pgf90 pgf95 lf95 gfortran}. For mpicc and mpicxx scripts, the only choice in the list is gcc and g++.

In order to use an AOCC compiler with these wrapper scripts, you must set some environment variables, as shown below. You can use the option -show of these wrapper scripts to find out which compiler is chosen.

```
% module use -a /nasa/modulefiles/testing
% module load comp-aocc/4.2.0
% module load mpi-hpe/mpt
```

- C code

  ```
  % export MPICC_CC=clang
  % mpicc -show
  clang -I/nasa/hpe/mpt/..../include -lpthreads -L/nasa/hpe/mpt/... -lmpi
  % mpicc [options] program.c
  ```

- C++ code

  ```
  % export MPICXX_CXX=clang++
  % mpicxx -show
  clang++ -I/nasa/hpe/mpt/..../include -lpthreads -L/nasa/hpe/mpt/... -lmpi
  % mpicxx [options] program.cc
  ```

- Fortran code

  ```
  % export MPIF90_F90=flang
  % mpif90 -show
  flang -I/nasa/hpe/mpt/..../include -lpthreads -L/nasa/hpe/mpt/... -lmpi
  % mpif90 [options] program.f
  ```

OpenMPI

OpenMPI provides the wrapper compilers mpicc, mpic++ (or mpicxx, mpiCC), and mpifort for compiling and linking MPI applications. The legacy wrappers mpif77 and mpif90 still exist, but are symbolic links to mpifort. The compilers used by these wrappers default to the compilers used to build the OpenMPI library. After loading an OpenMPI module, you can run one of these commands to find the compilers used: ompi_info, mpicc --showme, mpic++ --showme, or mpifort --showme .

When building your application with an AOCC compiler and an OpenMPI library, the OpenMPI library that you use should be one that is also built with the AOCC compiler. This is especially true if your application is written in Fortran.

For applications written in C, you may be able to compile with an AOCC compiler and link with an OpenMPI library built with a different compiler, as shown below:

```
% module use -a /nasa/modulefiles/testing
% module load comp-aocc/4.2.0
% module load openmpi/4.1.6-toss4-gnu

% export OMPI_CC=clang
% mpicc --showme
clang -I/nasa/openmpi/4.1.6-gnu/include -pthread -L/nasa/openmpi/4.1.6-gnu/lib -Wl,
      -rpath -Wl,/nasa/openmpi/4.1.6-gnu/lib -Wl,--enable-new-dtags -lmpi
% mpicc [options] program.c
```

References

- The Architecture of Open Source Applications (Volume 1) LLVM
- AMD Optimizing C/C++ and Fortran Compilers (AOCC)
- AMD AOCC User Guide
- AMD EPYC 9xx4-series Processors Compiler Options Quick Reference Guide
- AMD EPYC 7xx3-series Processors Compiler Option Quick Reference Guide

Intel Fortran expects numeric data—both integer and floating-point data—to be in native little-endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte).

If your program needs to read or write unformatted data files that are not in little-endian order, you can use one of the following six methods, which are provided by Intel. The methods are listed in the order of precedence.

Method 1. Setting a Variable for a Specific Unit Number

Set an environment variable for a specific unit number before the file is opened. The environment variable is named FORT_CONVERT*n*, where *n* is the unit number. For example:

setenv FORT_CONVERT28 BIG_ENDIAN

No source code modification or recompilation is needed.

Method 2. Setting A Variable for a Specific File Name

Set an environment variable for a specific file name extension before the file is opened. The environment variable is named FORT_CONVERT.ext or FORT_CONVERT_ext, where "ext" is the file name extension (suffix). The following example specifies that a file with an extension of ".dat" is in big-endian format:

setenv FORT_CONVERT.DAT BIG_ENDIAN

Some Linux command shells may not accept a dot (.) for environment variable names. In that case, use FORT_CONVERT_ext instead.

No source code modification or recompilation is needed.

Method 3. Setting a Variable for a Set of Units

Set an environment variable for a set of units before any files are opened. The environment variable is named F_UFMTENDIAN.

Syntax

Csh: setenv F_UFMTENDIAN MODE;EXCEPTION

Sh : export F_UFMTENDIAN=MODE;EXCEPTION

MODE = big | little

EXCEPTION = big:ULIST | little:ULIST | ULIST

ULIST = U | ULIST,U

U = decimal | decimal-decimal

MODE defines the current format of the data, represented in the files; it can be omitted. The keyword "little" means that the data has little-endian format and will not be converted. For IA-32 systems, this keyword is a default. The keyword "big" means that the data has big-endian format and will be converted. This keyword may be omitted together with the colon.

EXCEPTION is intended to define the list of exclusions for MODE; it can be omitted. EXCEPTION keyword ("little" or "big") defines data format in the files that are connected to the units from the EXCEPTION list. This value overrides MODE value for the units listed.

Each list member U is a simple unit number or a number of units. The number of list members is limited to 64. decimal is a non-negative decimal number less than 2**32.

The environment variable value should be enclosed in quotes if the semicolon is present.

Converted data should have basic data types, or arrays of basic data types. Derived data types are disabled.

