arm

Porting and Optimizing HPC Applications for Arm

Version 3.0

Documentation

Porting and Optimizing HPC Applications for Arm
Documentation

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	29 March 2019	Non-Confidential	First release.
0110-00	5 June 2019	Non-Confidential	Document update to version 1.1.
0200-00	11 November 2019	Non-Confidential	Document update to version 2.0.
0210-00	30 March 2021	Non-Confidential	Document update to version 2.1.
0300-00	24 August 2021	Non-Confidential	Document update to version 3.0.
0300-01	4 March 2022	Non-Confidential	Document update 1 for version 3.0.
0300-02	25 May 2022	Non-Confidential	Document update 2 for version 3.0.

Proprietary Notice

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.

Product Status

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

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

List of Figures

List of Tables

1 Introduction

Describes how to port your High Performance Computing (HPC) applications to Arm-based hardware, how to start optimizing the applications after they are ported, and what tools Arm provides that can help.

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm® Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Signal names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace bold`	Language keywords when used outside example code.
`monospace` <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.
⚠ Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
⚠ Warning	Requirements for the system. Not following these requirements might result in system failure or damage.

Convention	Use
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.
Note	An important piece of information that needs your attention.
Tip	A useful tip that might make it easier, better or faster to perform a task.
Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2 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 port onto the Arm architecture with little or no modification, because:

- All major Linux distributions support Arm. Major Linux distribution support provides an extensive library of common Linux packages that are 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.

- Arm® Compiler for Linux is available.

> **Note**
> Arm Compiler for Linux also accepts GCC compiler options, where possible. For more information, see the C/C++ Supporting Reference and Fortran Supporting reference chapters of the Arm C/C++ Compiler and Arm Fortran Compiler Developer and Reference guides.

However, there are a few features of the Arm architecture that might impact your application. These features are described in the **Troubleshooting** section in the **Port your application** topic.

2.1 Tools to Port and Optimize

Arm provides tools to help you port and optimize applications for Arm.

Arm Compiler for Linux

Arm® Compiler for Linux is comprised of Arm C/C++/Fortran Compiler and Arm Performance Libraries.

Arm C/C++/Fortran Compiler

Arm C/C++/Fortran Compiler is a Linux user space, C/C++, and Fortran compiler, tuned for scientific computing, HPC, and enterprise workloads. The compiler offers:

- Processor-specific optimizations for various server-class Arm-based platforms.

- Optimal shared-memory parallelism using latest Arm-optimized OpenMP run-time.

- Optimized scalar and vector maths functions.

- C++17 and Fortran 2003 language support (partial Fortran 2008 support) with OpenMP 4.5 (OpenMP 5.0 for C/C++).

- Support for Armv8-A, SVE, and SVE2 architecture extensions.

- Based on leading open-source compiler projects: LLVM, Clang, and Flang.

- Runs on leading Linux distributions: Red Hat, SLES, and Ubuntu.

Arm Performance Libraries

Arm Performance Libraries is a 64-bit, Armv8-A core math libraries, optimized for Server and HPC applications on Arm-based platforms, providing best-in-class serial and parallel performance. The libraries offer:

- Commonly used low-level math routines: BLAS, LAPACK, FFT, sparse linear algebra, and libamath functions.

- An 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 C/C++/Fortran Compiler and GCC.

Resources

- For the latest information and other resources, see the Arm Compiler for Linux Developer web page.
- Arm C/C++ Compiler Developer and Reference guide
- Arm Fortran Compiler Developer and Reference guide
- Arm Performance Libraries Developer and Reference guide

Arm Forge

A cross-platform toolkit to debug (Arm® DDT), profile (Arm® MAP), and analyze (Arm® Performance Reports) high-performance parallel applications. Arm Forge:

- Is available on most of the Top500 machines in the world.

- Is fully supported by Arm on x86, ppc64le, Nvidia GPUs, and AArch64.

- Has powerful and in-depth error detection mechanisms (including memory debugging).

- Identifies and understands bottlenecks using a sampling-based profiler.

- Available at any scale (from the desktop to leadership-class HPC).

- Has supports for remote interactive sessions.

- Analyzes metrics around CPU, memory, I/O, and hardware counters.

- Supports custom metrics.

- Provides simple guidance on how to improve workload efficiency.

- Can be integrated into various systems for validation (for example, continuous integration).

- Can be automated to remove the requirement for user intervention.

Resources

- For the latest release information and other resources, see the Arm Forge Developer web page.
- For Arm DDT documentation, see Arm DDT User Guide.

- For Arm MAP documentation, see Arm MAP User Guide.
- For Arm Performance Reports documentation, see Arm Performance Reports User Guide.

2.2 Port your application

This high-level topic describes what to consider when you are porting your application to run on Arm-based hardware.

To port your application, follow these steps:

Note

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

---

1. Ensure that all of your application dependencies have been ported.

Note

To ensure that the Fortran interface is compatible with your application, you must compile the application dependencies with the same toolchain that you use to compile your application.

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

- **I/O 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 Libraries provides an 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.

Often, these dependencies have been built on Arm before with the Arm and GNU toolchains:

- For a full list of all ported applications using the Arm and GNU toolchains, see the community Packages Wiki.

2. Ensure that you are using the correct compiler.

   During your build configuration, specify which C, C++, and Fortran compilers to use. For example, for Arm® C/C++/Fortran 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. Ensure that you are using the correct compiler options. Most GCC options are supported by Arm Compiler for Linux. If you are compiling and running your application on the same hardware, Arm recommends that you compile with the `-mcpu=native` option (in addition to any other options you regularly use to compile your application). `-mcpu=native` ensures that your compiler uses the micro-architectural optimizations suitable for your system. To enable the optimal version of Arm Performance Libraries for your target, also include `-armpl` (for Arm Compiler for Linux) or `-larmpl` (for GCC) in your compile and link commands.

   Note If you are compiling and running on different hardware, read Accessing the library to find out which compiler options you can use for Arm Performance Libraries.

   ---

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.

Porting - troubleshooting

Here are some problems that you might find when porting your application:

- *Configure is unable to identify your platform*

  Configure might not be able to identify your platform because the `config.guess` supplied with the application is out of date. This can also occur 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
  ```

- *Libtool fails to link Fortran applications or interfaces*

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

  Solution:

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

  ```
  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.

- *#ifdefs in the makefile are not being set*

  There might be compiler-dependent `#ifdefs` in the source which are not 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`.

- *Unsupported language features*

Your code might be using language features which are not supported by Arm Compiler for Linux.

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) support.

◦ C/C++ OpenMP 4.0, C/C++ OpenMP 4.5 (Partial), and C/C++ OpenMP 5.0 (Partial) support.

- *You experience a race condition you have not encountered before*

AArch64 uses a weak memory model. A weak memory model is where read and writes can be reordered. Sometimes, 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 architecture profile.

- *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.

Occasionally, 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:

```
c = a / b
```

Test for `b==0` before executing the divide, and generate a warning, or adjust the program flow accordingly.

- *Thread mapping and pinning on Arm*

Arm chips can have lots of cores. It is 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

- *Segmentation fault when calling an Arm Performance Libraries 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.

- *Building Position Independent Code (PIC) on AArch64*

Failure can occur at the linking stage when building Position-Independent Code (PIC) on AArch64 using the lowercase `-fpic` compiler flag with GCC compilers (gfortran, gcc, g++), instead of using the uppercase `-fPIC` flag.

