Porting and Optimizing HPC Applications for Arm

Version 1.0.0

Document Number 101725_0100_00

CONTENTS:

CHAPTER

ONE

PREFACE

1.1 Release Information

Table 1: Document history

Issue	Date	Confidentiality	Change
0100-00	29 March 2019	Non-Confidential	First release

1.2 Non-Confidential Proprietary Notice

LES-PRE-20349

1.3 Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

1.4 Product Status

The information in this document is Final, that is for a developed product.

1.5 Web Address

http://www.arm.com

CHAPTER

TWO

STEPS AND TOOLS TO PORT YOUR APPLICATION

This chapter describes the tools available, and the steps to take, to help you port and optimize your applications for the Arm architecture.

Most applications will port onto the Arm architecture with little or no modification, because:

- Arm is supported by all major Linux distributions, which provide a rich library of common Linux packages built for AArch64.
- Applications and dependencies can be recompiled using compilers that support AArch64 applications for Linux user space
- GNU Compiler Collection (GCC) is fully supported.
- The commercially-supported Arm Compiler is available, which also accepts GCC compiler options, wherever possible.

However, there are a few features of the Arm architecture that may impact your application, these are detailed under **Troubleshooting** in the *Port your Application* topic.

2.1 Tools to Port and Optimize

Arm Allinea Studio provides the following tools to help port and optimize applications for Arm:

2.1.1 Arm Compiler

A commercially-supported, Linux user-space, C/C++ and Fortran compiler, tuned for scientific computing, HPC, and enterprise workloads:

- Processor-specific optimizations for various server-class Arm-based platforms.
- Optimal shared-memory parallelism using latest Arm-optimized OpenMP runtime.
- Optimized scalar and vector maths functions.
- C++ 17 and Fortran 2003 language support with OpenMP 4.5.
- Support for Armv8-A and SVE architecture extension.
- Based on LLVM and Flang, leading open-source compiler projects.
- Runs on leading Linux distributions: RedHat, SUSE, and Ubuntu.

Resources

- For the latest release information and other resources, see the Arm Compiler Developer web page.
- *Arm Compiler quick reference*.
- Arm C/C++ Compiler documentation.
- Arm Fortran Compiler documentation.

2.1.2 Arm Performance Libraries

Commercially-supported, 64-bit Armv8-A core math libraries, optimized for HPC applications on Arm-based platforms, providing best-in-class serial and parallel performance:

- Commonly used low-level math routines: BLAS, LAPACK, and FFT.
- Provides FFTW-compatible interface for FFT routines.
- Batched BLAS support.
- Generic Armv8-A optimizations by Arm.
- Tuning for specific platforms in collaboration with silicon vendors.
- Validated with NAG's industry standard test suite.
- Available for Arm Compiler or GCC.

Resources

- For the latest release information and other resources, see the Arm Performance Libraries Developer web page.
- *Arm Performance Libraries Quick Reference*.
- For routine-specific reference information, see the Arm Performance Libraries reference guide.

2.1.3 Arm Forge Professional

A cross-platform toolkit to debug (Arm DDT) and profile (Arm MAP) high-performance parallel applications:

- Available on the vast majority of the Top500 machines in the world.
- Fully supported by Arm on x86, IBM Power, Nvidia GPUs, etc.
- Powerful and in-depth error detection mechanisms (including memory debugging).
- Sampling-based profiler to identify and understand bottlenecks.
- Available at any scale (from the desktop to leadership-class HPC).
- Unique capabilities to simplify remote interactive sessions.
- Innovative approach to present essential information to users.

Resources

- For the latest release information and other resources, see the Arm Forge Developer web page.
- *Arm DDT Quick Reference*.
- For debugging documentation, see Arm DDT documentation.
- *Arm MAP Quick Reference*.
- For profiling documentation, see Arm MAP documentation.

2.1.4 Arm Performance Reports

Characterize and understand the performance of HPC application runs:

- Analyze metrics around CPU, memory, IO, and hardware counters.
- Users can to add their own metrics.
- Analyze data and report the information that matters to users.
- Provides simple guidance on how to improve workload efficiency.
- Define application behavior and performance expectations.

- Integrate outputs to various systems for validation (for example, continuous integration).
- Can be automated completely (no user intervention).

Resources

- *Arm Performance Reports Quick Reference*.
- For the latest release information and other resources, see the Arm Performance Reports Developer web page.

2.2 Port your Application

To port your application, follow these steps:

Note: If you encounter any issues with your build, see the *Porting - Troubleshooting* section below.

---

1. Ensure all your application dependencies have been ported.

   Use of external libraries is increasingly common, and a conscious design choice for many projects. Common dependencies include:

   - **IO libraries.** For example, HDF5 and NetCDF (C, parallel, and Fortran flavors).
   - **Maths libraries and toolkits**. For example PETSc, HYPRE, Trilinos, ScaLAPACK, LAPACK and BLAS.

     Note: Arm Performance Libraries provides optimized LAPACK and BLAS implementations.

   - **Fast fourier transforms**. For example, FFTW.

     Note: Arm Performance Libaries provised an optimised FFT implementation which is compatible with FFTW's interface.

   - **Communication layers, or execution environments**. For example, Open MPI, OpenUCX, and Charm++.
   - **Libraries providing performance portability and memory abstraction**. For example, Kokkos and RAJA.

   In most cases you will find that these dependencies have been built on Arm before, with the Arm and GNU toolchains:

   - See the Porting and tuning guides on Arm Developer for instructions on porting many open-source applications using both the GCC and Arm toolchains.
   - For a full list of all ported applications using the Arm and GNU toolchains, see the community Packages Wiki.

2. Check you are using the correct compiler.

   During your build configuration, specify which C, C++, and Fortran compilers to use. For example, for Arm Compiler you would typically set:

   ```
   CC=armclang
   CXX=armclang++
   FC=armflang
   F77=armflang
   ```

   For GCC:

```
CC=gcc
CXX=g++
FC=gfortran
F77=gfortran
```

For MPI builds (for example, Open MPI) you might need to use the MPI wrappers. These are usually the same for all compilers:

```
CC=mpicc
CXX=mpicxx
FC=mpifort
```

3. Check you are using the right compiler options. Most GCC options are supported by Arm Compiler. It is recommended that you use `-mcpu=native` in addition to any other options to ensure you get compiled code that is tuned for the micro-architecture of your machine.
4. Build your application as you would normally.
5. Run your test suite.

> **Warning:** Regression tests that rely on bit-wise identical answers might not be portable between architectures.

2.2.1 Porting - troubleshooting

Here are some problems you might encounter while porting your application:

2.2.1.1 Configure is unable to identify your platform

This may be due to the `config.guess` supplied with the application being out of date. This can also be true for a `config.guess` already installed on your system and used by some configure scripts.

Solution

To fix this problem, obtain up-to-date versions:

```
wget 'http://git.savannah.org/gitweb/?p=config.git;a=blob_plain;f=config.guess;
↪hb=HEAD' -O config.guess
wget 'http://git.savannah.gnu.org/gitweb/?p=config.git;a=blob_plain;f=config.sub;
↪hb=HEAD' -O config.sub
```

2.2.1.2 Libtool fails to link Fortran applications or interfaces

Libtool does not recognize Arm Compiler as a Fortran compiler. Therefore, it is unable to set the correct flags for linking the binary.

Solution

Ensure Libtool uses the correct compiler options with Arm Compiler by modifying it after running configure:

```
sed -i -e 's#wl=""#wl="-Wl,"#g' libtool
sed -i -e 's#pic_flag=""#pic_flag=" -fPIC -DPIC"#g' libtool
```

Some widely used applications and libraries, for example Open MPI, have already incorporated a fix to address this issue at the configure stage.

2.2.1.3 #ifdefs in the makefile are not being set

There may be compiler-dependent `#ifdefs` in the source which are not being set.

Solution

You might need to update the source to use the `_FLANG` and `_clang` macros, or manually set existing compiler macros, such as `-D_PGI`.

2.2.1.4 Unsupported language features

Your code might be making use of language features which are not currently supported by Arm Compiler.

Solution

Check the support status of the compiler, for:

- Fortran 2003 standard support.
- Fortran 2008 standard support (Partial).
- Fortran OpenMP 4.0 and ‘ Fortran OpenMP 4.5 (Partial) <https://developer.arm.com/docs/101380/latest/standards-support/openmp-45>‘_ support.
- C/C++ OpenMP 4.0 and C/C++ OpenMP 4.5 (Partial) support.

2.2.1.5 You are experiencing a race condition you have not encountered before

AArch64 adopts a weak memory model. This means that read and writes can be re-ordered. In some cases it means that explicit memory barriers are needed on AArch64 that were not required on other architectures.

Solution

Implement explicit memory barriers for AArch64. These are described in Barriers in the Arm Cortex-A Series Programmer's guide for Armv8-A, and in Appendix J of the ARM Architecture Reference Manual ARMv8, for ARMv8-A archictecture profile.

2.2.1.6 Do you have an integer divide by zero?

On AArch64, an integer divide by zero does not generate an error; instead it returns as zero.

Note: This is not the case for floating-point divide by zero.

On rare occasions, an undetected divide by zero might be allowing an application to run erroneously when it should fail.

Solution

It might be necessary to explicitly catch attempted divide-by-zeros in software. For example, if you have an equation such as:

```
c = a / b
```

Explicitly catch for `b` being equal to zero, using:

```
if b==0 then throw_error("Attempted divide by zero")
```

2.2.1.7 Thread mapping and pinning on Arm

Arm chips can have lots of cores. It is very important to manage how your threads get mapped to the cores, and how they are pinned.

Solution

Map your threads to cores using the available mapping devices:

- OpenMP environment variables
- OpenMPI run flags
- Numactl

2.2.1.8 Segmentation fault when calling an Arm Performance Libaries function

Segmentation faults can occur when you are linking against the wrong version of the library with either 32-bit integers or 64-bit integers.

Solution

Compile and link for 32-bit integers (`-armpl=lp64`) or for 64-bit integers (`-armpl=ipl64`), as required.

2.2.1.9 Building Position Independent Code (PIC) on AArch64

Failure can occur at the linking stage when building Position-Independent Code (PIC) on AArch64 using the lower-case `-fpic` compiler flag with GCC compilers (gfortran, gcc, g++), in preference to using the upper-case `-fPIC` flag.

Note:

- This issue does not occur when using the `-fpic` flag with Arm compilers for HPC (armflang/armclang/armclang++), and it also does not occur on x86_64 because `-fpic` operates the same as `-fPIC`.
- PIC is code which is suitable for shared libraries.

Cause

Using the `-fpic` compiler flag with GCC compilers on AArch64 causes the compiler to generate one less instruction per address computation in the code, and can provide code size and performance benefits. However, it also sets a limit of 32k for the Global Offset Table (GOT), and the build can fail at the executable linking stage because the GOT overflows.

Note: When building PIC with Arm Compiler for HPC on AArch64, or building PIC on x86_64, `-fpic` does not set a limit for the GOT, and this issue does not occur.

Solution

Consider using the `-fPIC` compiler flag with GCC compilers on AArch64, because it ensures that the size of the GOT for a dynamically linked executable will be large enough to allow the entries to be resolved by the dynamic loader.

2.2.1.10 Applications supporting GCC builds on Arm - but use Arm®v7 compiler flags

Some Arm®v7 flags that are needed for Arm®v7, cause errors for Arm®v8 targets. For example, on Arm®v8 Neon is compulsory, so the flag `-fp=neon` does not exist on Arm®v8. If it is used when compiling for Arm®v8, GCC does not recognize it and causes an error.

Cause Typically, the flags are incorrect in makefiles.

Solution Update your makefiles to only use compatible Arm®v8 compiler flags.

2.2.2 Related information

For more information on porting your application to Arm, see:

- Get started on Arm
- Arm porting and tuning guides
- Packages Wiki
- Develop on Arm
- Arm Compiler for HPC
- Arm Performance Libraries

2.3 Optimize

To optimize your application:

1. Start by compiling your application with the `-mcpu=native` and `-O3` compiler options. Consider using the `-Ofast` option. For C code, try compiling your application with the `-fsimdmath` option. The `-fsimdmath` option provides a vectorised implementation of common libm calls.