Examples

setenv F_UFMTENDIAN big

All input/output operations perform conversion from big-endian to little-endian order on READ, and from little-endian to big-endian order on WRITE.

setenv F_UFMTENDIAN "little;big:10,20"

or setenv F_UFMTENDIAN big:10,20

or setenv F_UFMTENDIAN 10,20

In this case, only on unit numbers 10 and 20 the input/output operations perform big-little endian conversion.

```
setenv F_UFMTENDIAN "big;little:8"
```

In this case, on unit number 8 no conversion operation occurs. On all other units, the input/output operations perform big-little endian conversion.

```
setenv F_UFMTENDIAN 10-20
```

Define 10, 11, 12, ...19, 20 units for conversion purposes; on these units, the input/output operations perform big-little endian conversion.

Method 4. Using the CONVERT Keyword in the OPEN Statement

Specify the CONVERT keyword in the OPEN statement for a specific unit number. Note that a hard-coded OPEN statement CONVERT keyword value cannot be changed after compile time. The following OPEN statement specifies that the file graph3.dat is in VAXD unformatted format:

```
OPEN (CONVERT='VAXD', FILE='graph3.dat', FORM='UNFORMATTED', UNIT=15)
```

Method 5. Compiling with an OPTIONS Statement

Compile the program with an OPTIONS statement that specifies the CONVERT=*keyword* qualifier. This method affects all unit numbers using unformatted data specified by the program. For example, to use VAX F_floating and G_floating as the unformatted file format, specify the following OPTIONS statement:

```
OPTIONS /CONVERT=VAXG
```

Method 6. Compiling with the -convert keyword Option

Compile the program with the command-line -convert keyword option, which affects all unit numbers that use unformatted data specified by the program. For example, the following command line compiles program file.for to use VAXD floating-point data for all unit numbers:

```
ifort file.for -o vconvert.exe -convert vaxd
```

In addition, if the record length of your unformatted data is in byte units (Intel Fortran default is in word units), use the-assume byterecl compiler option when compiling your source code.

The GNU Compiler Collection (GCC) is an integrated distribution of compilers for several major programming languages, which currently include C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada.

The GNU C and C++ compiler (`gcc` and `g++`) and Fortran compiler (`gfortran`) are available on NAS systems through the Linux operating system distribution. The current version, which is installed in the `/usr/bin` directory, can be found by using the `gcc -v` command. For example:

```
% gcc -v
[...]
4.8.5 20150623 (Red Hat 4.8.5-44) (GCC)
```

To see which GCC versions are available as modules on NAS systems, run `module avail`.

Read **man gcc** and **man gfortran** for more information.

OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for various platforms.

Building OpenMP Applications

The following Intel compiler options can be used for building or analyzing OpenMP applications:

- **-qopenmp**

  Enables the parallelizer to generate multithreaded code based on OpenMP directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems. The -openmp option works with both *-O0* (no optimization) and any optimization level of -O. Specifying -O*0* with -openmp helps to debug OpenMP applications.

- **-assume cc_omp** or **-assume nocc_omp**

  -assume cc_omp enables conditional compilation as defined by the OpenMP Fortran API. That is, when "!$space" appears in free-form source or "c$spaces" appears in column 1 of fixed-form source, the rest of the line is accepted as a Fortran line.

  -assume nocc_omp tells the compiler that conditional compilation as defined by the OpenMP Fortran API is disabled unless option -openmp (Linux) or /Qopenmp (Windows) is specified.

- **-openmp-link dynamic** or **-openmp-link static**

  Choosing -openmp-link dynamic tells the compiler to link to dynamic OpenMP run-time libraries.

  Choosing -openmp-link static tells the compiler to link to static OpenMP run-time libraries.

  Note that the compiler options -static-intel and -shared-intel have no effect on which OpenMP run-time library is linked.

- **-qopenmp-threadprivate=legacy or -qopenmp-threadprivate=compat**

  Choose -qopenmp-threadprivate=legacy tells the compiler to use the legacy OpenMP threadprivate implementation used in the previous releases of the Intel(R) compiler. This setting does not provide compatibility with the implementation used by other compilers.

  Choose -qopenmp-threadprivate=compat tells the compiler to use the compatibility OpenMP thread private implementation based on applying the thread-local attribute to each threadprivate variable. This setting provides compatibility with the implementation provided by the Microsoft and GNU compilers.

- **-qopenmp-stubs**

  Enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.

- **-qopt-report-phase=openmp**

  This option generates optimization reports for the OpenMP phase.

OpenMP Environment Variables

There are a few OpenMP environment variables one can set. The most commonly used are:

- **OMP_NUM_THREADS *num***

  Sets number of threads for parallel regions. Default is 1 for PBS jobd on NAS systems. Note that you can use ompthreads in the PBS resource request to set values for OMP_NUM_THREADS. For example:

  ```
  %qsub -I -lselect=1:ncpus=4:ompthreads=4
  Job 991014.pbspl1.nas.nasa.gov started on Sun Sep 12 11:33:06 PDT 2010
  ...
  PBS r3i2n9> echo $OMP_NUM_THREADS
  4
  PBS r3i2n9>
  ```

- **OMP_SCHEDULE *type*[,*chunk*]**

  Sets the run-time schedule type and chunk size. Valid OpenMP schedule types are static, dynamic, guided, or auto. Chunk is a positive integer.

- **OMP_DYNAMIC *true*** or **OMP_DYNAMIC *false***

  Enables or disables dynamic adjustment of threads to use for parallel regions.