Note
- ◦ This issue does not occur when using the `-fpic` flag with Arm Compiler for Linux (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 Linux 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 are large enough to allow the entries to be resolved by the dynamic loader.

To increase code portability, Arm recommends using `-fPIC` (instead of `-fpic`) when compiling with Arm C/C++/Fortran Compiler.

For more information, see Building Position Independent Code (PIC) on AArch64.

- *Applications supporting GCC builds on Arm - but use Armv-7 compiler flags*

   Some Armv7 flags that are needed for Armv7, cause errors for Armv8 targets. For example, on Armv8 Neon® is compulsory, so the flag `-fp=neon` does not exist on Armv8. If it is used when compiling for Armv8, 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 Armv8 compiler flags.

   Related information

   Get started on Arm
   Develop on Arm
   Packages Wiki
   Arm Compiler for Linux
   Arm Forge

2.3 Optimize

After porting your application to Arm, the next step is to optimize it. The following steps describe how you can use the Arm compilers, debugger, and profiler to further enhance the performance of your application on Arm.

1. Optimize you code using compiler optimization options:

   a. Enable auto-vectorization with the `-O<x>` options:

Table 2-1: Arm Compiler for Linux optimization options

Option	Description	Auto-vectorization
-O0	Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this might result in a significantly larger image. This is the default optimization level.	Never
-O1	Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug.	Disabled by default.
-O2	High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information.	Enabled by default.
-O3	Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.	Enabled by default.
-Ofast	Enable all the optimizations from level 3, including those performed with the -ffp-mode=fast armclang option. This level also performs other aggressive optimizations that might violate strict compliance with language standards.	Enabled by default.

Note

- 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 has 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 more detailed description of auto-vectorizing your code for Arm Neon®, see Compile for Neon with auto-vectorization.

b. For C/C++ code, enable the vectorized implementation of `libm` math functions using the `-fsimdmath` option. Combine this with the `-O<x>` option from the previous step.

> **Note**
>
> For Fortran source, vector implementations are used, when possible, by default, but can be disabled using the `-fno-simdmath` compiler flag.

c. Optimize for your hardware. To compile your code for your specific core, use the `-mcpu` option. Compiling for the specific core enables the compile to optimize knowing the architecture and microarchitectural versions implemented on that core.

In summary, if you compile and run your application on the same hardware, a typical set of compiler optimization options are:

```
{armclang|armclang++|armflang} -fsimdmath -mcpu=native -c -O3
 <source>{.c|.cpp|.f}
```

For a full list of compiler optimization options, see the options descriptions in the Arm C/C++ Compiler Reference Guide or the options descriptions in the Arm Fortran Compiler Reference Guide.

2. Enable optimized Arm Performance Libraries functions.

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. To find out what the latest version of LAPACK that is supported in Arm Performance Libraries is, see Arm Performance Libraries.
- FFT functions: a set of Fast Fourier Transform routines for real and complex data using the FFTW interface.
- Sparse linear algebra.
- libamath: a subset of libm, which is a set of optimized mathematical functions.
- libastring: A subset of libc, which is a set of optimized string functions.

Arm Performance Libraries also provides improved Fortran math intrinsics with auto-vectorization.

To enable Arm Performance Libraries, add the `-armpl` option to your compile command line. `-armpl` provides a simple interface to select thread-parallelism and architectural tuning. Combining `-armpl` with the `-mcpu` option enables the compiler to find appropriate Arm Performance Libraries header files (during compilation) and libraries (during linking). Arm recommends using both options to achieve the best performance enhancement:

```
{armclang|armclang++|armflang} <options> code_with_math_routines{.c|.cpp|.f*} -
mcpu=<target> -armpl=<arg1>,<arg2>...
```

Note

- If your build process compiles and links as two separate steps, ensure that you add the same `-armpl` and `-mcpu` options to both. For more information about using the `-armpl` option, see Get Started with Arm Performance Libraries on the Arm Developer website.

- For GCC, you need to load the correct environment module for the system and explicitly link to your chosen flavor (lp64/ilp64, mp) with full library path:

```
{gcc|gfortran} <options> code_with_math_routines{.c|.f*} -larmpl
```

For a more detailed description about using `-armpl`, see the Library selection topic.

3. Use Arm Compiler for Linux Optimization Remarks.

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

To enable Optimization Remarks, pass one or more of the following `-Rpass` flags at compile time:

Table 2-2: -Rpass flags to enable optimization remarks

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

For example, to get actionable information on which loops can and cannot be vectorized at compile time, pass:

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

Note

- Optimization Remarks are only available when you have set an appropriate debug flag, for example `-g`.

- Optimization Remarks are piped to `stdout` at compile time.

For more information, refer to the C/C++ Arm Optimization Reports documentation or the Fortran Arm Optimization Reports documentation.

4. Use Arm Optimization Report.

    Arm Optimization Report is a feature of Arm Compiler for Linux that builds upon the llvm-opt-report tool available in open-source LLVM. The new Arm Optimization Report feature makes it easier to see what optimization decisions the compiler is making about unrolling, vectorizing, and interleaving, all in-line with your source code.

    To enable Arm Optimization Report:

    a. At compile time, add the `-fsave-optimization-record` to the command line.

        A `<filename>.opt.yaml` report is generated by the compiler, where `<filename>` is the name of the binary.

    b. Use Arm Optimization Report (`arm-opt-report`) to inspect the `<filename>.opt.yaml` report as augmented source code:

    ```
    arm-opt-report <filename>.opt.yaml
    ```

    The annotated source code appears in the terminal.

    For more information, refer to the C/C++ Arm Optimization Reports documentation or the Fortran Arm Optimization Reports documentation.

5. Use Arm C/C++/Fortran Compiler Optimization Remarks.

6. Use the Arm Compiler for Linux 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 th loop.

    *   `!$OMP SIMD` - An OpenMP directive to indicate that a loop can be transformed into a SIMD loop.

    *   `!$MEM PREFETCH` - Tells the compiler to generate prefetch instructions to fetch elements and load them in the data cache, ahead of their first use.

    *   `!DIR$ VECTOR ALWAYS` - Forces the compiler to vectorize a loop regardless 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.

7. Optimize by iteration.

   Use Arm Forge to analyze application performance, as well as debug and profile your ported application:

   • Use Arm Performance Reports to characterize and understand the performance of your application runs.

   • Use Arm DDT to debug your code to ensure that the application is correct. Arm DDT 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. To understand the code performance, Arm MAP collects:

     ◦ A broad set of performance metrics.

     ◦ A broad set of time classification metrics.

     ◦ Instruction information (hardware counters).

     ◦ Specific metrics (for example MPI call and message rates, I/O data rates, and energy data).

     ◦ Custom metrics (metrics defined by you).

   To optimize application performance, use the Arm Performance Reports, Arm DDT, Arm MAP tools and follow an iterative identification and resolving cycle:

   a. Run your code on real workloads and generate a performance report with Arm Performance Reports.

   b. Use Arm MAP to examine the I/O and trace and debug suspicious or slow access patterns.

      Common problems include:

      • Checkpointing too often.

      • Many small read and writes.

      • Using your home directory instead of the scratch directory.

      • Multiple nodes using the filesystem a the same time.

   c. Use Arm Performance Reports to identify the workload balance of your application, then use Arm DDT and Arm MAP to dive into partitioning code.

      Common problems include:

      • Your dataset is too small to efficiently run at scale.

      • I/O contention causes late sender.

      • Partitioning code bugs.

   d. Use Arm MAP to identify lines of code that are causing memory access pattern problems.