   Note:

   - For Fortran source, vector implementations are used, when possible, by default, but can be disabled using the `-fnosimdmath` compiler flag.
   - The `-Ofast` option enables all the optimizations from `-O3`, but also performs other aggressive optimizations that might violate strict compliance with language standards.
   - If your Fortran application runs into issues with `-Ofast`, to force automatic arrays on the heap, try `-Ofast -fno-stack-arrays`.
   - If `-Ofast` is not acceptable and produces the wrong results because of the reordering of math operations, use `-O3 -ffp-contract=fast`.
   - If `-ffp-contract=fast` does not produce the correct results, then use `-O3`.
   - For a full list of compiler options, see the Arm C/C++ Compiler reference guide and Arm Fortran Compiler reference guide.

   ---

2. Use the optimized Arm Performance Libraries with the `-armpl` compiler option.

   Arm Performance Libraries provide optimized standard core math libraries for high-performance computing applications on Arm processors:

   - **BLAS - Basic Linear Algebra Subprograms (including XBLAS, the** extended precision BLAS).
   - **LAPACK** - a comprehensive package of higher level linear algebra routines.
   - **FFT** - a set of Fast Fourier Transform routines for real and complex data. Arm Performance Libraries support FFTWs Basic, Advanced, Guru, and MPI interfaces.

   The compiler option `-armpl` makes these libraries significantly easier to use with a simple interface to select thread-parallelism and architectural tuning. Arm Performance Libraries also provides improved Fortran math intrinsics with auto-vectorization.

   The `-armpl` and `-mcpu` options enable the compiler to find appropriate Arm Performance Libraries header files (during compilation) and libraries (during linking). Both options are required for the best results.

Note:

- If your build process compiles and links as two separate steps, please ensure you add the same `-armpl` and `-mcpu` options to both. For more information about using the `-armpl` option, see Getting Started with Arm Performance Libraries on the Arm Developer website.
- For GCC, you will need to load the correct environment module for the system and explicitly link to your chosen flavor (lp64/ilp64, mp) with full library path.

For more information, refer to the Arm Performance Libraries Developer web page.

3. Use Arm Compiler optimization remarks.

   Optimization remarks provide you with information about the choices made by the compiler. They can be used to see which code has been inlined or to understand why a loop has not been vectorized.

   Optimization remarks are enabled by passing one or more of the following `-Rpass` flags at the command line:

Table 1: -Rpass flags to enable optimization remarks

-Rpass flags	Description
`-Rpass=<regexp>`	To request information about what Arm Compiler has optimized.
`-Rpass-analysis=<regexp>`	To request information about what Arm Compiler has analyzed.
`-Rpass-missed=<regexp>`	To request information about what Arm Compiler failed to optimize.

In each case, `<regexp>` is used to select the type of remarks to provide. For example, loop-vectorize for information on vectorization, and inline for information on in-lining, or `.*` to report all optimization remarks. `Rpass` accepts regular expressions, so (loop-vectorize|inline) can be used to capture any remark on vectorization or inlining.

Note:

- Optimization remarks are only available when you have set an appropriate debug flag, such as `-g`.
- Optimization remarks are piped to `stdout` at compile time.

For example, to get actionable information on which loops can, and cannot, be vectorized (including why) at compile time, set:

```
-Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-
↪vectorize -g
```

For more information, refer to the optimization remarks documentation for Fortran or for C/C++.

2.3.1 Get help with optimization

In armflang, optimization remarks are enabled by passing `-Rpass` command line options. Optimization remarks are a feature of LLVM compilers that provides information about the choices made by the compiler about inlining, vectorization, and more.

Optimization remarks are enabled by passing one or more of the following `-Rpass` flags at the command line:

Table 2: Optimization remarks

-Rpass flags	Description
`-Rpass=<regexp>`	To request information about what Arm Compiler has optimized.
`-Rpass-analysis=<regexp>`	To request information about what Arm Compiler has analyzed.
`-Rpass-missed=<regexp>`	To request information about what Arm Compiler failed to optimize.

In each case, `<regexp>` is used to select the type of remarks to provide. For example, loop-vectorize for information on vectorization, and inline for information on in-lining. `Rpass` accepts regular expressions, so (loop-vectorize|inline) can be used to capture any remark on vectorization or in-lining.

Optimization remarks are piped to `stdout` at compile time. For more information, see Using Optimization Remarks with Arm Fortran Compiler or Using Optimization Remarks with Arm C/C++ Compiler.

Note: Optimization remarks requires that an appropriate debug flag is set, such as `-g`.

---

1. Use the Arm Compiler directives.

   Arm Fortran Compiler supports general-purpose and OpenMP-specific directives:

   - `!DIR$ IVDEP` - A generic directive to force the compiler to ignore any potential memory dependencies of iterative loops and vectorize the loop.
   - `!$OMP SIMD` - An OpenMP directive to indicate a loop can be transformed into a SIMD loop.
   - `!DIR$ VECTOR ALWAYS` - Forces the compiler to vectorize a loop irrespective of any potential performance implications.

     Note: The loop must be vectorizable.

   - `!DIR$ NO VECTOR` - Disables vectorization of a loop.
   - `!DIR$ UNROLL` - Instructs the compiler to unroll the loop it precedes.
   - `!DIR$ NOUNROLL` - Instructs the compiler not to unroll the loop it precedes.

   For more information, see the directives section of the Arm Fortran Compiler reference guide.

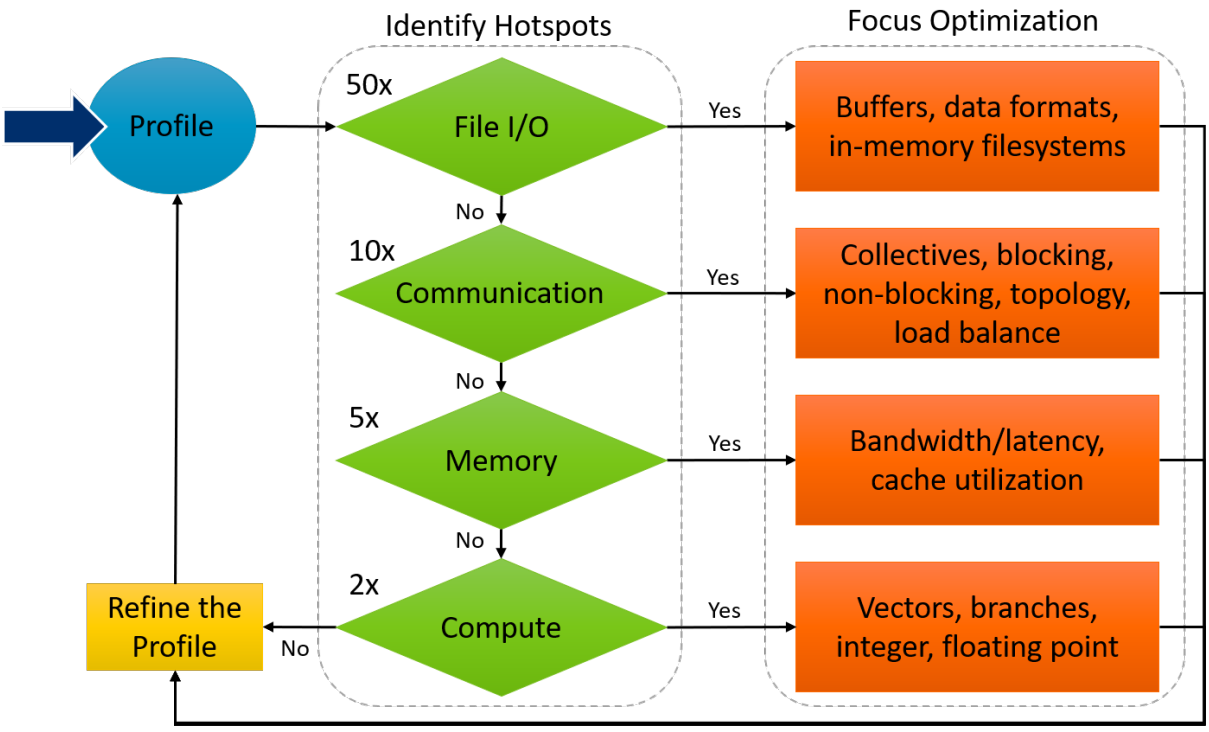
2. Optimize by iteration. Use Arm Forge Professional to iteratively debug and profile your ported application.

   Arm Forge is composed of the Arm DDT debugger and the Arm MAP profiler:

   - Use Arm DDT to debug your code to ensure application correctness. It can be used both in an interactive and non-interactive debugging mode, and optionally, integrated into your CI workflows.
   - Use Arm MAP to profile your code to measure your application performance. MAP collects a broad set of performance metrics, time classification metrics, specific metrics (for example MPI call and message rates, I/O data rates, energy data), and instruction information (hardware counters), to ensure a comprehensive understanding of the performance of your code. MAP also supports custom metrics so you can develope your own set of metrics of interest.

   Use the Arm Forge tools and follow an iterative identification and resolving cycle to optimize application performance:

   Note: The 50x, 10x, 5x, and 2x numbers in the figure below are potential slow down factors that Arm has observed in real-world applications (when that aspect of performance is done incorrectly).



For more information, see the Arm Forge User Guide.

2.3.2 Related information

For more information on optimizing your application on Arm, see:

- Arm porting and tuning guides
- Packages Wiki
- Latest additions to the Arm-v8A architecture
- Arm Compiler for HPC
- Arm Performance Libraries

2.4 Arm Compiler quick reference

2.4.1 Common Compiler Options

For a full list, see Arm C/C++ Compiler options Arm Fortran Compiler options.

2.4.1.1 General

Table 3: General common compiler options

Option	Description
`-o <file>`	Write output to <file>.
`-c`	Only run preprocess, compile, and assemble steps.

Continued on next page

Table 3 – continued from previous page

Option	Description
`-g`	Generate source-level debug information.
`-Wall`	Enable all warnings.
`-w`	Suppress all warnings.
`-fopenmp`	Enable OpenMP.
`-On`	Level of optimization to use (0, 1, 2, 3).
`-Ofast`	Enables aggressive optimization of floating point operations.
`-ffp-contract=(fast|on|off)`	Allow fused floating point operations (eg FMAs).
`-Rpass=\(loop-vectorize|inline\)` `-Rpass-missed=\(loop-vectorize|inline\)` `-Rpass-analysis=\(loop-vectorize|inline\)`	Optimization Remarks is a feature of LLVM compilers that provides you with information about the choices made by the compiler.

2.4.1.2 Fortran

Table 4: General common compiler options

Option	Description
`-cpp`	Preprocess Fortran files. Default for .F, .F90, .F95,...
`-module <path>`	Specifies a directory to place, and search for, module files.
`-Mallocatable=(95|03)`	95: Use preFortran 2003 standard semantics for assignments to allocatables 03: Use Fortran 2003 standard semantics for assignments to allocatables
`-fconvert=<setting>`	Set format for unformatted file access to numerical data to `big-endian`, `little-endian`, `swap` or `native`
`-r8`	Sets default KIND for real and complex declarations, constants, functions, and intrinsics to 64bit (i.e. real (KIND=8)). Unspecified real kinds are evaluated as KIND=8.
`-i8`	Set the default kind for INTEGER and LOGICAL to 64bit (i.e. KIND=8).

2.4.2 Pragmas

Arm C/C++ Compiler supports pragmas to both encourage and suppress auto-vectorization. These pragmas make use of, and extend, the pragma clang loop directives.

```
#pragma clang loop vectorize(assume_safety)
```

Allows the compiler to assume that there are no aliasing issues in a loop.

```
#pragma clang loop unroll_count(_value_)
```

Forces a scalar loop to unroll by a given factor

```
#pragma clang loop interleave_count(_value_)
```

Forces a vectorized loop to be interleaved by a given factor.