- **OMP_STACKSIZE *size***

  Specifies size of stack for threads created by the OpenMP implementation. Valid values for size (a positive integer) are *size,*

*size*B, *size*K, *size*M, *size*G. If units B, K, M or G are not specified, size is measured in kilobytes (K).

Note that Intel also provides a few additional environment variables. The most commonly used are:

- **KMP_AFFINITY *type***

  Binds OpenMP threads to physical processors. Available *type*: compact, disabled, explicit, none, and scatter.

  WARNING: There is a conflict between KMP_AFFINITY in Intel 11.*x* runtime library and dplace, causing all threads to be placed on a single CPU when both are used. It is recommended that KMP_AFFINITY be set to *disabled* when using dplace.
- **KMP_MONITOR_STACKSIZE**

  Sets stack size in bytes for monitor thread.

- **KMP_STACKSIZE**

  Sets stack size in bytes for each thread.

For more information, please see the official OpenMP web site.

As an alternative to using Intel compilers for Fortran, C, and C++ programs, you can use the compilers and program development tools from PGI Compilers and Tools. By default, the PGI compilers generate code that is optimized for the compilation host.

Note: The PGI compilers have been rebranded by NVIDIA. This article will be removed in the near future.

PGI Compiling

Several versions of PGI compilers and tools are installed as modules under the /nasa system directory. To view available versions, run:

```
pfe21% module use -a /nasa/modulefiles/testing
pfe21% module avail comp-pgi

-------- /nasa/modulefiles/testing ---------
comp-pgi/17.10 comp-pgi/18.10 comp-pgi/18.4  comp-pgi/19.10 comp-pgi/19.5  comp-pgi/20.4
```

The following command line will load version 18.10:

```
pfe21% module load comp-pgi/18.10
```

Note: Starting with the PGI compiler 2018, two different code generators (PGI-proprietary vs LLVM-based) are included in each package. The default code generator for 2018 versions is the PGI-proprietary code generator and for 2019 and later versions is the LLVM-based generator. The PGI 2019 compilers using LLVM are not link-compatible with PGI 2018 and prior releases.

Using the PGI Compilers

PGI provides multiple commands for different languages or functions. In 18.10 and earlier versions, the following are available.

Command	Language or Function
pgfortran	PGI Fortran
pgf77	Fortran 77
pgf90 or pgf95	Fortran 90/95/F2003
pgcc	ANSI C99 and K&R (Kernighan and Ritchie) C
pgc++	GNU-compatible C++
pgdbg	Source code debugger (supports OpenMP and MPI)
pgprof	Performance profiler (supports OpenMP and MPI)

Note: By default, the PGI compilers generate code that is optimized for the type of processor on which compilation is performed (the compilation host). Be aware that the processors are forward-compatible, but not backward-compatible. Thus a code compiled and optimized on a newer-generation processor will not necessarily execute correctly on previous-generation processors.

If you want to build an executable that targets a specific processor type, use the -tp flag:

-tp= skylake-64	Intel Skylake Xeon processor, 64-bit mode (includes Skylake and Cascade Lake)
-tp= haswell-64	Intel Haswell architecture Core processor, 64-bit mode (includes Haswell and Broadwell)
-tp= sandybridge-64	Intel SandyBridge architecture Core processor, 64-bit mode (includes Sandy Bridge and Ivy Bridge)

TIP: Using the -tp=sandybridge-64,haswell-64,skylake-64 option will generate a single executable in which the code is optimized for the Intel Sandy Bridge, Haswell, and Skylake architectures. The choice of which optimized copy to execute is made at run time and depends on the machine executing the code. You must load comp-pgi/18.4 or later versions in order to use -tp= skylake-64.

PGI recommends that for best performance on processors which support SSE instructions (including all NAS processor types), use pgfortran, even for FORTRAN 77 code, use the -fastsse option.

For more information about the PGI compilers, see **man pgfortran, man pgcc,** and **man pgc++** or use the pgfortran -help, pgcc -help, or pgc++ -help commands. Information about the PGI debugger and performance analysis tool can be found using the commands pgdbg -help and pgprof -help.

Intel compilers are recommended for building your applications on HECC systems with Intel x86-64 CPUs. There is no default compiler—you must load a specific module.

Note: The Intel classic compilers are no longer available in newer versions of the Intel OneAPI toolkits distributions.

To see what modules are available, run the module avail command:

```
pfe21% module avail
```

In the output, look for the comp-intel/*versions* entries. Then, load the Intel compiler module you want to use. For example:

```
pfe21% module load comp-intel/2020.4.304
```

When you load a compiler module, some environment variables (such as PATH, CPATH, and LD_LIBRARY_PATH) are set or modified to add the paths to certain commands, include files, or libraries to your environment. This helps to simplify the way you do your work.