      Common problems include:

- Initializing memory on one core but using it on a different core.

- Arrays of structures causing inefficient cache utilization.

- Caching results when re-computation is more efficient.

e. Use Arm Performance Reports to track communication performance, and Arm MAP to see which communication calls are slow and why.

Common problems include:

- Short, high-frequency messages are very sensitive to latency.

- Too many synchronizations.

- No overlap between communication and computation.

f. Use Arm Performance Reports to observe the core utilization and synchronization overhead, then use Arm MAP to identify the corresponding code.

Common problems include:

- Implicit thread barriers inside tight loops.

- significant core idle time because of worload imbalance.

- Threads migrating between cores at runtime.

g. Use Arm Performance Reports to observe the numerical intensity and level of vectorization, and Arm MAP to identify the hot loops and unvectorized code.

Common problems include:

- Sub-optimal compiler options for your system.

- Numerically-intensive loops with hard to vectorize patterns.

- Not utilizing highly-optimized math libraries.

Example slowdown factors that can occur are:

Figure 2-1: Iterative optimization cycle.
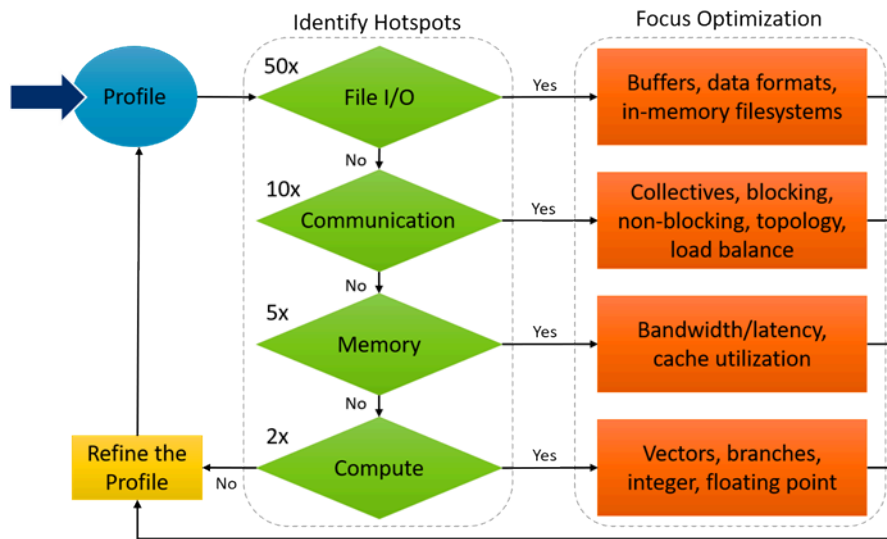
> **Note**
>
> The 50x, 10x, 5x, and 2x numbers in the figure are potential slowdown factors that Arm has observed in real-world applications (when that aspect of performance is done incorrectly).

For more information about using analysis, debugging, and profiling tools, see the Arm Forge user guide.

Related information

Packages Wiki
Latest additions to the Armv8-A architecture
Arm Compiler for Linux
Arm Forge

2.4 Tool quick references

This section collects some quick reference information for Arm® Compiler for Linux and Arm Forge.

2.4.1 Arm C/C++/Fortran Compiler Quick Reference

Quick reference for using Arm® C/C++/Fortran Compiler.

Common compiler options

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

General

Table 2-3: Common compiler options for armclang, armclang++, and armflang

Option	Description
`-o <file>`	Write output to <file>.
`-c`	Only run pre-process, compile, and assemble steps.
`-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, or 3).
`-Ofast`	Enables aggressive optimization of floating-point operations.
`-ffp-contract=(fast|on|off)`	Allow fused floating-point operations (for example a Fused Multiply-Add (FMA)).
`-fsave-optimization-record`	Enable the generation of a YAML optimization record file to use with Arm Optimization Report. Arm Optimization Report shows you the optimization decisions that the compiler is making, in-line with your source code, enabling you to better understand the unrolling, vectorization, and interleaving behavior.

Fortran

Table 2-4: armflang-specific Fortran options

Option	Description
`-cpp`	Preprocess Fortran files. Default for `.F`, `.F90`, and `.F95`.
`-module <path>`	Specifies a directory to place, and search for, module files.
`-Mallocatable=(95|03)`	95: Use Fortran 95 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 64-bit (that is, real (`KIND=8`)). Unspecified real kinds are evaluated as `KIND=8`.
`-i8`	Set the default kind for `INTEGER` and `LOGICAL` to 64-bit (that is, `KIND=8`).

Pragmas

Arm C/C++ Compiler supports pragmas to both encourage and suppress auto-vectorization. These pragmas use, 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 interleave by a given factor.

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

Further resources

- Arm C/C++ Compiler Developer and Reference guide
- Get started with Arm C/C++ Compiler
- Arm Fortran Compiler Developer and Reference guide
- Get started with Arm Fortran Compiler

2.4.2 Arm Performance Libraries Quick Reference

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

Basic usage

To configure Arm Performance Libraries for your application, there are two decisions to make:

- **Decision 1: 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 You can also enable the parallel Arm Performance Libraries using:

  ```
  -armpl -fopenmp
  ```

- **Decision 2: 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**
> - Options can be combined, for example:
>
>   ```
>   -armpl=ilp64, parallel
>   ```
>
> - If you are compiling Fortran code, you can also enable 64-bit integers using:
>
>   ```
>   -armpl -i8
>   ```

Porting to Arm Performance Libraries

If your software uses standard BLAS, LAPACK, and FFTW interfaces, your code should port without problems.

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

Further resources
Arm Performance Libraries Reference Guide

2.4.3 Arm DDT Quick Reference

Quick reference for using Arm® DDT.

Workstation or remote interactive sessions

1. Log in to a terminal session and prepare your environment. Load the environment module for Arm Forge.
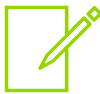
> **Note**
> The name of the environment variable is set by the system administrator. Check with your system administrator what the environment variable is called for your system.

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 reorder 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**.

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

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

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

     > **Note**
     > The name of the environment variable is set by the system administrator. Check with your system administator what the environment variable is called 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 reorder 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 if 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**.

Further resources

Arm Forge user guide

2.4.4 Arm MAP Quick Reference

Quick reference for using Arm® MAP.

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 set by the system administrator. Check with your system administator what the environment variable is called 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.

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 minimize interference between tasks.

Look at whether hot sections have similar work between compilers

- To locate sections of code that consume most of the run-time, use Arm MAP profile runs.

- Are the locations the same for both compilers? Consider if these sections could be improved.

- Report major run-time differences across compilers to the Arm C/C++/Fortran 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 C/C++/Fortran Compiler or platform-compatible versions.

- If AVX512 vector instructions are present, consider converting them.

Further resources

Arm Forge user guide

2.4.5 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 set by the system administrator. Check with your system administrator what the environment variable is called 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.

Further resources

Arm Performance Reports user guide

2.5 Porting and Tuning Recipes

Describes where to find recipes for building many common HPC applications.

For detailed, community-driven instructions on how to build many common scientific applications, benchmarks, and libraries using both open-source tools and the Arm HPC tools, see the Packages wiki.

3 Thread Mapping

This chapter describes how thread mapping impacts application performance.

3.1 OpenMP Thread Mapping

Describes what to consider when managing OpenMP thread mapping on AArch64 platforms.