For more information about the pragma clang loop directives, see Auto-Vectorization in LLVM on LLVM's website, and Using pragmas to control auto-vectorization on the Arm Developer website.

2.4.3 Related information

- Arm C/C++ Compiler reference guide
- Arm Fortran Compiler reference guide

2.5 Arm Performance Libraries Quick Reference

Compiler command options that control the use of Arm Performance Libraries (ArmPL).

2.5.1 Basic usage

To link Arm Performance Libraries, and provide a serial implementation with 32-bit integers that are optimized for the host CPU, use:

```
-mcpu=native -armpl
```

To tailor Arm Performance Libraries to your application and hardware, there are three decisions to make:

- **Decision 1: Which microarchitecture are you compling for?**

  To compile for the host CPU, use:

  ```
  -mcpu=native
  ```

  To compile for a specific CPU microarchitecture, use:

  ```
  -mcpu=<target-processor>
  ```

- **Decision 2: Do you want an OpenMP-enabled build?**

  To use serial Arm Performance Libraries:

  ```
  -armpl
  ```

  (defaults to no OpenMP)

  To use parallel Arm Performance Libraries

  ```
  -armpl=parallel
  ```

  Note: In Fortran, it is equivalent to specify:

  ```
  -armpl -fopenmp
  ```

- **Decision 3: Do you need 32-bit or 64-bit integers?**

  To use 32-bit integers (the default):

  ```
  -armpl
  ```

  To use 64-bit integers:

  ```
  -armpl=ilp64
  ```

  Note:

  – In Fortran is is equivalent to specify

```
-armpl -i8
```

- Options can be combined, for example:

```
-armpl=ilp64, parallel
```

2.5.2 Porting to Arm Performance Libraries

Most users port their codes without issues because all calls are the same as they have been on previous systems.

Warning: If applicable, ensure you include `armpl.h` rather than, for example, `mkl.h`.

2.5.3 Related information

- Arm Performance Libraries reference guide

2.6 Arm DDT Quick Reference

2.6.1 Workstation or remote interactive sessions

1. Log in to a terminal session and prepare your environment. Load the environment module for Arm Forge.

   Note: The name of the environment variable is determined by the system administrator, please check with them for the environment variable you should use for your system.

2. Either, compile your application (including the `-g` flag), or locate an appropriately pre-compiled binary. To prepare the code and compile without optimizations, include the `-O0` optimization flag on the compile line:

```
mpicc -O0 -g myapp.c -o myapp.exe
```

   Note: Turning off optimization flags is optional. Optimization flags can re-order the code in unexpected ways and make application debugging less intuitive. If a bug only occurs within an optimized binary, keep the relevant optimizations.

3. Launch Arm DDT in interactive mode, use the Express Launch syntax:

```
ddt mpirun -n 8 ./myapp.exe arg1 arg2
```

4. Configure any advanced features for your job, such as memory debugging, in the **Run** dialog.
5. To start debugging, click **Run**.

2.6.2 Sessions on an HPC cluster with a job scheduler

To run Arm DDT in interactive mode, you can use the Arm Forge Remote Client or X-forwarding.

Note: For more information on using Arm Forge in non-interactive mode, see the Arm Forge user guide.

1. Start Arm Remote client, or an X-forwarding session. Either:
   - Arm Remote Client:
     1. Start Arm Remote Client.
     2. If it is your first time using the remote client, add the configuration details for your remote host.
     3. Select the connection from the **Remote Launch** drop down menu.
   - X-forwarding session:
     1. Connect to the remote host system with X-forwarding enabled:

        ```
        ssh -X <remote-host>
        ```

     2. Prepare your environment. Load the environment module for Arm Forge.

        Note: The name of the environment variable is determined by the system administrator, please check with them for the environment variable you should use for your system.

2. On the login node, launch the Arm DDT debugger GUI:

   ```
   ddt &
   ```

3. Either, compile your application (including the `-g` flag), or locate an appropriately pre-compiled binary. To prepare the code and compile without optimizations, include the `-O0` optimization flag on the compile line:

   ```
   mpicc -O0 -g myapp.c -o myapp.exe
   ```

   Note: Turning off optimization flags is optional. Optimization flags can re-order the code in unexpected ways and make application debugging less intuitive. If a bug only occurs within an optimized binary, keep the relevant optimizations.

4. Edit your job script to run the *Reverse Connect* Arm DDT commands `ddt --connect`:

   ```
   ddt --connect mpirun -n 8 ./myapp.exe arg1 arg2
   ```

5. Submit your script
6. When the GUI displays asking whether you want to accept the incoming connection, click **Yes**.
7. Configure any advanced features for your job, such as memory debugging, in the **Run** dialog.
8. To start debugging, click **Run**.

2.6.3 Related information

- Arm Forge user guide

2.7 Arm MAP Quick Reference

2.7.1 Run Arm MAP on a workstation or remote interactive session

To run Arm MAP in non-interactive (*offline*) mode:

1. Log in to a terminal session and prepare your environment. Load the environment module for Arm Forge.

   Note: The name of the environment variable is determined by the system administrator, please check with them for the environment variable you should use for your system.

2. Prepare the code and compile with optimizations:

```
mpicc -O3 -g myapp.c -o myapp.exe
```

3. Generate a profile with the Express Launch syntax:

```
map --profile mpirun -n 8 ./myapp.exe arg1 arg2
```

   Note: In Arm MAP 19.x+ versions, you can profile python applications (sequential and parallel using mpi4py). To profile python applications, use:

```
map --profile python ./myapp.exe
```

4. Open the resulting `.map` file:

```
map ./myapp_8p_1n_YYYY-MM-DD_HH-MM.map
```

   Note: The `.map` file can be opened anywhere, no compute node allocation is needed. However, you must tell the GUI where to look for the program source files.

2.7.2 Strategy

Serial comparison

Profile benchmark run, with codes compiled with both compilers, using map or perf.

Scale comparison

- Find a suitable benchmark to do scaling runs across various numbers of cores (weak and strong scaling).
- Consider placement of tasks to minimise interference between tasks.

Look at whether hot sections have similar work between compilers

- To locate sections of code that consume most of the runtime, use Arm MAP profile runs.
- Are the locations the same for both compilers? Consider whether these sections be improved.
- Report major runtime differences across compilers to the Arm Compiler team.

Search for compiler specific sections/intrinsics

- Search the source for compiler-specific intrinsics, or pragmas, and consider if these can be ported to Arm Compiler or platform-compatible versions.
- If AVX512 vector instructions are present, consider converting these.

2.7.3 Related information

- Arm Forge user guide

2.8 Arm Performance Reports Quick Reference

To start Arm Performance Reports:

1. Log in to a terminal session and prepare your environment. Load the environment module for Arm Performance Reports.

   Note: The name of the environment variable is determined by the system administrator, please check with them for the environment variable you should use for your system.

2. Prepare the code and compile with optimizations:

```
mpicc -O3 -g myapp.c -o myapp.exe
```

3. Generate your performance report with the Express Launch syntax:

```
perf-report mpirun -n 8 ./myapp.exe arg1 arg2
```

4. Open the resulting `.html` or `.txt` file.

2.8.1 Related information

- Arm Performance Reports user guide

2.9 Porting and Tuning Recipes

For detailed instructions on how to build many common scientific applications, benchmarks, and libraries using the Arm HPC tools suite, see the Porting and Tuning web page. If you do not find the recipe you are looking for here, try the community-driven Packages wiki.

If you still cannot find the recipe you are looking for, please contact support.

CHAPTER

THREE

COMPILER MIGRATION GUIDES

To assist Fortran developers using the gfortran, ifort, and pgfortran compilers, this chapter provides an overview of armflang, and dicusses the differences between each compiler and armflang.

3.1 Overview of Arm Fortran Compiler (armflang)

This topic introduces Arm Fortran Compiler. For more information on Arm Fortran Compiler, see the Arm Fortran Compiler Reference Guide or Arm Fortran Compiler product web page.

3.1.1 Invoking Arm Fortran Compiler

To invoke Arm Fortran Compiler for preprocessing, compilation, assembly and linking, use `armflang`.

To access compiler details and documentation, use:

Table 1: GNU and Arm Compiler commands

	Arm
Version details	`armflang --version`
Help and documentation	`armflang --help` `man armflang`

3.1.2 Supported file types

The extensions `.f90`, `.f95`, `.f03`, and `.f08` are used for modern, free-form source code that conforms to the Fortran 90, Fortran 95, Fortran 2003, or Fortran 2008 standards.

The extensions `.F90`, `.F95`, `.F03`, and `.F08` are used for source code that requires preprocessing, and which is preprocessed automatically.

It is possible to instruct `armflang` to preprocess source irrespective of file extension by using the `-cpp` flag, as detailed in the next section.

The `.f` and `.for` extensions are typically used for older, fixed-form code such as FORTRAN77.

3.1.3 Optimization remarks

Optimization remarks are described in *Optimize*.

For more information on optimization remarks, see the Fortran and C/C++ compiler reference guides.

3.1.4 Arm hardware flags

GCC and Arm Compiler, have three hardware compiler flags in common: `-march`, `-mtune`, and `-mcpu`:

- `-march=X`: Tellsthe compiler that X istheminimalarchitecture the binary must run on.The compiler is free to use architecture-specificinstructions. This flag behaves differently on Arm and x86. On Arm, `-march` does not override `-mtune`, but on x86 `-march` does override both `-mtune` and `-mcpu`.
- `-mtune=X`: Tells the compiler tooptimize formicroarchitecture X, but does not allow the compiler tochange the ABI or make assumptions aboutavailable instructions. This flag has the more-or-less the same meaning on Arm and x86.
- `-mcpu=X`: On Arm, this flag is a combination of `-march` and `-mtune`. It simultaneously specifies the target architecture and optimizes for a given microarchitecture. On x86, this flag is a deprecated synonym for `-mtune`.

GCC and Arm Compiler support passing the special parameter value `native` to these flags. The `native` value tells the compiler to automatically detect the architecture or microarchitecture of the machine on which the compiler is executing.

Note: Arm Compiler does not support the use of `-march=native`. To aid portability, GCC on AArch64 does support the use of `-march=native`.

These flags control binary code generation, so the correct use of these flags can dramatically improve runtime performance. If you are not cross compiling, the simplest and easiest method to get the best performance on Arm, with both GCC and LLVM-based compilers, is to only use `-mcpu=native`, and actively avoid using `-mtune` or `-march`.

Note: Automatic detection of the architecture and processor is independent of the optimization level denoted by the `-On` flag and similar flags, as detailed in the **Commonly used flags**and **Optimization compiler options** sections in each compiler guide.

3.1.5 Optimized math functions with Arm Performance Libraries

Arm Performance Libraries (ArmPL) provide the following optimized standard core math libraries for high-performance computing applications on Arm processors:

- BLAS - Basic Linear Algebra Subprograms (including XBLAS which is extended precision BLAS).
- LAPACK - a comprehensive package of higher level linear algebra routines.
- FFT - a set of Fast Fourier Transform routines for real and complex data.
- Math routines - optimized implementations of common maths intrinsics (on by default in Arm Performance Libraries versions 19.1+).
- Auto-vectorization of Fortran math intrinsics (disable this with `-fno-simdmath`).

Arm Compiler for HPC 19.0+ introduces the `-armpl` compiler flag that simplifies using Arm Performance Libraries. This new flag provides a straightforward interface for selecting thread-parallelism and architectural tuning. Arm Performance Libraries also provides improved Fortran math intrinsics with auto-vectorization.

The `-armpl` and `-mcpu` flags enable the compiler to find appropriate Arm Performance Libraries header files during compilation, and appropriate libraries during linking. Both flags are required to achieve the best results.

Note: If your build process compiles and links as two separate steps, please ensure that you add the same `-armpl` and `-mcpu` options to both.

For more information on Arm Performance Libraries, see Arm Performance Libraries.