You can use the module show command check what environment variables will be modified for a particular module. For example:

```
pfe21% module show comp-intel/2020.4.304
```

Intel Compilers for HECC Systems

On HECC systems, there are Intel compilers for both Fortran and C/C++:

Intel Fortran Compiler: ifort (version 8 and above)

The ifort command invokes the Intel Fortran Compiler to preprocess, compile, assemble, and link Fortran programs.

```
pfe21% ifort [options] file1 [file2 ...]
```

Read **man ifort** for all available compiler options. To see the compiler options by categories, run:

```
pfe21% ifort -help
```

file*N* is a Fortran source (.f .for .ftn .f90 .fpp .F .FOR .FTN .FPP .F90 .i .i90), assembly (.s .S), object (.o), static library (.a), or other linkable file. Interpret the suffixes of the Fortran source files as follows:

- .f, .for, or .ftn = Fixed-form source files.
- .f90 = Free-form F95/F90 source files.
- .fpp, .F, .FOR, .FTN, or .FPP = Fixed-form source files that must be pre-processed by the FPP pre-processor before being compiled.
- .F90: = Free-form Fortran source files that must be pre-processed by the FPP pre-processor before being compiled.

Intel C/C++ Compiler: icc and icpc Commands (version 8 and above)

The Intel C++ Compiler is designed to process C and C++ programs on Intel-architecture-based systems. You can preprocess, compile, assemble, and link these programs.

```
pfe21% icc [options] file1 [file2 ...]
% icpc [options] file1 [file2 ...]
```

Read **man icc** for all available compiler options. To see the compiler options by categories, run:

```
pfe21% icc -help
```

The icpc command uses the same compiler options as the icc command, with the following differences:

- Invoking the compiler using icpc compiles .c and .i files as C++, and always links in C++ libraries.
- Invoking the compiler using icc compiles .c and .i files as C, and only links in C++ libraries if C++ source is provided on the command line.

file*N* represents a C/C++ source (.C .c .cc .cp .cpp .cxx .c++ .i), assembly (.s), object (.o), static library (.a), or other linkable file.

In late 2019, Intel migrated away from its Parallel Studio Cluster Edition and introduced its oneAPI initiative to enable application developers to use a unified programmng model that supports CPUs, GPUs, FPGAs (Field Programmable Gate Arrays), and other accelerators. The oneAPI Toolkits include compilers, libraries, analyzers, debuggers, AI tools, and so on.

Starting in 2021, Intel provided two sets of compilers in its oneAPI distribution:

- Intel C++ Compiler Classic (icc for C, icpc for C++) and Intel Fortran Compiler Classic (ifort)
- LLVM-based Intel oneAPI Data Parallel C++ (DPC++)/C++ Compiler (icx for C, icpx for C++, dpcpp for C++ with SYCL extensions), and Intel Fortran Compiler (ifx)

  Note: SYCL is an open standard developed by the Khronos Group that allows developers to program heterogeneous architectures in standard C++.

Intel Classic Compilers

The Intel Classic icc and icpc compilers have been discontinued starting with the oneAPI 2024.0 release. The Intel Classic ifort compiler has been discontinued starting with the oneAPI 2025.0 release. Only the LLVM-based compilers (icx, icpx, dpcpp, and ifx) are available in new oneAPI distributions.

To continue to access icc, icpc, and ifort, load an existing intel-comp module such as comp-intel/2023.2.1 or older. For example:

```
% module use -a /nasa/modulefiles/testing
% module load comp-intel/2023.2.1
% which icc
/nasa/intel/oneapi/compiler/2023.2.1/linux/bin/intel64/icc
% which icx
/nasa/intel/oneapi/compiler/2023.2.1/linux/bin/icx
% which ifort
/nasa/intel/oneapi/compiler/2023.2.1/linux/bin/intel64/ifort
% which ifx
/nasa/intel/oneapi/compiler/2023.2.1/linux/bin/ifx
```

Read the Intel documentation listed in the Reference section for information on how to transition from the Intel Classic Compilers to the LLVM-based Intel Compilers.

What is LLVM?

LLVM is a compiler infrastructure, provided by the LLVM Compiler Infrastructure Project, that includes a collection of modular and reusable compiler and toolchain technologies. LLVM provides multiple benefits to compiler developers. For instance, due to its modular nature, you can choose which components and optimizations to include in the compiler. You can also insert your own optimization into the pipeline.

In addition, LLVM is publicly available under an open source license. You can therefore benefit from the large suite of advanced optimizations collected by the LLVM Project. In fact, LLVM has been adopted by many commerical and open source projects. For example, Intel, Cray, AMD, and IBM have all completed LLVM adoption of their compilers. NVIDIA's nvcc compiler is also based on LLVM.

With LLVM, converting a source code to a machine code (also known as an object code) for a target processor involves three phases:

1. **Frontend Phase**

   In this phase, the front end parses, validates, and diagnoses errors in the source code, and converts the parsed code into the LLVM language-independent Intermediate Representation (IR).

2. **Optimizer Phase**

   The IR goes through a series of optimization passes (e.g., inlining, unrolling, dead code elimination, etc.)

3. **Backend Phase**

   The optimized IR is transformed into a target-specific machine code.

Front end options include Clang (for a C/C++ language compiler) and Flang (a Fortran language front end integrated with LLVM). Both are projects under the LLVM Project umbrella.

The Intel oneAPI DPC++/C++ Compiler uses the Clang front end and supports the C, C++, and SYCL language standards and the OpenMP 5.0/5.1 parallel programming model.