The placement and management of OpenMP threads can have a significant impact on the performance of OpenMP enabled applications. By default, no environment variables are set to control the placement and binding of OpenMP threads. Not controlling the OpenMP threads leaves the kernel free to distribute the threads over the available resources, and swap the OpenMP threads between physical cores dynamically, which is unlikely to be the best-performing configuration for your applications.

With the large physical core-counts on infrastructure-scale AArch64 CPUs, the impact of the kernel swapping OpenMP threads between available cores, a process known as *thread migration*, can influence High Performance Computing (HPC) application performance more than it would on other platforms. More specifically, HPC applications are data driven, so the locality of data and threads is important. Thread migration causes problems because it moves threads away from the caches that have the data in. This is especially problematic for NUMA systems, where the tread is migrated to another NUMA node, and the data must be fetched across NUMA domains.

If your application depends on OpenMP parallelism, it is important to understand:

* How to control and manage OpenMP thread placement.

* How to understand where OpenMP threads are being executed.

* What configuration works best for each application.

With any compiler, Arm recommends using `lstopo` to understand the thread layout of the system. For more information, see the **lstopo** section.

OpenMP environment variables

By default, the following OpenMP environment variables are unset:

Table 3-1: OpenMP environment variables

Environment variable	Description
`OMP_NUM_THREADS=<value>[, <value>[, ...]]`	Specifies the default number of threads to use: • If a single value is passed, your application uses a single level of parallelism. In other words, nested parallelism is disabled. • If a comma-separated list of values are passed, the values denote the number of threads to use at each level of nesting, starting from the outermost parallel region.

Environment variable	Description
`OMP_PROC_BIND={true|false|close|spread|master}`	Specifies whether threads can be moved between processors, and controls how the threads are distributed: • `close`: Implies that threads are bound and laid out in successive order, based on the unit of `OMP_PLACES`. `close` packs OpenMP threads as close to each other as possible, which is good for communication (avoids inter-socket) and cache sharing. • `spread`: Implies that threads are bound and distributed as far apart as possible, based on the unit of `OMP_PLACES`. `spread` places OpenMP threads as far away from each other as possible, which is good for resource utilization (FP units, memory bandwidth). • `master`: The OpenMP worker threads are in the same place partition as the master thread. **Note:** `spread` or `close` placement of OpenMP threads only matters when you are under-populating a node.
`OMP_PLACES={threads|cores|sockets}|{<lower-bound>:<length>:<stride>}`	Specifies the thread placement (threads, cores, or sockets), and can also be used for explicit thread placement. For example, where `OMP_PLACES={0:2:2},{4:2:2},{8:2:2}` means: • T0: 0,2 • T1: 4,6 • T2: 8,10
`OMP_DISPLAY_ENV={true|false|verbose}`	Controls what OpenMP environment information displays on application on startup. `OMP_DISPLAY_ENV` can be set to: • `true`: On startup, your application displays the version of OpenMP along with the value for all of the OpenMP Internal Control Variables (ICVs) which are affected by environment variables in this topic, and other factors. **Note:** There might be a discrepancy between the value of your environment variables and the ICVs reported at runtime because ICVs can be controlled in other ways. • `false`: No output is produced. • `verbose`: The values of any implementation-specific variables are displayed in addition to the standard OpenMP ICVs.

If you are using Arm® Compiler for Linux, Arm also recommends that you set `KMP_AFFINITY=verbose` to help you to understand the thread placement on your system and the impact on runtime, so that you can choose the best configuration for your application.

> **Note**
> `KMP_AFFINITY` only works for LLVM-based compilers, it does not work for GNU compilers. If you are using a GNU compiler, you can use `OMP_DISPLAY_ENV=VERBOSE` to list all the OpenMP environment variables, however it does not give the core mappings.

OpenMP thread placement basics

To avoid multithreading performance problems when using Arm Compiler for Linux, it is important that you have the appropriate environment set up.

Set the number of OpenMP threads

To set the number of threads to use in your program, set the environment variable `OMP_NUM_THREADS`. `OMP_NUM_THREADS` sets the number of threads used in OpenMP parallel regions defined in your own code, and within Arm Performance Libraries. If you set `OMP_NUM_THREADS` to a single value, your program uses a single level of parallelism. In this case, nested parallelism is disabled.

> **Note**
> The information about setting `OMP_NUM_THREADS` applies to both compilers supported by Arm Performance Libraries in the 22.0 release: Arm C/C++/Fortran Compiler 22.0 and GCC 11.2.

For example, consider the following code, `threading.c`, which defines a nested parallel region:

```c
#include <stdio.h>
#include <omp.h>
int main() {
      #pragma omp parallel
      {
              printf("outer: omp_get_thread_num = %d omp_get_level = %d\n",
 omp_get_thread_num(), omp_get_level());
              #pragma omp parallel
              {
                  printf("inner: omp_get_thread_num = %d omp_get_level = %d\n",
 omp_get_thread_num(), omp_get_level());
              }
      }
}
```

Compiling and running the code gives the following output:

- Arm Compiler for Linux, building the executable `a1.out`:

  ```
  armclang -o a1.out -fopenmp threading.c
  OMP_NUM_THREADS=2 ./a1.out
  outer: omp_get_thread_num = 0 omp_get_level = 1
  inner: omp_get_thread_num = 0 omp_get_level = 2
  outer: omp_get_thread_num = 1 omp_get_level = 1
  inner: omp_get_thread_num = 0 omp_get_level = 2
  ```

- GCC, building the executable `g1.out`:

  ```
  gcc -o g1.out -fopenmp threading.c
  OMP_NUM_THREADS=2 ./g1.out
  outer: omp_get_thread_num = 0 omp_get_level = 1
  inner: omp_get_thread_num = 0 omp_get_level = 2
  outer: omp_get_thread_num = 1 omp_get_level = 1
  inner: omp_get_thread_num = 0 omp_get_level = 2
  ```

The program above reports the thread number and level of parallel nesting. Executables that are built with either GCC or Arm Compiler for Linux show the same behavior when `OMP_NUM_THREADS` is set to a single value (and all other settings use default values).

The example given earlier sets `OMP_NUM_THREADS=2` and the output shows that two threads are used for the outer parallel region. The nested parallel regions do not create any new threads:

Figure 3-1: No nested parallelism diagram.
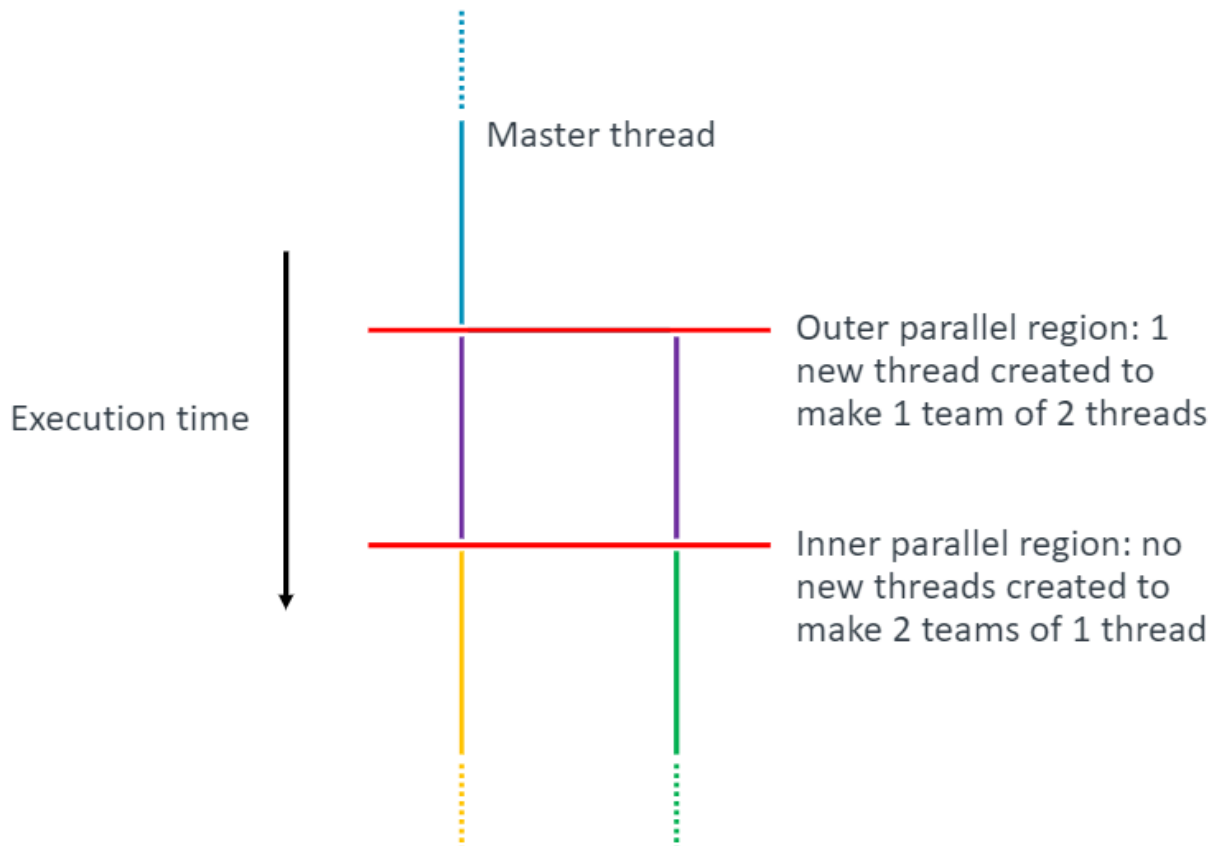
> **Note**
>
> The actual number of threads used during execution of your program might differ from the value specified in `OMP_NUM_THREADS`. The number of threads can differ if the number of threads is set explicitly in the code using the OpenMP API, or if a system-defined limit is encountered.

`OMP_NUM_THREADS` can also be set to a comma-separated list of values. Where a list of values is passed to `OMP_NUM_THREADS`, the values denote the number of threads to use at each level of nesting, starting from the outermost parallel region.

• Arm Compiler for Linux:

```
OMP_NUM_THREADS=2,2 ./a1.out
outer: omp_get_thread_num = 0 omp_get_level = 1
outer: omp_get_thread_num = 1 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
```

```
inner: omp_get_thread_num = 1 omp_get_level = 2
inner: omp_get_thread_num = 0 omp_get_level = 2
inner: omp_get_thread_num = 1 omp_get_level = 2
```

- GCC:

```
OMP_NUM_THREADS=2,2 ./g1.out
outer: omp_get_thread_num = 0 omp_get_level = 1
outer: omp_get_thread_num = 1 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
inner: omp_get_thread_num = 0 omp_get_level = 2
inner: omp_get_thread_num = 1 omp_get_level = 2
inner: omp_get_thread_num = 1 omp_get_level = 2
```

The examples above specify that the two parallel regions in the code can each use two threads. With both compilers, the executables create a new thread in each of the two inner parallel regions, enabling nested parallelism.

Figure 3-2: Nested parallelism diagram.

Note
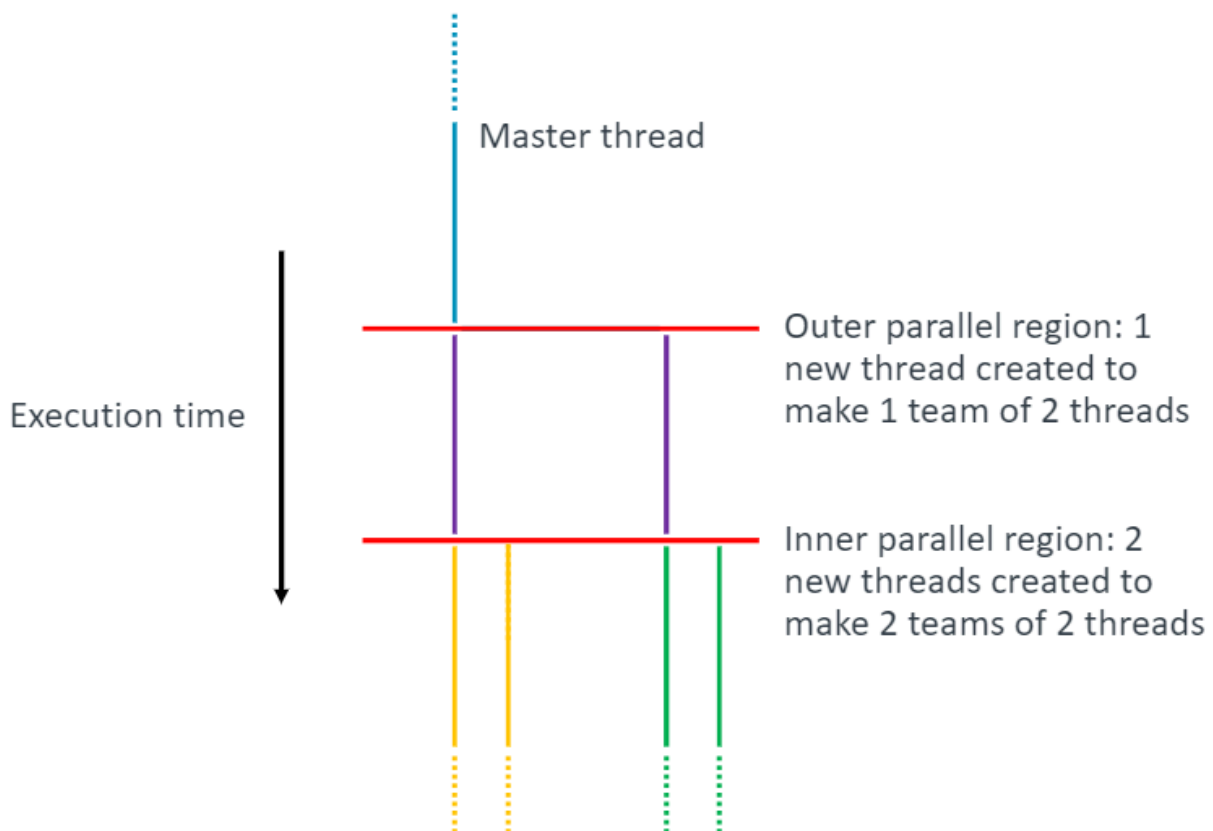
- Defaulting to nested parallelism was a change of behavior for executables linked to the OpenMP runtime in Arm Compiler for Linux version 20.0 and GCC version 11.0. In earlier compiler versions, the default was for nested parallelism to be disabled.

- The OMP_NESTED setting is being deprecated for OpenMP 5.0. Instead, we recommend that you use OMP_MAX_ACTIVE_LEVELS to control the depth of nesting.

Nested parallelism in Arm Performance Libraries is handled in the same way as shown in these examples; if an Arm Performance Libraries routine is called from a parallel region in your code, then the routine spawns threads in the same way as shown for the nested parallel region in the examples above.