3.1.6 Compiler directives

Directives are used to provide additional information to the compiler, and to control the compilation of specific code blocks, for example, loops. The Arm Fortran Compiler supports the following common directives:

Table 2: Arm Fortran Compiler directives

Directive	Usage	Description
`IVDEP`	`!DIR$ IVDEP` `<do loop>`	A generic directive which forces the compiler to ignore any potential memory dependencies of iterative loops, and to vectorize the loop.
`OMP SIMD`	`!$OMP SIMD` `<do loop>`	An OpenMP directive to indicate that a loop can be transformed into a Single instruction, multiple data (SIMD) loop. **Note:** • -fopenmp must be set. • There is currently no support for `OMP SIMD` clauses.
`VECTOR ALWAYS`	`!DIR$ VECTOR ALWAYS` `<do loop>`	Forces the compiler to vectorize a loop, and ignores any potential performance implications. **Note:** The loop must be vectorizable.
`NOVECTOR`	`!DIR$ NOVECTOR` `<do loop>`	Disables the vectorization of a loop.
`UNROLL`	`!DIR$ UNROLL` `<do loop>`	Instructs the compiler optimizer to unroll a DO loop when optimization is enabled with the compiler optimization flags `-O2` or higher.

3.1.7 Generating position independent code with fPIC on AArch64

The generation of position independent code is typically required for building shared libraries. Supplying the command line flag `-fPIC` at compile time instructs `armflang` to generate position independent code. This is broadly consistent with the behavior of other compilers.

Note: PGI compilers do not differentiate between `-fPIC` and `-fpic` which are documented as interchangeable on x86 architectures. For more information on migrating from the PGI `pgfortran` compiler to Arm Compiler, see *armflang for pgfortran Developers*.

However, while the use of `-fpic` is often interchangeable with `-fPIC` on x86, not the case with GCC on AArch64. `-fpic` uses an address mode with a smaller number of entries in the Global Offset Table. As a result, use of `-fpic` must not be considered to be portable between x86_64 and AArch64 architectures.

3.1.8 Allocating stack variables

- **Thread-safe recursion**

  The `-frecursive` flag allocates all local variables on the stack. This allows thread-safe recursion and is applied implicitly for source compiled with the `-fopenmp` flag.

  Use the `-frecursive` options when compiling a procedure that:

  – Has no OpenMP elements and is not compiled using the `-fopenmp` flag.

  – Is called from within an OpenMP parallel region in source, compiled with the `-fopenmp` flag.

- **Automatic arrays**

  This feature of Fortran 2003 allows allocatable arrays to be allocated, and dynamically resized without the need for calls to `ALLOCATE` and `DEALLOCATE`. Automatic arrays are stored on the heap, regardless of the `-frecursive` flag, unless `-fstack-arrays` is specified.

Note: Use of the stack for local variables and automatic arrays can have implications for the stack size. To avoid running out of stack, it might be necessary to increase the stack size. For example, to remove the stack-size limit, enter `ulimit -s unlimited` at the command line.

3.1.9 Line lengths

The Fortran standard for free-form source (from Fortran90 onwards) sets a maximum line length of 132 characters. Statements can be broken over a maximum of 255 lines using the ampersand (&) continuation mark. Many compilers permit the use of lines significantly longer than 132 characters.

`armflang` limits line lengths to 2100 characters and generates a compile time error if there are source lines, including comments, longer than 2100 characters. To compile with Arm Fortran Compiler, you must ensure that all source lines are within this limit.

Note: Arm Compiler versions earler than 19.1 limited line lengths to 264 characters. Using compiler macros in versions earlier than 19.1 can lead to the generation of source lines longer than 264 characters at compile time.

3.1.10 Language extensions

There are a number of common extensions to the Fortran language which are typically supported by many existing compilers, generally for legacy reasons, including armflang. In many cases, the required functionality is now part of the language standard, even though it uses a different syntax. The following table shows common language extensions and their standards-compliant alternatives, where available.

Table 3: Fortran language extensions and their standard-compliant alternatives

Extension	Purpose	Standard-compliant alternative	Notes
`IARGC()`	Function call which returns the number of command line arguments supplied	`COMMAND_ARGUMENT_COUNT()`	Introduced with 2003 standard.

Continued on next page

Table 3 – continued from previous page

Extension	Purpose	Standard-compliant alternative	Notes
`GETARG(pos, arg)`	Subroutine call which returns the pos-th argument passed at the command line when the programme was invoked, and returns it as arg.	`GET_COMMAND_ARGUMENT(pos,arg, len, status)`	Introduced with 2003 standard. arg, len, and status are optional arguments.
`GETENV(name, arg)`	Subroutine call which returns the environment variable name as arg.	`GET_ENVIRONMENT_VARIABLE(name, arg, len, status, trim_name)`	Introduced with 2003 standard. arg, len, and status are optional arguments.
`GETCWD(dir, status)`	Subroutine call which returns the current working directory as dir. status is an optional argument which returns 0 on success, and a nonzero error code when not successful.	No equivalent functionality at present.	No equivalent functionality in the 2003 standard.

Some commonly supported language extensions are not supported in armflang:

Table 4: unsupported language extensions in Arm Compiler

Extension	Purpose	`armflang` equivalent	Notes
`ISNAN(x)`	Logical function returns `.TRUE.` if the REAL argument x is Not-a-Number (NaN).	`IEEE_IS_NAN(x)`	Introduced with 2003 standard. Requires IEEE_ARITHMETIC module.

3.1.11 Pre-defined macros

armflang has the following compiler and machine-specific predefined processor macros:

Table 5: Pre-defined macros

Macro	Value	Purpose
`__aarch64__`	1	Selection of architecture-dependent source at compile time.
`__ARM_ARCH`	8	Selection of architecture-dependent source at compile time.
`__FLANG`	1	Selection of compiler-dependent source at compile time.
`__clang__`	1	Selection of compiler-dependent source at compile time.

`__clang_version__`
`__clang_major_`
`__clang_minor__`
`__clang_patchlevel__`
	“7.1.0”
	7
	1
	0
		Underlying Clang version details.

3.1.12 Detailed compiler options

Passing the flag `-###` to `armflang` causes it to print the complete options used at each stage of the compilation, without executing them.

3.1.13 Related information

Standards compliance

- Arm provides full support for Fortran 2003 and prior standards, and partial support for Fortran 2008.

Additional information

- Arm Fortran Compiler Language Reference Guide
- Arm Fortran statement support
- Supported Fortran Intrinsics
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

Getting help

Contact Arm HPC support

3.2 armflang for gfortran Developers

The reference versions used in this guide are:

- GCC (`gfortran` 8.2.0)
- Arm Fortran Compiler 19.1

3.2.1 Invoking the compiler

The following table gives the equivalent GCC and Arm Compiler commands to invoke Arm Fortran Compiler for preprocessing, compilation, assembly and linking.

Table 6: Invoking the compiler

GCC	Arm
`gfortran <options> <filename>`	`armflang <options> <filename>`

The following table gives the equivalent GCC and Arm Compiler commands to access compiler details and documentation.

Table 7: Accessing version information and documentation

	GCC	Arm
Version details	`gfortran --version`	`armflang --version`
Help and documentation	`gfortran --help` `man gfortran`	`armflang --help` `man armflang`

3.2.2 Commonly used flags

The following table summarizes some of the compiler options most commonly used with GCC and gives the equivalent options to use with the Arm Fortran Compiler:

Table 8: GCC and Arm Compiler equivalent options

GCC	Arm	Description
`-c`	`-c`	Run only preprocess, compile and assemble steps.
`-o filename`	`-o filename`	Write to output filename.
`-g`	`-g`	Generate source level debug information.
`-Wall` `-warn none`	`-Wall` `-w`	Enable all warnings. Suppress all warnings.
`-cpp` `-nocpp`	`-cpp` `-nocpp`	Preprocess Fortran source files. Do not preprocess Fortran source files. **Note:** By default, source files with the extensions, .F, .F90, .F95, .F03 and .F08 are preprocessed. -cpp forces the compiler to use the processor for all source files.
`-fopenmp`	`-fopenmp`	Enable OpenMP. See OpenMP support for Arm Fortran Compiler.
`-J path` `-I path`	`-module path`	Specifies a directory to place and search for module files.
`-On`	`-On`	Level of optimization to use, where n=0,1,2,3. See Optimization with the Arm Fortran Compiler.

Continued on next page

Table 8 – continued from previous page

GCC	Arm	Description
`-frealloc-lhs` `-fno-realloc-lhs`	`-frealloc-lhs` `-fno-realloc-lhs`	`-frealloc-lhs` uses Fortran 2003 standard semantics for assignments to allocatables. An allocatable object on the left-hand side of an assignment is (re)allocated to match the dimensions of the right-hand side. `-fno-realloc-lhs` uses pre-Fortran 2003 standard semantics for assignments to allocatables. The left-hand side of an allocatable assignment is assumed to be allocated with the correct dimensions. Incorrect behavior can occur if the left-hand side is not allocated with the correct dimensions. **Note:** • Default behavior in `armflang` versions 19.0+ supports the Fortran 2003 standard feature: (re)allocation on assignment. By default, earlier versions of `armflang` do not support this feature.

`-byteswapio`	`-fconvert=big-endian`
	`-fconvert=little-endian`
	`-fconvert=native`
	`-fconvert=swap`
		Swap the byte ordering for unformatted file access of numeric data to big endian from little endian, or the other way round.
		`armflang` also provides options to set the byte order explicitly to big endian, little endian, or native.
		Note: Default behavior is native.

`-Dmacro=value`	`-Dmacro=value`	Set macro to value.
`-Ldirectory`	`-Ldirectory`	Add directory to the include search path.
`-llib`	`-llib`	Search for the library lib when linking.
`-fdefault-real-8`	`-r8`	Set the default KIND for real and complex declarations, constants, functions, and intrinsics to 64bit (such as real (KIND=8)). Unspecified real kinds are evaluated as KIND=8.
`-fdefault-integer-8`	`-i8`	Set the default kind for INTEGER and LOGICAL to 64bit (KIND=8).

Continued on next page

Table 8 – continued from previous page

GCC	Arm	Description
`-frecord-marker`	N/A	Length of record markers for unformatted files. Can be 4 or 8. Default is 4. However, older versions of gfortran default to 8. `armflang` uses a record marker of length 4 bytes.
`-fpic` `-fPIC`	`-fpic` `-fPIC`	Generate position independent code. For more information on the use of `-fpic` and `-fPIC` on AArch64, see Note about building Position Independent Code PIC on AArch64.

3.2.3 Optimization compiler options

The following table summarizes some of the most commonly used compiler options provided by GCC and Arm Fortran Compiler:

Table 9: Commonly used optimization options

Description	Syntax	Notes
Basic optimization switches	`-On`	Optimization level where n=0,1,2,3. There is no direct correlation between the optimizations employed at each level between the two compilers. At n=0, the compiler performs little or no optimization. At n=3, the compiler performs aggressive optimization. At n=2 and n=3, debug information might not be satisfactory because the mapping of object code to source code is not always clear and the compiler can perform optimizations that cannot be described in the debug information.
Aggressive optimization	`-Ofast`	Enables all `-O3` optimizations from level 3 and performs aggressive optimization, which can violate strict language compliance. With `armflang`, this is equivalent to: • Setting: `-O3 -Menable-no-infs -Menable-no-nans -Menable-unsafe-fp-math -fno-signed-zeros -freciprocal-math -fno-trapping-math -ffp-contract=fast -ffast-math -ffinite-math-only -fstack-arrays` • Unsetting: `-fmath-errno`

Continued on next page

Table 9 – continued from previous page

Description	Syntax	Notes
Fused floating-point operations	`-ffp-contract=fast/off`	Instructs `armflang` to perform fused floating-point operations, such as fused multiply adds. • fast = always on (default for -O1 and above) • off = never
Reduced floating-point precision	`-ffast-math -funsafe-math-optimizations`	Allows aggressive, lossy, floating-point optimizations. Allows reciprocal optimizations and does not honor trapping or signed zero.
Finite maths	`-finite-maths-only`	Enable optimizations that ignore the possibility of NaNs and Infs.