The Intel Fortran Compiler (ifx) is based on Intel Fortran Compiler Classic (ifort) frontend and runtime libraries, but uses the LLVM backend compiler technology. The ifx release included in oneAPI 2023.0 completely implements Fortran 77, Fortran 90/95, Fortran 2003, Fortran 2008, and Fortran 2018 language standards, as well as most OpenMP 4.5 and OpenMPI 5.0/5.1 directives and offloading features. The ifx release included in oneAPI 2025.0 adds support for OpenMP 6.0 TARGET, automatic DO CONCURRENT offload to Intel GPUs, and select Fortran 2023 features.

Comparison of icc/icpc and icx/icpx

The following table provides a few similarities and differences between icc/icpc and icx/icpx. For more information, see the Porting Guide for ICC Users to DPCPP or ICX.

Compiler Driver	icc, icpc	icx, icpx
Hardware Support	CPU	CPU, GPU
Compiler Options	See icc --help	See icx --help; icx/icpx will accept most icc/icpc options or Clang/LLVM options; The option -Xclang can be used to pass options to Clang when necessary
Optimization Levels	-O2 (default)	-O2 (default); note: -O2 or -O3 in icx/icpx is not sufficient for good performance, need processor target option (e.g., -xskylake-avx512) to enable extra levels of loop optimization and vectorization
Floating Point Model	Default to -fp-model=fast	Default to -fp-model=fast; note: this behavior is a deviation from Clang
Cross Multi-file Optimization	Interprocedural Optimization (IPO)	Link Time Optimization (LTO)
Pragma Support Besides OpenMP	icc has many proprietary Intel pragmas	Only a subset of Intel pragmas are supported in icx. Add the option -Wunknown-pragmas during compile time to check if a pragma is supported

Comparison of ifort and ifx

The following table provides a few similarities and differences between ifort and ifx. For more information, see the Porting Guide for Intel Fortran Compiler.

Compiler Driver	ifort	ifx
Hardware Support	CPU	CPU, GPU
OpenMP Offload Support	N/A	5.0, 5.1 GPU offload
Compiler Options	See ifort --help	See ifx --help; most but not all documented ifort options are supported by ifx, e.g., -m32, -fp-model={source, consistent} are not supported; -parallel is available in 2025.0 version but not in 2023 versions; undocumented ifort options are not accepted by ifx
Cross Multi-file Optimization	Interprocedural Optimization (IPO)	Link Time Optimization (LTO)
Binary (.o/.obj), Library and Module (.mod) Compatibility	Those generated by ifort can be linked/used with those built with ifx; One exception is when -ipo is used	vice versa
Runtime Library	Same as ifx	Same as ifort
CPU Performance	Best in class	Continued improvement by Intel to catch up with ifort performance

References

- Getting to Know LLVM
- Getting Started with the Intel oneAPI DPC++/C++ Compiler
- Getting Started with the Intel Fortran Compiler
- Porting Guide for ICC Users to DPCPP or ICX
- Porting Guide for Intel Fortran Compiler
- Intel oneAPI DPC++/C++ Compiler Developer Guide and Reference
- Intel Fortran Compiler Developer Guide and Reference

Intel compilers are recommended for building your applications on HECC systems with Intel x86-64 CPUs. There is no default compiler—you must load a specific module. To see what modules are available, run the module avail command:

```
pfe21% module avail
```

To load a specific version, replace *version* in the command below with the actual version number:

```
pfe21% module load comp-intel/version
```

Some important compiler flags to keep in mind are described in the following sections.

Note: Most of the flags described in this article apply to both the Intel classic compilers and the Intel LLVM-based compilers.

Turn On Optimization

If you do not specify an optimization level (-O*n*, *n=0,1,2,3*), the default is -O2. If you want more aggressive optimizations, you can use -O3.

Note: Using -O3 may not improve performance for some programs.

Generate Optimized Code for a Processor Type

Intel compilers provide flags for generating optimized codes for various instruction sets used in specific processors or micro-architectures. If your goal is to achieve the best performance from a specific processor, we recommend that you build your application using the latest Intel compiler with the flag that corresponds with the processor you plan to use:

Processor Type	System	Flag
Cascade Lake	Aitken/Endeavour	-xCORE-AVX512
Skylake	Electra	
Broadwell	Electra Pleiades (Retired)	-xCORE-AVX2
Haswell	Pleiades (Retired)	
Ivy Bridge	Pleiades (Retired)	-xAVX
Sandy Bridge	Pleiades (Retired)	

Note: The instruction sets are upward compatible, therefore, applications compiled with:

- -xAVX can run on Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, and Cascade Lake processors.
- -xCORE-AVX2 can run on Haswell, Broadwell, Skylake, and Cascade Lake processors.
- -xCORE-AVX512 can run *only* on Skylake and Cascade Lake processors.

WARNING: Executables built with a flag that is not compatible with a processor will not run on that processor, and will result in an error with a message similar to the following:

```
Fatal Error: This program was not built to run in your system. Please verify that both the operating system and the processor support Intel(R) AVX2
```

AMD processors: See Preparing to Run on Aitken Rome Nodes for information about compilers to use for Rome processors. See Preparing to Run on Aitken Milan Nodes for information about compilers to use for Milan processors.

Generate Code for Any Processor Type

If your goal is to create a portable executable file that can run on any processor type on Electra, Aitken, and Endeavour, you can choose one of the following approaches:

- Use none of the above flags (which defaults to -mSSE2)
- Use -O3 -axCORE-AVX512,CORE-AVX2 -xAVX (with the latest Intel compiler)

This allows a single executable that will run on any of the processor types, with suitable optimization to be determined at run time.