Control the placement of threads

The value of the environment variable `OMP_PROC_BIND` affects how threads are assigned to cores on your system (also known as thread affinity). If `OMP_PROC_BIND=false` or is unset, then threads are unpinned. Unpinned threads might be migrated between cores in the system during execution, and thread migration will most likely degrade performance significantly.

Arm recommends setting `OMP_PROC_BIND` to either `true`, `close` or `spread`, as required:

- If `close` is set, then the OpenMP threads are pinned to cores close to the parent thread. `OMP_PROC_BIND=close` is useful where threads in a team are working on locally shared data. For example, if threads are pinned to neighboring cores there might be a performance benefit from the data being stored in a shared level of cache.

- If `spread` is set, then the OpenMP threads are pinned to cores that are distant from the parent thread. OMP_PROC_BIND=spread is useful to avoid contention on hardware resources. For example, if threads are working on large amounts of private data then there might be an advantage to using `spread`. Using `spread` can reduce contention on a shared level of cache or memory bandwidth.

- If `true` is set, thread migration is avoided and no affinity policy is specified.

- If `master` is set, all OpenMP threads in a team are pinned to the same core as the master thread.

To set the affinity policy separately for each level of nested parallelism, set `OMP_PROC_BIND` to a comma-separated list of the values described above.

> The values assigned to OpenMP environment variables are case insensitive.
>
> **Note**

The statements above describe how OpenMP threads are pinned to cores in the system. However, the OpenMP specification uses the term "place" to denote a hardware resource for which threads can have affinity. The environment variable `OMP_PLACES` allows you to define what is meant by a "place" in the system.

`OMP_PLACES` can be set to one of three pre-defined values: `threads`, `cores` or `sockets`. Setting `OMP_PLACES=threads` assigns OpenMP threads to hardware threads in the system. On a system where a single core supports multiple hardware threads (for example, Marvell ThunderX2 systems

with SMT>1), assigning OpenMP threads to hardware threads allows for the co-location of several threads in a single core.

If the value is set to `cores`, each OpenMP thread is assigned to a different core in the system, which might support more than one hardware thread.

If the value is set to `sockets`, each OpenMP thread is assigned to a single socket in the system, which contains multiple cores. Where `sockets` is set, the OpenMP threads might migrate in the assigned socket.

To more finely control the placement of OpenMP threads in your system, set `OMP_PLACES` to a list of numbers that indicate the IDs of hardware places in your system (typically hardware threads). There is a considerable amount of flexibility availability using `OMP_PLACES`, including the ability to exclude places from thread placement. If you are interested in this level of control, refer to the OpenMP specification and experiment on your system.

Simultaneous Multithreading (SMT)

With the high core counts on server scale AArch64 CPUs, you might experience an unexpected performance degradation of OpenMP codes because of the kernel swapping threads.

Arm recommends that you set the key environment variables (see previous section) and make an informed choice based on the parallel programming model and system configuration (for example, Simultaneous Multithreading (SMT) `SMT={1,2,4}`) being used for your code.

> **Note**
>
> For `SMT=1`, `threads` is equivalent to `cores`, but where SMT is > 1 you must decide if you want to pin to the local core (`threads`) or the physical core (`core`).

SMT=1

> Default settings (unset): `OMP_NUM_THREADS=<number of logical cores available on the system>`, `OMP_PROC_BIND=false`, and `OMP_PLACES=cores`.
>
> For example, in a 64-core system the number of logical cores (given by the product of the number of physical cores and the number of SMT threads) is 64. Therefore, `OMP_NUM_THREADS=64`.
>
> Job launches with all available OpenMP threads, each 'bound' to the full processor set (in other words, free to move between any physical core).
>
> - `OMP_PROC_BIND=close` and `OMP_PLACES=cores`
>
>   Job launches with all available OpenMP threads, each 'bound' to one core, filling sequentially: socket 0, then socket 1. The OpenMP threads are pinned to one physical core.
>
> - `OMP_PROC_BIND=close` and `OMP_PLACES=sockets`

Job launches with all available OpenMP threads, each 'bound' to one socket, filling sequentially: socket 0, then socket 1. The OpenMP threads are free to swap within that socket, before being evenly distributed between the sockets.

- `OMP_PROC_BIND=spread` and `OMP_PLACES=cores`

  Job launches with all available OpenMP threads, each 'bound' to one core and spreading out as far away from each other as possible.

- `OMP_PROC_BIND=spread` and `OMP_PLACES=sockets`

  Job launches with all available OpenMP threads, each 'bound' to one socket, filling sequentially: socket 0, then socket 1. The OpenMP threads spread out as far away from each other as possible within that socket, before being evenly distributed between the sockets.

SMT={2|4}

Default settings (unset): `OMP_NUM_THREADS=<number of logical cores available on the system>`, `OMP_PROC_BIND=false`, and `OMP_PLACES=cores`.

For example, in a 64-core system the number of logical cores is 256 cores. Therefore, `OMP_NUM_THREADS=256`.

Job launches with all available OpenMP threads, each 'bound' to the full processor set (in other words, free to move between either thread on any physical core).

- `OMP_PROC_BIND=close` and `OMP_PLACES=cores`

  Job launches with all available OpenMP threads, two (`SMT=2`) or four (`SMT=4`) per core, OpenMP threads pinned to these sets. Each OpenMP thread can migrate between all the hardware threads on a physical core. Each core belongs to a single OpenMP thread. Cores are filled sequentially: socket 0, then socket 1. All cores on socket are filled before moving onto the next socket.

- `OMP_PROC_BIND=close` and `OMP_PLACES=threads`

  Pins each process to one slot on one core. Available slots are filled, as with `OMP_PLACES=cores`.

- `OMP_PROC_BIND=close` and `OMP_PLACES=sockets`

  Pins each process to any thread on a socket, but fills all the cores (not all the slots) on one socket, before moving onto the next. In other words, each OpenMP thread gets a single hardware thread and are packed in. Behaves the same as `OMP_PROC_BIND=spread OMP_PLACES=sockets`.

- `OMP_PROC_BIND=spread` and `OMP_PLACES=cores`

  Job launches with all available OpenMP threads, each bound to a core and free to migrate between the two or four slots. OpenMP threads are spread between available sockets (in other words, one thread per core until physical cores are all in use).

- `OMP_PROC_BIND=spread` and `OMP_PLACES=sockets`

Pins each process to any thread on a socket, but fills all the cores (not all the slots) on one socket, before moving onto the next.

Example: Investigating an OpenMP where `SMT=1`

Note

This example uses a 64-core node and only 8 cores for illustration purposes.

---

1. Run your OpenMP application, for example `example-01`, and specify the number of OpenMP threads to run it on. For this example, 8 are specified:

```
KMP_AFFINITY=verbose OMP_NUM_THREADS=8  ./example-01
```

This gives you an output similar to:

```
...
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected:
  {0,1,2,...,62,63}
OMP: Info #156: KMP_AFFINITY: 64 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 32 cores/pkg x 1 threads/core
  (64 total cores)
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24047 thread 0 bound to OS proc set
  {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24048 thread 1 bound to OS proc set
  {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24049 thread 2 bound to OS proc set
  {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24050 thread 3 bound to OS proc set
  {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24051 thread 4 bound to OS proc set
  {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24052 thread 5 bound to OS proc set
  {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24053 thread 6 bound to OS proc set
  {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24054 thread 7 bound to OS proc set
  {0,1,2,...,62,63}
...
```

Where:

- All 64 (0 to 63) physical cores are shown, split over two sockets of 32 cores, and each physical core showing one thread per core.