3.2.4 Language extensions

Some commonly supported language extensions are not supported in armflang:

Table 10: GCC and Arm Compiler options

Extension	Purpose	`armflang` equivalent	Notes
`ISNAN(x)`	Logical function returns `.TRUE.` if the REAL argument x is Not-a-Number (NaN).	`IEEE_IS_NAN(x)`	Introduced with 2003 standard. Requires IEEE_ARITHMETIC module.

3.2.5 Fortran formatted I/O

`armflang` adopts the Linux/UNIX convention of using the line-feed character (`'LF'`, `'0x0A'`, `'\n'`) as the record terminator in formatted I/O, for both read and write operations. However, gfortran also accepts the carriage return character (`'CR'`, `'0x0D'`, `'\r'`) to denote the end of records on read operations. This can lead to differing behavior between `armflang` and gfortran builds when accessing files containing 'CR' characters, such as text files generated on Windows platforms, which use 'CR-LF' to denote the end of lines.

3.2.6 Related information

Standards compliance

- Arm provides full support for Fortran 2003 and prior standards, and partial support for Fortran 2008.

Additional information

- Arm Fortran Compiler Language Reference Guide
- Arm Fortran statement support
- Supported Fortran Intrinsics
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

Getting help

Contact Arm HPC support

3.3 armflang for ifort Developers

The reference versions used in this guide are:

- Intel Fortran Compiler 17.0.1
- Arm Fortran Compiler 19.1

3.3.1 Invoking the compiler

The following table gives the equivalent Intel and Arm Compiler commands to invoke the Fortran compiler for preprocessing, compilation, assembly and linking.

Table 11: Invoking the compiler

Intel	Arm
`ifort <options> <filename>`	`armflang <options> <filename>`

The following table gives the equivalent Intel and Arm Compiler commands to access compiler details and documentation.

Table 12: Accessing version information and documentation

	Intel	Arm
Version details	`ifort --version`	`armflang --version`
Help and documentation	`ifort --help` `man ifort`	`armflang --help` `man armflang`

3.3.2 Commonly used flags

The following table summarizes some of the compiler options most commonly used with the Intel Fortran compiler and gives the equivalent options to use with the Arm Fortran Compiler:

Table 13: Intel and Arm Compiler equivalent options

Intel	Arm	Description
`-c`	`-c`	Run only preprocess, compile and assemble steps.
`-o filename`	`-o filename`	Write to output filename.
`-g`	`-g`	Generate source level debug information.
`-warn all` `-warn none`	`-Wall` `-w`	Enable all warnings. Suppress all warnings.
`-fpp` `-nofpp`	`-cpp` `-nocpp`	Preprocess Fortran source files. Do not preprocess Fortran source files. **Note:** By default, source files with the extensions, .F, .F90, .F95, .F03 and .F08 are preprocessed. -cpp forces the compiler to use the processor for all source files.

Continued on next page

Table 13 – continued from previous page

Intel	Arm	Description
`-qopenmp`	`-fopenmp`	Enable OpenMP. See OpenMP support for Arm Fortran Compiler.
`-module path`	`-module path`	Specifies a directory to place and search for module files.
`-On`	`-On`	Level of optimization to use, where n=0,1,2,3. See Optimization with the Arm Fortran Compiler.
`-standard-realloc-lhs` `-nostandard-realloc-lhs`	`-frealloc-lhs` `-fno-realloc-lhs`	`-frealloc-lhs` uses Fortran 2003 standard semantics for assignments to allocatables. An allocatable object on the left-hand side of an assignment is (re)allocated to match the dimensions of the right-hand side. `-fno-realloc-lhs` uses pre-Fortran 2003 standard semantics for assignments to allocatables. The left-hand side of an allocatable assignment is assumed to be allocated with the correct dimensions. Incorrect behavior can occur if the left-hand side is not allocated with the correct dimensions. **Note:** • Default behavior in `armflang` versions 19.0+ supports the Fortran 2003 standard feature: (re)allocation on assignment. By default, earlier versions of `armflang` do not support this feature.

`-convert big-endian`
`-convert little-endian`
`-convert native`
	`-fconvert=big-endian`
	`-fconvert=little-endian`
	`-fconvert=native`
	`-fconvert=swap`
		Swap the byte ordering for unformatted file access of numeric data to big endian from little endian, or the other way round.
		`armflang` also provides options to set the byte order explicitly to big endian, little endian, or native.
		Note: Default behavior is native.

`-Dmacro=value`	`-Dmacro=value`	Set macro to value.
`-Ldirectory`	`-Ldirectory`	Add directory to the include search path.
`-llib`	`-llib`	Search for the library lib when linking.

Continued on next page

Table 13 – continued from previous page

Intel	Arm	Description
`-real-size 64` `-r8`	`-r8`	Set the default KIND for real and complex declarations, constants, functions, and intrinsics to 64bit (such as real (KIND=8)). Unspecified real kinds are evaluated as KIND=8.
`-integer-size 64` `-i8`	`-i8`	Set the default kind for INTEGER and LOGICAL to 64bit (KIND=8).
`-fpic` `-fPIC`	`-fpic` `-fPIC`	Generate position independent code. For more information on the use of `-fpic` and `-fPIC` on AArch64, see Note about building Position Independent Code PIC on AArch64.

3.3.3 Optimization compiler options

The following table summarizes some of the most commonly used compiler options provided by the Intel and Arm Fortran Compiler:

Table 14: Commonly used optimization options

Description	Syntax	Notes
Basic optimization switches	`-On`	Optimization level where n=0,1,2,3. There is no direct correlation between the optimizations employed at each level between the two compilers. At n=0, the compiler performs little or no optimization. At n=3, the compiler performs aggressive optimization. At n=2 and n=3, debug information might not be satisfactory because the mapping of object code to source code is not always clear and the compiler can perform optimizations that cannot be described in the debug information.
Aggressive optimization	`-Ofast`	Enables all `-O3` optimizations from level 3 and performs aggressive optimization, which can violate strict language compliance. With `armflang`, this is equivalent to: • Setting: `-O3 -Menable-no-infs -Menable-no-nans -Menable-unsafe-fp-math -fno-signed-zeros -freciprocal-math -fno-trapping-math -ffp-contract=fast -ffast-math -ffinite-math-only -fstack-arrays` • Unsetting: `-fmath-errno`

Continued on next page

Table 14 – continued from previous page

Description	Syntax	Notes
Fused floating-point operations	`-ffp-contract=fast/off`	Instructs `armflang` to perform fused floating-point operations, such as fused multiply adds. • fast = always on (default for -O1 and above) • off = never
Reduced floating-point precision	`-ffast-math -funsafe-math-optimizations`	Allows aggressive, lossy, floating-point optimizations. Allows reciprocal optimizations and does not honor trapping or signed zero.
Finite maths	`-finite-maths-only`	Enable optimizations that ignore the possibility of NaNs and Infs.

3.3.4 Handling backslash characters

The default behavior in `armflang` is for backlash () to be treated as a special character; this is not the case for ifort.

To make `armflang` match ifort's behavior, use `-fno-backslash`.

To make ifort match armflang's behavior use `-assume bscc`.

3.3.5 Related information

Standards compliance

- Arm provides full support for Fortran 2003 and prior standards, and partial support for Fortran 2008.

Additional information

- Arm Fortran Compiler Language Reference Guide
- Arm Fortran statement support
- Supported Fortran Intrinsics
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

Getting help

Contact Arm HPC support

3.4 armflang for pgfortran Developers

The reference versions used in this guide are:

- PGI Fortran Compiler 18.5
- Arm Fortran Compiler 19.1

3.4.1 Invoking the compiler

The following table gives the equivalent PGI and Arm Compiler commands to invoke the Fortran compiler for preprocessing, compilation, assembly and linking.

Table 15: Invoking the compiler

PGI	Arm
`pgfortran <options> <filename>`	`armflang <options> <filename>`

The following table gives the equivalent PGI and Arm Compiler commands to access compiler details and documentation.

Table 16: Accessing version information and documentation

	PGI	Arm
Version details	`pgfortran --version`	`armflang --version`
Help and documentation	`pgfortran --help` `man pgFortran`	`armflang --help` `man armflang`

3.4.2 Commonly used flags

The following table summarizes some of the compiler options most commonly used with the PGI Fortran compiler and gives the equivalent options to use with the Arm Fortran Compiler:

Table 17: PGI and Arm Compiler equivalent options

PGI	Arm	Description
`-c`	`-c`	Run only preprocess, compile and assemble steps.
`-o filename`	`-o filename`	Write to output filename.
`-g`	`-g`	Generate source level debug information.

`-Minform=inform`
`-w`
`-silent`
`-Minform=severe`
	`-Wall`
	`-w`
		Enable all warnings.
		Suppress all warnings.

`-Mpreprocess`	`-cpp` `-nocpp`	Preprocess Fortran source files. Do not preprocess Fortran source files. **Note:** By default, source files with the extensions, .F, .F90, .F95, .F03 and .F08 are preprocessed. -cpp forces the compiler to use the processor for all source files.
`-mp`	`-fopenmp`	Enable OpenMP. See OpenMP support for Arm Fortran Compiler.
`-module path`	`-module path`	Specifies a directory to place and search for module files.
`-On`	`-On`	Level of optimization to use, where n=0,1,2,3. See Optimization with the Arm Fortran Compiler.

Continued on next page

Table 17 – continued from previous page

PGI	Arm	Description
`-Mallocatable=03` `-Mallocatable=95`	`-frealloc-lhs` `-fno-realloc-lhs`	`-frealloc-lhs` uses Fortran 2003 standard semantics for assignments to allocatables. An allocatable object on the left-hand side of an assignment is (re)allocated to match the dimensions of the right-hand side. `-fno-realloc-lhs` uses pre-Fortran 2003 standard semantics for assignments to allocatables. The left-hand side of an allocatable assignment is assumed to be allocated with the correct dimensions. Incorrect behavior can occur if the left-hand side is not allocated with the correct dimensions. **Note:** • Default behavior in `armflang` versions 19.0+ supports the Fortran 2003 standard feature: (re)allocation on assignment. By default, earlier versions of `armflang` do not support this feature. • In version 19.0 of armflang, `-Mallocatable=03` and `-Mallocatable=95` are supported instead of `-frealloc-lhs` and `-fno-realloc-lhs`, respectively. The `-Mallocatable` option continues to be supported in the armflang, but from versions 19.1+ the documentation refers to the `-frealloc-lhs` and `-fno-realloc-lhs` nomenclature.

`-byteswapio`	`-fconvert=big-endian`
	`-fconvert=little-endian`
	`-fconvert=native`
	`-fconvert=swap`
		Swap the byte ordering for unformatted file access of numeric data to big endian from little endian, or the other way round.
		`armflang` also provides options to set the byte order explicitly to big endian, little endian, or native.
		Note: Default behavior is native.

Continued on next page

Table 17 – continued from previous page

PGI	Arm	Description
`-Dmacro=value`	`-Dmacro=value`	Set macro to value.
`-Ldirectory`	`-Ldirectory`	Add directory to the include search path.
`-llib`	`-llib`	Search for the library lib when linking.
`-r8`	`-r8`	Set the default KIND for real and complex declarations, constants, functions, and intrinsics to 64bit (such as real (KIND=8)). Unspecified real kinds are evaluated as KIND=8.
`-i8`	`-i8`	Set the default kind for INTEGER and LOGICAL to 64bit (KIND=8).
`-fpic` `-fPIC`	`-fpic` `-fPIC`	Generate position independent code. With both `armflang` and `pgf90`, `-fpic` and `-fPIC` are equivalent. For more information on the use of `-fpic` and `-fPIC` on AArch64, see Note about building Position Independent Code PIC on AArch64.