WARNING: Be careful if you plan to add the -Ofast flag, which sets -xHost (and other options) automatically during compilation. If you place Ofast after -axCORE-AVX512,CORE-AVX2 -xAVX, -xHost takes precedence over -axCORE-AVX512,CORE-AVX2 -xAVX, and the executable is generated for the compilation host processor.

Turn On Inlining

Use the -ip or -ipo flags.

Using -ip enables additional interprocedural (IP) optimizations for single-file compilation. One of these optimizations enables the compiler to perform inline function expansion for calls to functions defined within the current source file.

Using -ipo enables multi-file IP optimizations between files. When you specify this option, the compiler performs inline function

expansion for calls to functions defined in separate files.

Use a Specific Memory Model

If you get a link time error relating to R_X86_64_PC32, add the compiler option `-mcmodel=medium` and the link option `-shared-intel`. This happens if a common block is more than 2 GB in size.

Turn Off All Warning Messages

Using the `-w` option disables all warnings, and `-qopt-report=0` disables printing of optimization reports.

Parallelize Your Code

The `-qopenmp` option handles OMP directives and `-parallel` looks for loops to parallelize.

Note: The `-parallel` option is available in the Intel 2025 release of `ifx`, but not in the 2023 release of `ifx`.

For more compiler/linker options, see **man ifort, man icc, man ifx, man icx** .

Debugging Options

See Recommended Intel Compiler Debugging Options for descriptions of several more compiler flags.

UPDATE IN PROGRESS: On August 24, 2026, the Skylake/V100 and Cascade Lake/V100 nodes were decommissioned from the Cabeus supercomputer. This article is currently undergoing updates to reflect the changes in the NAS environment and may not accurately reflect the resources available to users.

To develop code for use on the NVIDIA GPU nodes, you can choose one of the three NVIDIA-recommended programming models or combine them to take advantage of the underlying NVIDIA acceleration libraries:

- Standard-language-based parallel programming model.
- Directive-based OpenACC model.
- Language-extension-based CUDA model.

Each model is briefly described in this article. Read the relevant references for each model listed at the end of this article to learn more information and limitations.

Note: NVIDIA provides limited support for using OpenMP on the GPUs. This model is not described in this article.

Compiler and CUDA Support

Compiler support for these models is provided through the NVIDIA HPC Software Development Kit (SDK), which includes these compilers:

- NVIDIA HPC Compilers for NVIDIA GPUs and Intel, AMD, OpenPower, and Arm CPUs:
  - NVFORTRAN: Fortran compiler with support for CUDA Fortran, originally developed by PGI
  - NVC: C11 compiler
  - NVC++: C++17 compiler
- NVIDIA CUDA C and C++ Compiler Driver for NVIDIA GPUs: NVCC

Note: The PGI compilers and tools were rebranded to become parts of the NVIDIA HPC SDK.

As these models are under active development or improvement, we recommend using the latest version of the NVIDIA HPC SDK, as it should have better support for new features when they are added to these programming models. Be aware that the available versions of NVIDIA HPC SDK at NAS vary between the GPU nodes with x86_64-based CPU (sky_gpu, cas_gpu, rom_gpu, mil_a100) and those with the aarch64-based CPU (gh200). Run the command `module avail nvhpc` to see what are available on each system before using the command `module load` to load one of them.

For more information on each compiler, read its man page or help page after loading a module. For example:

- man nvfortran
- nvfortran -h
- nvfortran -gpu -h

During runtime, your CUDA application requires one CUDA-capable GPU (for example, a V100, A100, or H100 GPU) and a CUDA driver that is compatible with the version of CUDA Toolkit used for building the application. Be aware that the CUDA driver is backward compatible. Therefore, an application compiled with an older CUDA Toolkit version can run with a newer CUDA driver. See CUDA Toolkit and CUDA Driver to learn more.

Note: The CUDA driver is installed by the system administrator on the GPU. Run either the `nvidia-smi` or the `nvaccelinfo` command to find the CUDA driver version installed on the system.

The Standard-Language-Based Parallel Programming Model

In this model, parallelism for running on the NVIDIA GPUs is achieved solely using features provided by the standard C++ and Fortran languages without the use of compiler directives such as OpenMP and OpenACC or language extensions such as CUDA. In addition, applications written with standard language parallelism can run on any platform—such as different multi-core CPUs or GPUs from different vendors—that supports these languages.

International Organization for Standardization (ISO) C++

The C++17 Standard introduced higher-level parallelism features that allow you to request a specific execution policy as the first parameter to a Standard Library algorithm. Most Standard C++ algorithms support the execution policies. The four available policies are:

- **std::execution::seq:** Sequential execution. No parallelism is allowed.
- **std::execution::unseq:** Vectorized execution on the calling thread (this execution policy was added in C++20).
- **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.

Here is an example from NVIDIA documentation—a sort algorithm with the par execution policy:

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

NVC++ can compile Standard C++ algorithms with the `par` or the `par_unseq` execution policies through the use of the `-stdpar` compiler

flag and automatically offloads C++ parallel algorithms to NVIDIA Volta, Turing, Amphere, or Hopper GPUs.