- There are no bindings or mappings between logical cores and physical cores.

- The Process ID (PID) shows the full logical core set being available to each of the eight OpenMP threads. The full set of logical cores being available is because `OMP_PROC_BIND` is unset and defaults to `OMP_PROC_BIND=false`.

2. To control how the threads are distributed, set `OMP_PROC_BIND=true`:

```
KMP_AFFINITY=verbose OMP_PROC_BIND=true OMP_NUM_THREADS=8   ./example-01
```

This output is now similar to:

```
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,...,62,63}
...
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 2
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 3
...
OMP: Info #171: KMP_AFFINITY: OS proc 31 maps to package 0 core 31
OMP: Info #171: KMP_AFFINITY: OS proc 32 maps to package 1 core 0
...
OMP: Info #248: KMP_AFFINITY: pid 24055 tid 24055 thread 0 bound to OS proc set
  {0}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24056 thread 1 bound to OS proc set
  {8}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24057 thread 2 bound to OS proc set
  {16}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24058 thread 3 bound to OS proc set
  {24}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24059 thread 4 bound to OS proc set
  {32}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24060 thread 5 bound to OS proc set
  {40}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24061 thread 6 bound to OS proc set
  {48}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24062 thread 7 bound to OS proc set
  {56}
...
```

Where:

- The same 64 logical cores are available.

- Each logical core is now 'bound' (mapped) to a physical core because the `OMP_PROC_BIND` is set: logical cores 0-31 are mapped to physical cores 0-31, before the system cycles back to map logical cores 32-63 to physical cores 0-31 again.

- The Process ID (PID) shows the eight OpenMP threads being evenly distributed over the 64 logical cores: {0}, {8}, .., {56}. This spread distribution is because the combination of `OMP_PROC_BIND=true` and an unset `OMP_PLACES` results in `OMP_PROC_BIND=spread` effectively being set.

3. To bring the threads closer together, use `OMP_PROC_BIND=close`:

```
KMP_AFFINITY=verbose OMP_PROC_BIND=close OMP_NUM_THREADS=8   ./example-01
```

The output is now similar to:

```
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,...,62,63}
OMP: Info #156: KMP_AFFINITY: 64 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 32 cores/pkg x 1 threads/core
  (64 total cores)
...
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0
```

```
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 2
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 3
...
OMP: Info #171: KMP_AFFINITY: OS proc 31 maps to package 0 core 31
OMP: Info #171: KMP_AFFINITY: OS proc 32 maps to package 1 core 0
...
OMP: Info #248: KMP_AFFINITY: pid 24063 tid 24063 thread 0 bound to OS proc set
  {0}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24064 thread 1 bound to OS proc set
  {1}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24065 thread 2 bound to OS proc set
  {2}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24066 thread 3 bound to OS proc set
  {3}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24067 thread 4 bound to OS proc set
  {4}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24068 thread 5 bound to OS proc set
  {5}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24069 thread 6 bound to OS proc set
  {6}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24070 thread 7 bound to OS proc set
  {7}
...
```

Where:

- The same 64 logical cores are available.

- Each logical core is still mapped to a physical core because the `OMP_PROC_BIND` is set: logical cores 0-31 are mapped to physical cores 0-31, before the system cycles back to map logical cores 32-63 to physical cores 0-31 again.

- The Process ID (PID) shows the eight OpenMP threads being closely distributed over the first 8 of the 64 logical cores: {0}, {1}, .., {8}. This close distribution is because `OMP_PROC_BIND` is now set to `OMP_PROC_BIND=close`.

4. If your application has specific functions being set to specific OpenMP threads, it can be useful to weight the cores according to the computational intensity of these functions.

> **Note**
>
> This will vary from application to application.

To precisely distribute the OpenMP threads to specific logical cores, you can explicitly set these for `OMP_PLACES`. For example, set `OMP_PLACES={0:2}:8:4` to request that each OpenMP thread is given two logical cores, with eight multiples of two logical cores ({0:2}), spaced four logical cores apart:

```
KMP_AFFINITY=verbose OMP_PLACES={0:2}:8:4 OMP_NUM_THREADS=8  ./example-01
```

The output is similar to:

```
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,...,62,63}
...
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 2
```

```
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 3
...
OMP: Info #171: KMP_AFFINITY: OS proc 31 maps to package 0 core 31
OMP: Info #171: KMP_AFFINITY: OS proc 32 maps to package 1 core 0
...
OMP: Info #248: KMP_AFFINITY: pid 24071 tid 24071 thread 0 bound to OS proc set
  {0,1}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24072 thread 1 bound to OS proc set
  {4,5}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24073 thread 2 bound to OS proc set
  {8,9}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24074 thread 3 bound to OS proc set
  {12,13}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24075 thread 4 bound to OS proc set
  {16,17}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24076 thread 5 bound to OS proc set
  {20,21}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24077 thread 6 bound to OS proc set
  {24,25}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24078 thread 7 bound to OS proc set
  {28,29}
...
```

Where:

- The same 64 logical cores are available.

- Each logical core is still mapped to a physical core because the `OMP_PROC_PLACES` is set:
  logical cores 0-31 are mapped to physical cores 0-31, before the system cycles back to map
  logical cores 32-63 to physical cores 0-31 again.

- The Process ID (PID) shows the eight OpenMP threads being explicitly places onto two
  logical cores each, at a distance of 4 logical cores apart: {0,1}, {4,5}, .., {28,29}. This explicit
  placement is because `OMP_PLACES` is set to `OMP_PLACES={0:2}:8:4`.

Affinity in a hybrid environment: Open MPI

Hybrid applications are applications that employ multiple parallel programming models, for example
MPI and OpenMP. Running hybrid applications introduces a second layer of complexity. By default,
MPI runtimes provide options to control thread and task placement, and might set bindings for MPI
processes.

Note
This topic uses Open MPI as an example MPI implementation to discuss the
concepts being described. Other MPI implementations are available, such as MPICH
and MVAPICH2. For recipes to port your MPI implementation to Arm, see the
Porting and Tuning web page.

Some common Open MPI runtime options include:

`--bind-to [hwthread\|core\|socket]`

Controls what unit of hardware to bind threads to.

`--map-by [hwthread\|core\|slot][:PE=x]`

Controls the distribution of resources to MPI ranks. `x` in `PE=x` is how many processing
elements to give to each rank. For example, to give two tasks per socket using `--map-by`, use:

`--map-by ppr:2:socket.`

`--report-bindings`

> Report the MPI ranks bindings for launched processes. `report-bindings` is useful for obtaining information on how tasks and OpenMP threads are mapped and bound at run-time.

`--display-map`

> Display a table showing the mapped location of each process ahead of launch.

Useful for obtaining information on how tasks and OpenMP threads that are mapped and bound at run-time.

`-use-hwthread-cpus`

> Treat SMT hardware threads as distinct cores.

Example: Investigating a hybrid OpenMPI and OpenMP application where `SMT=1`

> **Note**
>
> This example uses a 64-core node and only launches 4 Open MPI processes per node for illustration purposes.