3.4.3 Optimization compiler options

The following table summarizes some of the most commonly used compiler options provided by the PGI and Arm Fortran Compiler:

Table 18: Commonly used optimization options

Description	Syntax	Notes
Basic optimization switches	`-On`	Optimization level where n=0,1,2,3. There is no direct correlation between the optimizations employed at each level between the two compilers. At n=0, the compiler performs little or no optimization. At n=3, the compiler performs aggressive optimization. At n=2 and n=3, debug information might not be satisfactory because the mapping of object code to source code is not always clear and the compiler can perform optimizations that cannot be described in the debug information.

Continued on next page

Table 18 – continued from previous page

Description	Syntax	Notes
Aggressive optimization	`-Ofast`	Enables all `-O3` optimizations from level 3 and performs aggressive optimization, which can violate strict language compliance. With `armflang`, this is equivalent to: • Setting: `-O3 -Menable-no-infs -Menable-no-nans -Menable-unsafe-fp-math -fno-signed-zeros -freciprocal-math -fno-trapping-math -ffp-contract=fast -ffast-math -ffinite-math-only -fstack-arrays` • Unsetting: `-fmath-errno`
Fused floating-point operations	`-ffp-contract=fast/off`	Instructs `armflang` to perform fused floating-point operations, such as fused multiply adds. • fast = always on (default for -O1 and above) • off = never
Reduced floating-point precision	`-ffast-math -funsafe-math-optimizations`	Allows aggressive, lossy, floating-point optimizations. Allows reciprocal optimizations and does not honor trapping or signed zero.
Finite maths	`-finite-maths-only`	Enable optimizations that ignore the possibility of NaNs and Infs.

3.4.4 Related information

Standards compliance

- Arm provides full support for Fortran 2003 and prior standards, and partial support for Fortran 2008.

Additional information

- Arm Fortran Compiler Language Reference Guide
- Arm Fortran statement support
- Supported Fortran Intrinsics
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

Getting help

Contact Arm HPC support

CHAPTER

FOUR

CODING FOR NEON

The topics in this chapter discuss coding for Neon.

4.1 Introducing NEON for Arm®v8-A

This guide introduces Arm Neon technology, the Advanced SIMD (Single Instruction Multiple Data) architecture extension for implementation of the Armv8–A architecture profile.

Neon technology provides a dedicated extension to the Instruction Set Architecture, providing additional instructions that can perform mathematical operations in parallel on multiple data streams.

Neon can be used to accelerate the core algorithms used in many compute-intensive applications, and is commonly used by core maths libraries. Neon can also accelerate signal processing algorithms and functions to speed up applications such as audio and video processing, voice and facial recognition, computer vision, and deep learning.

As an application developer, there are a number of ways you can make use of Neon technology:

- Neon-enabled open source libraries such as Arm Performance Libraries or Ne10 provide one of the easiest ways to take advantage of Neon. Another example is FFTW.
- Auto-vectorization features in your compiler can automatically optimize your code to take advantage of Neon.
- Neon intrinsics are function calls that the compiler replaces with appropriate Neon instructions. This gives you direct, low-level access to the exact Neon instructions you want, from C/C++ code.
- Hand-coded Neon assembler can be an alternative approach for experienced programmers.

4.1.1 Before you begin

If you are completely new to Arm technology, you can read the Cortex-A Series Programmer's Guide for general information about the Arm architecture and programming guidelines.

The information in this guide relates to Neon for Arm®v8.

If you are hand-coding in assembler for a specific device, refer to the Technical Reference Manual (TRM) for that processor to see the microarchitectural details that can help you maximize performance. For some processors, Arm also publishes a Software Optimization Guide which might be of use. For example, see the Arm Cortex-A75 Technical Reference Manual and the Arm Cortex-A75 Software Optimization Guide.

4.1.2 Data processing methodologies

When processing large sets of data, a major performance-limiting factor is the amount of CPU time taken to perform data processing instructions. This CPU time depends on the number of instructions it takes to deal with the entire data set. And the number of instructions depends on how many items of data each instruction can process.

4.1.2.1 Single Instruction Single Data (SISD)

Most Arm instructions are **Single Instruction Single Data** (SISD). Each instruction performs its specified operation on a single datum. Processing multiple items requires multiple instructions. For example, to perform four addition operations, requires four instructions to add values from four pairs of registers:

```
ADD x0, x0, x5
ADD x1, x1, x6
ADD x2, x2, x7
ADD x3, x3, x8
```

This method is relatively slow and it can be difficult to see how different registers are related. To improve performance and efficiency, media processing is often off-loaded to dedicated processors such as a **Graphics Processing Unit** (GPU) or **Media Processing Unit** which can process more than one data value with a single instruction.

If the values you are dealing with are smaller than the maximum bit size, that extra potential bandwidth is wasted with SISD instructions. For example, when adding 8-bit values together, each 8-bit value needs to be loaded into a separate 64-bit register. Performing large numbers of individual operations on small data sizes does not use machine resources efficiently because processor, registers, and data path are all designed for 64-bit calculations.

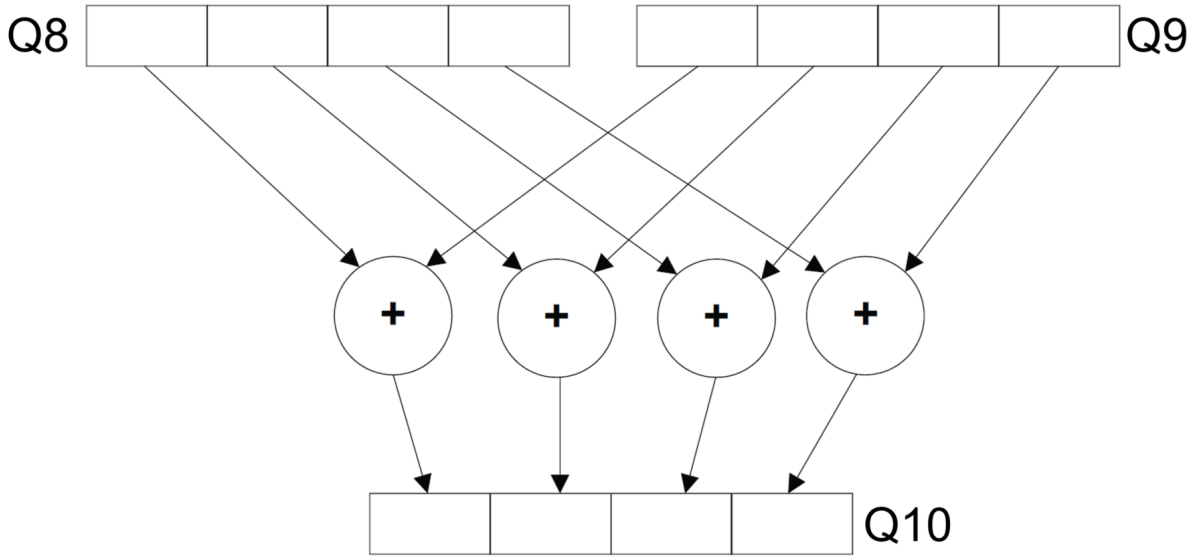
4.1.2.2 Single Instruction Multiple Data (SIMD)

Single Instruction Multiple Data (SIMD) instructions perform the same operation simultaneously for multiple items. These items are packed as separate lanes in a larger register. For example, the following instruction adds four pairs of single-precision (32-bit) values together. However, in this case, the values are packed as separate lanes in two pairs of 128-bit registers. Each lane in the first source register is then added to the corresponding lane in the second source register, before being stored in the destination register:

```
ADD Q10.4S, Q8.4S, Q9.4S
```

In the above example, this operation adds two 128-bit (quadword) registers, Q8 and Q9, and stores the result in Q10. Each of the four 32-bit lanes in each register is added separately. There are no carries between the lanes.

This single instruction operates on all data values in the large register at the same time:



Performing the four operations with a single SIMD instruction is faster than with four separate SISD instructions. The diagram shows 128-bit registers each holding four 32-bit values, but other combinations are possible for Neon registers:

- Four 32-bit, eight 16-bit, or sixteen 8-bit integer data elements can be operated on simultaneously in a single 128-bit register.
- Two 32-bit, four 16-bit, or eight 8-bit integer data elements can be operated on simultaneously in a single 64-bit register.

Media processors, such as used in mobile devices, often split each full data register into multiple sub-registers and perform computations on the sub-registers in parallel. If the processing for the data sets are simple and repeated many times, SIMD can give considerable performance improvements. It is also beneficial for:

- Audio, video, and image processing codecs.
- 2D graphics based on rectangular blocks of pixels.
- 3D graphics
- Color-space conversion.
- Physics simulations.
- Bioinformatics.
- Chemistry simulations.

4.1.3 Fundamentals of Arm®v8 Neon technology

Arm®v8 includes both 32-bit execution and 64-bit execution states, each with their own instruction sets:

- AArch64 is the name used to describe the 64-bit execution state of the Arm®v8 architecture.

  In AArch64 state, the processor executes the A64 instruction set, which contains Neon instructions (also referred to as SIMD instructions).

- AArch32 describes the 32-bit execution state of the Arm®v8 architecture, which is almost identical to Arm®v7.

  Note: GNU and Linux documentation sometimes refers to AArch64 as ARM64.

  In AArch32 state, the processor can execute either the A32 (called ARM in earlier versions of the architecture) or the T32 (Thumb) instruction set. The A32 and T32 instruction sets are backwards compatible with Arm®v7, including Neon instructions.

4.1.3.1 Registers, vectors, lanes, and elements

The Neon unit operates on a separate register file of 128-bit registers. The Neon unit is fully integrated into the processor and shares the processor resources for integer operation, loop control, and caching. This significantly reduces the area and power cost compared to a hardware accelerator. It also uses a much simpler programming model, since the Neon unit uses the same address space as the application.

The Neon register file is a collection of registers which can be accessed as 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit registers.

The Neon registers contain **vectors** of elements of the same data type. A vector is divided into **lanes** and each lane contains a data value called an **element**.

Usually each Neon instruction results in **n** operations occurring in parallel, where **n** is the number of lanes that the input vectors are divided into. Each operation is contained within the lane. There cannot be a carry or overflow from one lane to another. The number of lanes in a Neon vector depends on the size of the vector and the data elements in the vector. A 128-bit Neon vector can contain the following element sizes:

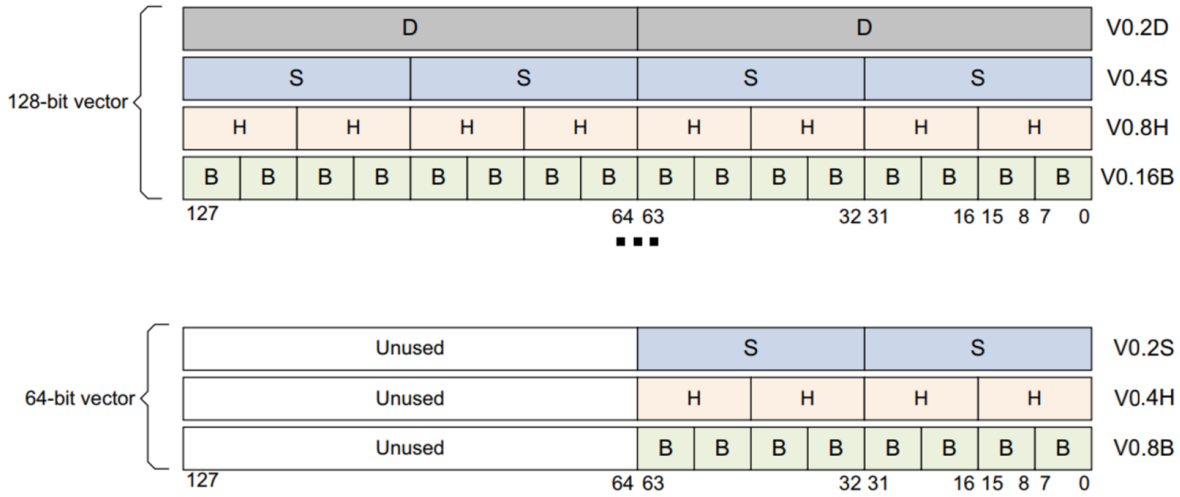
- Sixteen 8-bit elements (operand suffix `.16B`, where `B` indicates byte)
- Eight 16-bit elements (operand suffix `.8H`, where `H` indicates halfword)
- Four 32-bit elements (operand suffix `.4S`, where `S` indicates word)