ISO Fortran

ISO Fortran 2018 introduced the DO CONCURRENT construct to allow you to express loop-level parallelism:

```
DO CONCURRENT (i=1:n)
  loop body
END DO
```

When the NVFORTRAN -stdpar option is specified, the compiler parallelizes the DO CONCURRENT loops and offloads them to the GPU.

Example codes that use this model are available in the NVIDIA HPC SDK installed on NAS systems.

For GPU nodes with x86_64-based CPU:

```
/nasa/nvidia/hpc_sdk/..../examples/stdpar
```

For GPU nodes with aarch64-based CPU:

```
/nasa/arm/software/nvhpc/..../examples/stdpar
```

You can compile code either on a GPU node; on a Cabeus front end (CFE), to run on the GPUs with x86_64-based CPUs (sky_gpu, cas_gpu, rom_gpu, mil_a100); or on cghfe01 and cghfe02 to run on the GPUs with aarch64 based CPU (gh200):

```
nvc++ -stdpar[=gpu] [-gpu=ccXY] program.cpp
nvfortran -stdpar[=gpu] [-gpu=ccXY] program.f90
```

In the event you want to compile the same code for running on multicore CPUs, use:

```
nvc++ -stdpar=multicore program.cpp
nvfortran -stdpar=multicore program.f90
```

To facilitate such standard language parallelism, NVC++ and NVFORTRAN rely on CUDA Unified Memory for all data movement between CPU and GPU memory. 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.

NVC++ and NVFORTRAN auto-detect the host CPU and device GPU architectures of the system where you perform the compilation. An executable built on one system may not run on another system if the host CPU instructions and/or the GPU instructions in the executable are not compatible between the two systems.

To generate an executable for a specific CPU architecture, add the -tp option. For example, use -tp=skylake for a CPU host with Intel Skylake or Cascade Lake processors, -tp=zen2 for the AMD Rome processor, or -tp=zen3 for the AMD Milan processor. For the GH200 GPU nodes (gh200), NVIDIA recommends using the flag -tp=native or -mcpu=native.

To generate an executable for a specific GPU architecture, add the -gpu=*ccXY* option, where *ccXY* stands for the compute capability of an architecture. For example, use -gpu=cc70 for V100 nodes, -gpu=cc80 for A100 nodes, and -gpu=cc90 for the GH200 nodes. Using multiple -gpu=*ccXY* options in a single compilation will generate one version of code for each architecture, but the compilation time will be longer.

Executables built for the x86_64 platform (PFE, AFE, CFE, ivy, has, bro, sky_ele, cas_ait, rom_ait, mil_ait, sky_gpu, cas_gpu, rom_gpu, and mil_100 GPU nodes) cannot be used on the aarch64 platform (gh200 GPU nodes) because x86_64 and aarch64 architectures have different instruction sets.

Note: A executable built with just -gpu=cc70 can run on an architecture with compute capability cc70 (V100) or cc80 (A100). However, an executable built with just -gpu=cc80 can run on an architecture with compute capability cc80 (A100) but not cc70 (V100).

Note: -stdpar defaults to -stdpar=gpu. If you compile the application with -stdpar on a Cabeus front end (cfe01 and cfe02, for running on the GPU nodes with x86-based CPUs) or on cghfe01 or cghfe02 (for running on the GPU nodes with aarch64-based CPUs) but if you do not explicitly specify one or more -gpu=*ccXY*, it will automatically enable: -gpu=cc50 -gpu=cc60 -gpu=cc61 -gpu=cc70 -gpu=cc75 -gpu=cc80 -gpu=cc86 -gpu=cc90

The Directive-Based OpenACC Model

OpenACC is a programming standard developed by PGI, CRAY, CAPS, and NVIDIA for offloading programs written in C, C++, and Fortran from a host CPU to an attached accelerator device.

Similar to OpenMP, OpenACC requires you to annotate your source code with directives describing sections of your code and corresponding data that are to be executed on the GPU:

- For C and C++, the directive is #pragma acc ...
- For Fortran, the directive is !$acc ...

Example OpenACC codes from NVIDIA are available on the NAS systems under:

For GPU nodes with x86_64-based CPUs:

```
/nasa/nvidia/hpc_sdk/..../examples/OpenACC
```

For GPU nodes with aarch64-based CPUs:

/nasa/arm/software/nvhpc/..../examples/OpenACC

To compile your OpenACC code, use the NVIDIA HPC compilers (nvfortran, nvc, or nvc++) and options on a system with NVIDIA GPUs. For example:

```
nvfortran -acc=gpu -fast -Minfo=all your_program.f
nvc       -acc=gpu -fast -Minfo=all your_program.c
nvc++     -acc=gpu -fast -Minfo=all your_program.cc
```

where -acc=gpu will parallelize and offload OpenACC regions to the NVIDIA GPUs. The option -Minfo will generate informational messages about optimization. You can specify -acc=multicore to parallelize for a multicore CPU or -acc=host to generate an executable that will run serially on the host CPU.

As in the case of the standard-language-based model described above, you can also use the -tp option to specify a target CPU architecture and/or the -gpu=*ccXY* option to target GPUs with the specified compute capability.

NVIDIA OpenACC supports multiple environment variables to control behavior of accelerator-enabled programs at execution. See the Environment Variables section of the OpenACC Getting Started Guide listed in the References.