- Two 64-bit elements (operand suffix `.2D`, where `D` indicates doubleword)

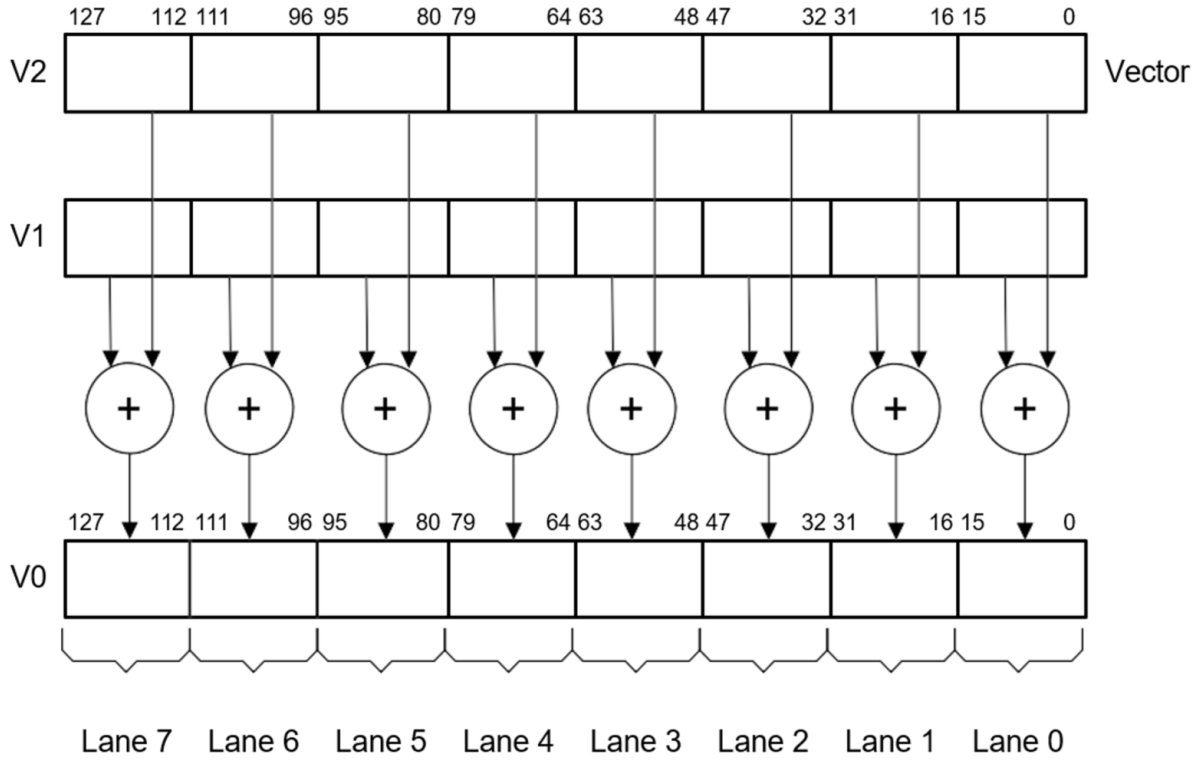
A 64-bit Neon vector can contain the following element sizes:

- Eight 8-bit elements (operand suffix `.8B`, where `B` indicates byte)
- Four 16-bit elements (operand suffix `.4H`, where `H` indicates halfword)
- Two 32-bit elements (operand suffix `.2S`, where `S` indicates word)

Note: If you want the equivalent of 1D, use `dn`.



Elements in a vector are ordered from the least significant bit. That is, element `0` uses the least significant bits of the register. Looking at an example of a Neon instruction, the instruction `ADD V0.8H, V1.8H, V2.8H` performs a parallel addition of eight lanes of 16-bit (8 x 16 = 128) integer elements from vectors in `V1` and `V2`, storing the result in `V0`:



4.1.4 Next steps

The *Optimizing C Code with Neon Intrinsics* topic provides a useful introduction to Neon programming. The tutorial describes how to use Neon intrinsics by examining an example which processes a matrix multiplication.

4.1.5 Related information

Some useful resources are:

- For definitive information about the SIMD instructions and registers, refer to the Arm Architecture Reference Manual for the Armv8-A architecture profile.
- The ISA exploration tools provide descriptions in XML and HTML format for the A64 Instruction Set Architecture, including the SIMD instructions.
- The Neon Intrinsics Reference provides information about Neon intrinsics (function calls that let C and C++ programmers code directly for Neon without writing assembly).
- Arm Community has a useful article that introduces Neon: Arm Neon Programming Quick Reference.

4.2 Optimizing C Code with Neon Intrinsics

This guide shows you how to use Neon intrinsics in your C, or C++, code to take advantage of the Advanced SIMD technology in the Arm®v8 architecture. The simple example demonstrates how to use the intrinsics and provides an opportunity to explain their purpose.

At the end of the topic, there is a **Quick reference** section to summarize the following key concepts:

- What is Neon and how can it be used?
- What are the basics of using Neon intrinsics in the C language.

4.2.1 What is Neon?

Neon is the implementation of the Arm Advanced SIMD architecture.

The purpose of Neon is to accelerate data manipulation by providing:

- 32 128-bit vector registers, each capable of containing multiple lanes of data.
- SIMD instructions to operate simultaneously on those multiple lanes of data.

Applications that can benefit from Neon technology include multimedia and signal processing, 3D graphics, scientific simulations, image processing, or other applications where fixed and floating-point performance is critical.

As an application developer, there are a number of ways you can make use of Neon technology:

- Neon-enabled open source libraries such as the Arm Compute Library or Ne10 provide one of the easiest ways to take advantage of Neon.
- Auto-vectorization features in your compiler can automatically optimize your code to take advantage of Neon.
- Neon intrinsics are function calls that the compiler replaces with appropriate Neon instructions. The intrinsics give you direct, low-level access to the exact Neon instructions you want, from C, or C++ code.
- For very high performance, hand-coded Neon assembler can be the best approach for experienced developers.

In this guide the focus is on using the Neon intrinsics for AArch64.

4.2.2 Why intrinsics?

Intrinsics are functions whose precise implementation is known to a compiler. The Neon intrinsics are a set of C and C++ functions defined in `arm_neon.h` which are supported by the Arm compilers and GCC. These functions let you use Neon without having to write assembly code because the functions themselves contain short assembly kernels, which are inlined into the calling code. In addition, register allocation and pipeline optimization are handled by the compiler so many difficulties faced by the assembly developer are avoided.

Fr a list of all the Neon intrinsics, see the Neon Intrinsics Reference. The Neon intrinsics engineering specification is contained in the Arm C Language Extensions (ACLE).

Using Neon intrinsics has a number of benefits:

- Powerful: Intrinsics give the developer direct access to the Neon instruction set, without the need for hand-written assembly code.
- Portable: Hand-written Neon assembly instructions might need to be re-written for different target processors. C and C++ code containing Neon intrinsics can be compiled for a new target or a new execution state with minimal or no code changes.
- Flexible: The developer can exploit Neon when needed, or use C/C++ when it is not, while avoiding many low-level engineering concerns.

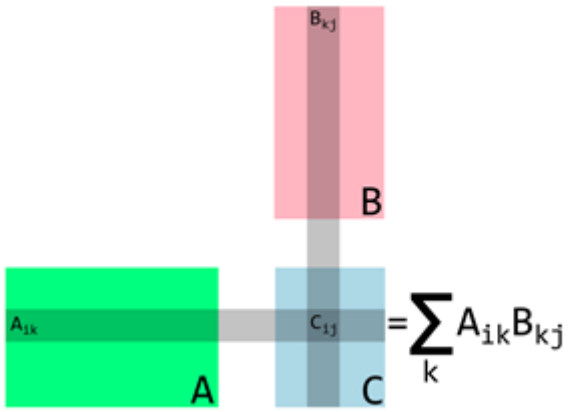
However, intrinsics might not be the right choice in all situations:

- There is a steeper learning curve to use Neon intrinsics than importing a library or relying on a compiler.
- Hand-optimized assembly code might offer the greatest scope for performance improvement even if it is more difficult to write.

4.2.3 Example: Matrix multiplication

This example re-implements some C functions using Neon intrinsics. The example chosen does not reflect the full complexity of their application, but illustrates the use of intrinsics and is a starting point for more complex code.

Matrix multiplication is an operation performed in many data intensive applications. It is made up of groups of arithmetic operations which are repeated in a straightforward way:



The matrix multiplication process is as follows:

1. Take a row in the first matrix - 'A'
2. Perform a dot product of this row with a column from the second matrix - 'B'
3. Store the result in the corresponding row and column of a new matrix - 'C'

For matrices of 32-bit floats, the multiplication could be written as:

```
void matrix_multiply_c(float32_t *A, float32_t *B, float32_t *C, uint32_t n,␣
↪uint32_t m, uint32_t k) {
    for (int i_idx=0; i_idx < n; i_idx++) {
        for (int j_idx=0; j_idx < m; j_idx++) {
            C[n*j_idx + i_idx] = 0;
            for (int k_idx=0; k_idx < k; k_idx++) {
                C[n*j_idx + i_idx] += A[n*k_idx + i_idx]*B[k*j_idx + k_idx];
            }
        }
    }
}
```

Assume a column-major layout of the matrices in memory. That is, an `n` x `m` matrix `M`, is represented as an array `M_array`, where `Mij = M_array[n*j + i]`.

This code is sub-optimal, because it does not make full use of Neon. Intrinsics can be used to improve it.

In this example, we will examine small, fixed-size matrices before moving on to larger matrices.

The following code uses intrinsics to multiply two 4x4 matrices. Since there is a small, fixed number of values to process, all of which can fit into the Neon registers of the processor at once, the loops can be completely unrolled.

```
void matrix_multiply_4x4_neon(float32_t *A, float32_t *B, float32_t *C) {
    // these are the columns A
    float32x4_t A0;
    float32x4_t A1;
    float32x4_t A2;
    float32x4_t A3;

    // these are the columns B
    float32x4_t B0;
    float32x4_t B1;
    float32x4_t B2;
    float32x4_t B3;

    // these are the columns C
    float32x4_t C0;
    float32x4_t C1;
```

(continues on next page)

(continued from previous page)

```
    float32x4_t C2;
    float32x4_t C3;

    A0 = vld1q_f32(A);
    A1 = vld1q_f32(A+4);
    A2 = vld1q_f32(A+8);
    A3 = vld1q_f32(A+12);

    // Zero accumulators for C values
    C0 = vmovq_n_f32(0);
    C1 = vmovq_n_f32(0);
    C2 = vmovq_n_f32(0);
    C3 = vmovq_n_f32(0);

    // Multiply accumulate in 4x1 blocks, i.e. each column in C
    B0 = vld1q_f32(B);
    C0 = vfmaq_laneq_f32(C0, A0, B0, 0);
    C0 = vfmaq_laneq_f32(C0, A1, B0, 1);
    C0 = vfmaq_laneq_f32(C0, A2, B0, 2);
    C0 = vfmaq_laneq_f32(C0, A3, B0, 3);
    vst1q_f32(C, C0);

    B1 = vld1q_f32(B+4);
    C1 = vfmaq_laneq_f32(C1, A0, B1, 0);
    C1 = vfmaq_laneq_f32(C1, A1, B1, 1);
    C1 = vfmaq_laneq_f32(C1, A2, B1, 2);
    C1 = vfmaq_laneq_f32(C1, A3, B1, 3);
    vst1q_f32(C+4, C1);

    B2 = vld1q_f32(B+8);
    C2 = vfmaq_laneq_f32(C2, A0, B2, 0);
    C2 = vfmaq_laneq_f32(C2, A1, B2, 1);
    C2 = vfmaq_laneq_f32(C2, A2, B2, 2);
    C2 = vfmaq_laneq_f32(C2, A3, B2, 3);
    vst1q_f32(C+8, C2);

    B3 = vld1q_f32(B+12);
    C3 = vfmaq_laneq_f32(C3, A0, B3, 0);
    C3 = vfmaq_laneq_f32(C3, A1, B3, 1);
    C3 = vfmaq_laneq_f32(C3, A2, B3, 2);
    C3 = vfmaq_laneq_f32(C3, A3, B3, 3);
    vst1q_f32(C+12, C3);
}
```

Fixed-size 4x4 matrices are chosen because:

- Some applications need 4x4 matrices specifically, for example: graphics or relativistic physics.
- The Neon vector registers hold four 32-bit values. Matching the program to the architecture makes it easier to optimize.
- This 4x4 kernel can be used in a more general kernel.

Summarizing the intrinsics that have been used here:

Code element	What is it?	Whay are we using it?
`float32x4_t`	An array of four 32-bit floats.	One `uint32x4_t` fits into a 128-bit register. We can ensure there are no wasted register bits even in C code.
`vld1q_f32(...)`	A function which loads four 32-bit floats into a `float32x4_t`.	To get the matrix values we need from A and B.
`vfmaq_lane_f32(...)`	A function which uses the fused multiply accumulate instruction. Multiplies a `float32x4_t` value by a single element of another `float32x4_t` then adds the result to a third `float32x4_t` before returning the result.	Since the matrix row-on-column dot products are a set of multiplications and additions, this operation fits quite naturally.
`vst1q_f32(...)`	A function which stores a `float32x4_t` at a given address.	To store the results after they are calculated.

To multiply larger matrices, treat them as blocks of 4x4 matrices. However, this approach only works with matrix sizes which are a multiple of four in both dimensions. To use use this method without changing it, pad the matrix with zeroes.

The code for a more general matrix multiplication is listed below. The structure of the kernel has changed very little, with the addition of loops and address calculations being the major changes. Like in the 4x4 kernel, unique variable names are used for the B columns. The alternative would be to use one variable and re-load it. This acts as a hint to the compiler to assign different registers to these variables. Assigning different registers enables the processor to complete the arithmetic instructions for one column, while waiting on the loads for another.

```
void matrix_multiply_neon(float32_t  *A, float32_t  *B, float32_t *C, uint32_t n,␣
↪uint32_t m, uint32_t k) {
    /*
     * Multiply matrices A and B, store the reult in C.
     * It is the users responsibility to make sure the matrices are compatible.
     */

    int A_idx;
    int B_idx;
    int C_idx;

    // these are the columns of a 4x4 sub matrix of A
    float32x4_t A0;
    float32x4_t A1;
    float32x4_t A2;
    float32x4_t A3;

    // these are the columns of a 4x4 sub matrix of B
    float32x4_t B0;
    float32x4_t B1;
    float32x4_t B2;
    float32x4_t B3;

    // these are the columns of a 4x4 sub matrix of C
    float32x4_t C0;
    float32x4_t C1;
    float32x4_t C2;
    float32x4_t C3;

    for (int i_idx=0; i_idx<n; i_idx+=4 {
            for (int j_idx=0; j_idx<m; j_idx+=4){
```

(continues on next page)

(continued from previous page)

```
                    // zero accumulators before matrix op
                    c0=vmovq_n_f32(0);
                    c1=vmovq_n_f32(0);
                    c2=vmovq_n_f32(0);
                    c3=vmovq_n_f32(0);
                    for (int k_idx=0; k_idx<k; k_idx+=4){
                         // compute base index to 4x4 block
                         a_idx = i_idx + n*k_idx;
                         b_idx = k*j_idx k_idx;

                         // load most current a values in row
                         A0=vld1q_f32(A+A_idx);
                         A1=vld1q_f32(A+A_idx+n);
                         A2=vld1q_f32(A+A_idx+2*n);
                         A3=vld1q_f32(A+A_idx+3*n);

                         // multiply accumulate 4x1 blocks, i.e. each column C
                         B0=vld1q_f32(B+B_idx);
                         C0=vfmaq_laneq_f32(C0,A0,B0,0);
                         C0=vfmaq_laneq_f32(C0,A1,B0,1);
                         C0=vfmaq_laneq_f32(C0,A2,B0,2);
                         C0=vfmaq_laneq_f32(C0,A3,B0,3);

                         B1=vld1q_f32(B+B_idx+k);
                         C1=vfmaq_laneq_f32(C1,A0,B1,0);
                         C1=vfmaq_laneq_f32(C1,A1,B1,1);
                         C1=vfmaq_laneq_f32(C1,A2,B1,2);
                         C1=vfmaq_laneq_f32(C1,A3,B1,3);

                         B2=vld1q_f32(B+B_idx+2*k);
                         C2=vfmaq_laneq_f32(C2,A0,B2,0);
                         C2=vfmaq_laneq_f32(C2,A1,B2,1);
                         C2=vfmaq_laneq_f32(C2,A2,B2,2);
                         C2=vfmaq_laneq_f32(C2,A3,B3,3);

                         B3=vld1q_f32(B+B_idx+3*k);
                         C3=vfmaq_laneq_f32(C3,A0,B3,0);
                         C3=vfmaq_laneq_f32(C3,A1,B3,1);
                         C3=vfmaq_laneq_f32(C3,A2,B3,2);
                         C3=vfmaq_laneq_f32(C3,A3,B3,3);
                    }
   //Compute base index for stores
   C_idx = n*j_idx + i_idx;
   vstlq_f32(C+C_idx, C0);
   vstlq_f32(C+C_idx+n,C1);
   vstlq_f32(C+C_idx+2*n,C2);
   vstlq_f32(C+C_idx+3*n,C3);
  }
 }
}
```

Compiling and disassembling this function, and comparing it with the C function shows:

- Fewer arithmetic instructions for a given matrix multiplication, because we are leveraging the Advanced SIMD technology with full register packing. Typical C code, generally, does not do this.
- `FMLA` instead of `FMUL` instructions. As specified by the intrinsics.
- Fewer loop iterations. When used properly intrinsics allow loops to be unrolled easily.

However, there are unnecessary loads and stores due to memory allocation and initialization of data types (for example, `float32x4_t`) which are not used in the pure C code.

4.2.4 Program conventions

4.2.4.1 Macros

In order to use the intrinsics, the Advanced SIMD architecture must be supported. Some specific instructions might not be enabled. When the following macros are defined and equal to 1, the corresponding features are available:

- `__aarch64__`
  - Selection of architecture dependent source at compile time.
  - Always 1 for AArch64.
- `_ARM_NEON`
  - Advanced SIMD is supported by the compiler.
  - Always 1 for AArch64.
- `_ARM_NEON_FP`
  - Neon floating-point operations are supported.
  - Always 1 for AArch64
- `_ARM_FEATURE_CRYPTO`
  - Crypto instructions are available.
  - Cryptographic Neon intrinsics are therefore available.
- `_ARM_FEATURE_FMA`
  - The fused multiply-accumulate instructions are available.
  - Neon intrinsics which use these are therefore available.

This list is not exhaustive and further macros are detailed in the Arm C Language Extensions document.

4.2.4.2 Types

There are three major categories of data type available in `arm_neon.h` which follow these patterns:

`baseW_t` Scalar data types

`baseWxL_t` Vector data types

`baseWxLxN_t` Vector array data types

Where:

- `base` refers to the fundamental data type.
- `W` is the width of the fundamental type.
- `L` is the number of scalar data type instances in a vector data type, for example an array of scalars.
- `N` is the number of vector data type instances in a vector array type, for example a struct of arrays of scalars.

Generally, `W` and `L` are such that the vector data types are 64 or 128 bits long, and so fit completely into a Neon register. `N` corresponds with those instructions which operate on multiple registers at once.

4.2.4.3 Functions

As per the Arm C Language Extensions, the function prototypes from `arm_neon.h` follow a common pattern. At the most general level this is:

```
ret v[p][q][r]name[u][n][q][x][_high][_lane | laneq][_n][_result]_type(args)
```

Some of the letters and names are overloaded, but in the order above:

ret The return type of the function.

v Short for `vector` and is present on all the intrinsics.

p Indicates a pairwise operation. (`[value]` means `value` may be present).

q Indicates a saturating operation (with the exception of `vqtb[l][x]` in AArch64 operations where the `q` indicates 128-bit index and result operands).

r Indicates a rounding operation.

name The descriptive name of the basic operation. Often, this is an Advanced SIMD instruction, but it does not have to be.

u Indicates signed-to-unsigned saturation.

n Indicates a narrowing operation.

q Postfixing the name indicates an operation on 128-bit vectors.

x Indicates an Advanced SIMD scalar operation in AArch64. It can be one of `b`, `h`, `s`, or `d` (that is, 8, 16, 32, or 64 bits).

_high In AArch64, used for widening and narrowing operations involving 128-bit operands. For widening 128-bit operands, `high` refers to the top 64-bits of the source operand (or operands). For narrowing, it refers to the top 64-bits of the destination operand.

_n Indicates a scalar operand supplied as an argument.

_lane Indicates a scalar operand taken from the lane of a vector. `_laneq` indicates a scalar operand taken from the lane of an input vector of 128-bit width. (`left | right` means only `left` or `right` would appear).

type The primary operand type in short form.

args The arguments of the function.

4.2.5 Quick reference

What is Neon?

Neon is the implementation of the Advanced SIMD extension to the Arm architecture.

All processors compliant with the Arm®v8-A architecture (for example, the Cortex-A76 or Cortex-A57) include Neon. In the developer's view, Neon provides an additional 32 128-bit registers with instructions that operate on 8, 16, 32, or 64 bit lanes within these registers.

Which header file must you include in a C file in order to use the Neon intrinsics?

`arm_neon.h`

`#include <arm_neon.h>` must appear before the use of any Neon intrinsics.

What do the data types float64_t, poly64x2_t, and int8x8x3_t represent?

- `float64_t` is a scalar type which is a 64-bit floating-point type.
- `poly64x2_t` is a vector type of two 64-bit polynomial scalars.
- `int8x8x3_t` is a vector array type of three vectors of eight 8-bit signed integers.

What does the int8x16_t vmulq_s8 (int8x16_t a, int8x16_t b) function do?

The `mul` in the function name indicates that this intrinsic uses the `MUL` instruction. The types of the arguments and return value (sixteen bytes of signed integers) inform you that this intrinsic maps to the following instruction:

```
MUL Vd.16B, Vn.16B, Vm.16B
```

This function multiplies corresponding elements of `a` and `b`, and returns the result.

The deinterleave function defined in this tutorial can only operate on blocks of sixteen 8 bit unsigned integers. If you had an array of uint8_t values that was not a multiple of sixteen in length, how might you

account for this while: 1) Changing the arrays, but not the function? and 2) Changing the function, but not the arrays?

1. Padding the arrays with zeros would be the simplest option, but padding might have to be accounted for in other functions.
2. One method would be to use the Neon de-interleave for every whole multiple of sixteen values, and then use the C de-interleave for the remainder.

4.2.6 Related information

- Engineering specifications for the Neon intrinsics can be found in the Arm C Language Extensions (ACLE).
- The Neon Intrinsics Reference provides a searchable reference of the functions specified by the ACLE.
- The Architecture Exploration Tools let you investigate the Advanced SIMD instruction set.
- The Arm Architecture Reference Manual provides a complete specification of the Advanced SIMD instruction set.

4.3 Useful Neon Resources

Some useful resources when coding for Neon, are:

- Arm Neon web page
- Neon Intrinsics
- Arm Neon Optimization blog
- Arm Neon programming quick reference
- Coding for Neon: Matrix Multiplication
- Coding for Neon: Rearranging Vectors