Note on OpenMP Support: Although NVIDIA does not emphasize the use of OpenMP, NVIDIA HPC compilers do support a subset of the OpenMP API with a focus on OpenMP 5.0 features for running on CPUs and GPUs. Similar to compiling OpenACC code with -acc=gpu -gpu=cc80, use the compiler flags -mp=gpu -gpu=cc80 to compile OpenMP code for offloading to run on the A100 GPUs. For codes that are well-structured for the GPUs, these should compile and execute with performance equal to or close to equivalent OpenACC. Codes that are not well-structured for GPUs may perform poorly but should execute correctly. Read the OpenMP chapter of the NVIDIA HPC Compilers User's Guide for more information.

The Language-Based CUDA Model

The Compute Unified Device Architecture (CUDA) was created by NVIDIA and implemented on the NVIDIA GPUs. With CUDA, code developers write programs in C, C++, and Fortran, and incorporate CUDA extensions and optimized libraries. This is the approach to take if you want to obtain the best possible performance on the NVIDIA platform.

A CUDA code usually includes the following sequence of operations:

1. Declare and allocate the host memory (i.e., the CPU) and device (i.e. the GPU) memory.
2. Initialize host data.
3. Copy data from host to device.
4. Run the kernels on the GPUs.
5. Copy the results from device to host.
6. Free the allocated host and device memory.

You can use either of the following programming environments:

CUDA C/C++

CUDA C/C++ is the original CUDA programming environment developed by NVIDIA as extensions to the C/C++ languages for GPUs. It allows direct programming for the GPU from a high-level language. This NVIDIA technical blog provides an easy introduction to CUDA C and C++.

If you choose to write or rewrite portions of your code in CUDA C/C++, you will need to use the NVIDIA CUDA C/C++ compiler (nvcc) to build the executable. For example:

nvcc *your_program*.cu

The suffix .cu indicates it is a CUDA C/C++ file.

A CUDA application binary (with one or more GPU kernels) can contain compiled GPU code in two forms: (1) binary cubin objects and (2) forward-compatible PTX assembly for each kernel. During application load time, the PTX form is compiled just-in-time (JIT) to a cubin for execution.

To build an application that can run on either V100 or A100 nodes, or on a future architecture, use the -gencode option as follows:

```
nvcc -gencode=arch=compute_70,code=sm_70
    -gencode=arch=compute_80,code=sm_80
    -gencode=arch=compute_80,code=compute_80
    your_program.cu
```

where the first and second -gencode options ending with code=sm_70 and sm_80 cause the generation of a cubin targeting V100 or A100, respectively, and the last -gencode option ending with code=compute_80 allows for the generation of PTX assembly to provide compatibility of future architectures through JIT compiling.

To build an application that can run on the GH200 nodes, use:

```
nvcc -gencode=arch=compute_90,code=sm_90
    -gencode=arch=compute_90,code=compute_90
    your_program.cu
```

CUDA Fortran

CUDA Fortran is a product of a collaboration between PGI and NVIDIA. It is based on the CUDA C/C++ runtime API, and includes a Fortran 2003 compiler and tool chain for programming NVIDIA GPUs using Fortran. Read this NVIDIA Developer Blog to get a quick introduction of CUDA Fortran.

Example CUDA Fortran codes are available in the NVIDIA HPC SDK installed on NAS systems:

For GPU nodes with x86_64-based CPUs:

/nasa/nvidia/hpc_sdk/..../examples/CUDA-Fortran

For GPU nodes with aarch64-based CPUs:

/nasa/arm/software/nvhpc/..../examples/CUDA-Fortran

To use CUDA Fortran, use the compiler nvfortran to build the executable. For example:

nvfortran *your_program*.cuf

Files with .cuf or .CUF extensions automatically enable CUDA Fortran compilation. If the filename of your program does not end with the .cuf or .CUF suffix, use the -cuda option. For example:

nvfortran -cuda *your_program.xx*

As in the case of the previous two programming models described above, you can also use the -tp option to specify a target CPU architecture and/or the -gpu=*ccXY* option to target GPUs with the specified compute capability.

If you have any questions or need assistance, contact the NAS Control Room at (800) 331-8737, (650) 604-4444, or support@nas.nasa.gov.

References for NVIDIA HPC Compilers

- NVIDIA HPC Compilers User's Guide

References for the Standard-Language-Based Parallel Programming Model

- Video - No More Porting: Coding for GPUs with Standard C++, Fortran and Python
- Developing Accelerated Code with Standard Language Parallelism
- C++ Parallel Algorithms
- Accelerating Standard C++ with GPUs Using stdpar
- Multi-GPU programming with standard Parallel C++, Part 1
- Multi-GPU programming with standard Parallel C++, Part 2
- Accelerating Fortran DO CONCURRENT with GPUs and the NVIDIA HPC SDK
- Using Fortran Standard Parallel Programming for GPU Acceleration
- Bringing Tensor Cores to Standard Fortran

Reference for the OpenACC Programming Model

- OpenACC Getting Started Guide

References for the the Language-Extension-Based CUDA Model

- An Even Easier Introduction to CUDA
- CUDA C++ Programming Guide
- NVIDIA Amphere GPU Architecture Compatability Guide for CUDA Applications
- An Easy Introduction to CUDA Fortran
- NVIDIA CUDA Fortran Programming Guide
- Using Tensor Cores in CUDA Fortran