---

1. Run your hybrid application, for example `example-02`, and specify the number of OpenMP threads (in this case four) and Open MPI nodes (in this case four) to run it on. This example sets the Open MPI options `--map-by slot:PE=4` to distribute the MPI ranks over slots, and use 4 processing elements. `-bind-to core` binds the MPI processes to logical cores:

   ```
   KMP_AFFINITY=verbose  OMP_NUM_THREADS=4 mpirun -np 4 --map-by slot:PE=4 --bind-to
   core --report-bindings  ./example-02
   ```

   This give you the following extracts from the output:

   ```
   MCW rank 0 bound to socket 0[core 0[hwt 0]],
     socket 0[core 1[hwt 0]], socket 0[core 2[hwt 0]], socket 0[core 3[hwt 0]]:
     [B/B/B/B/./././././././././././././././././././././././././././././././.]
   [./././././././././././././././././././././././././././././././././././.]
   MCW rank 1 bound to socket 0[core 4[hwt 0]],
     socket 0[core 5[hwt 0]], socket 0[core 6[hwt 0]], socket 0[core 7[hwt 0]]:
     [././././B/B/B/B/./././././././././././././././././././././././././././.]
   [./././././././././././././././././././././././././././././././././././.]
   MCW rank 2 bound to socket 0[core 8[hwt 0]],
     socket 0[core 9[hwt 0]], socket 0[core 10[hwt 0]], socket 0[core 11[hwt 0]]:
     [././././././././B/B/B/B/./././././././././././././././././././././././.]
   [./././././././././././././././././././././././././././././././././././.]
   MCW rank 3 bound to socket 0[core 12[hwt 0]],
     socket 0[core 13[hwt 0]], socket 0[core 14[hwt 0]], socket 0[core 15[hwt 0]]:
     [././././././././././././B/B/B/B/./././././././././././././././././././.]
   [./././././././././././././././././././././././././././././././././././.]
   ...
   OMP: Info #179: KMP_AFFINITY: 1 packages x 4 cores/pkg x 1 threads/core (4 total
   cores)
   OMP: Info #248: KMP_AFFINITY: pid 24635 tid 24635 thread 0 bound to OS proc set
   {0,1,2,3}
   OMP: Info #248: KMP_AFFINITY: pid 24635 tid 24684 thread 1 bound to OS proc set
   {0,1,2,3}
   OMP: Info #248: KMP_AFFINITY: pid 24635 tid 24687 thread 2 bound to OS proc set
   {0,1,2,3}
   OMP: Info #248: KMP_AFFINITY: pid 24635 tid 24690 thread 3 bound to OS proc set
   {0,1,2,3}
   ```

```
OMP: Info #248: KMP_AFFINITY: pid 24636 tid 24636 thread 0 bound to OS proc set
  {4,5,6,7}
OMP: Info #248: KMP_AFFINITY: pid 24636 tid 24689 thread 1 bound to OS proc set
  {4,5,6,7}
OMP: Info #248: KMP_AFFINITY: pid 24636 tid 24691 thread 2 bound to OS proc set
  {4,5,6,7}
OMP: Info #248: KMP_AFFINITY: pid 24636 tid 24693 thread 3 bound to OS proc set
  {4,5,6,7}
OMP: Info #248: KMP_AFFINITY: pid 24637 tid 24637 thread 0 bound to OS proc set
  {8,9,10,11}
OMP: Info #248: KMP_AFFINITY: pid 24637 tid 24692 thread 1 bound to OS proc set
  {8,9,10,11}
OMP: Info #248: KMP_AFFINITY: pid 24637 tid 24694 thread 2 bound to OS proc set
  {8,9,10,11}
OMP: Info #248: KMP_AFFINITY: pid 24637 tid 24695 thread 3 bound to OS proc set
  {8,9,10,11}
OMP: Info #248: KMP_AFFINITY: pid 24639 tid 24639 thread 0 bound to OS proc set
  {12,13,14,15}
OMP: Info #248: KMP_AFFINITY: pid 24639 tid 24685 thread 1 bound to OS proc set
  {12,13,14,15}
OMP: Info #248: KMP_AFFINITY: pid 24639 tid 24686 thread 2 bound to OS proc set
  {12,13,14,15}
OMP: Info #248: KMP_AFFINITY: pid 24639 tid 24688 thread 3 bound to OS proc set
  {12,13,14,15}
...
```

Where:

- Each MPI rank is bound to a set of 4 logical cores, sequentially filling the first 16 of the available 64 logical cores.

- The Process ID (PID) shows each MPI process (0-3) receiving an OpenMP thread (four of each process) each with the OpenMP thread able to be placed on any of the next 4 logical cores in the core set: ranks 0 get {0,1,2,3},..., {12,13,14,15}, ..., ranks 3 get {0,1,2,3},..., {12,13,14,15}.

- The binding of the OpenMP threads cycles over 0-15 for each processing element is because `--bind-to core` is set.

2. To introduce more control into the location of the OpenMP threads, bind the OpenMP threads to cores. Set `OMP_PROC_BIND=true`:

```
KMP_AFFINITY=verbose  OMP_PROC_BIND=true OMP_NUM_THREADS=4 mpirun -np 4 --map-by
  slot:PE=4 --bind-to core --report-bindings  ./example-02
```

This give you the following extracts from the output:

```
MCW rank 0 bound to socket 0[core 0[hwt 0]],
  socket 0[core 1[hwt 0]], socket 0[core 2[hwt 0]], socket 0[core 3[hwt 0]]:
  [B/B/B/B/./././././././././././././././././././././././././././.]
[./././././././././././././././././././././././././././././././.]
MCW rank 1 bound to socket 0[core 4[hwt 0]],
  socket 0[core 5[hwt 0]], socket 0[core 6[hwt 0]], socket 0[core 7[hwt 0]]:
  [./././/B/B/B/B/./././././././././././././././././././././././.]
[./././././././././././././././././././././././././././././././.]
MCW rank 2 bound to socket 0[core 8[hwt 0]],
  socket 0[core 9[hwt 0]], socket 0[core 10[hwt 0]], socket 0[core 11[hwt 0]]:
  [./././././././/B/B/B/B/./././././././././././././././././././.]
[./././././././././././././././././././././././././././././././.]
MCW rank 3 bound to socket 0[core 12[hwt 0]],
  socket 0[core 13[hwt 0]], socket 0[core 14[hwt 0]], socket 0[core 15[hwt 0]]:
  [./././././././././././/B/B/B/B/./././././././././././././././.]
[./././././././././././././././././././././././././././././././.]
```

```
...
OMP: Info #248: KMP_AFFINITY: pid 24780 tid 24780 thread 0 bound to OS proc set
  {0}
OMP: Info #248: OMP_PROC_BIND: pid 24780 tid 24846 thread 1 bound to OS proc set
  {1}
OMP: Info #248: OMP_PROC_BIND: pid 24780 tid 24850 thread 2 bound to OS proc set
  {2}
OMP: Info #248: OMP_PROC_BIND: pid 24780 tid 24853 thread 3 bound to OS proc set
  {3}
OMP: Info #248: KMP_AFFINITY: pid 24781 tid 24781 thread 0 bound to OS proc set
  {4}
OMP: Info #248: OMP_PROC_BIND: pid 24781 tid 24847 thread 1 bound to OS proc set
  {5}
OMP: Info #248: OMP_PROC_BIND: pid 24781 tid 24849 thread 2 bound to OS proc set
  {6}
OMP: Info #248: OMP_PROC_BIND: pid 24781 tid 24856 thread 3 bound to OS proc set
  {7}
OMP: Info #248: KMP_AFFINITY: pid 24782 tid 24782 thread 0 bound to OS proc set
  {8}
OMP: Info #248: OMP_PROC_BIND: pid 24782 tid 24845 thread 1 bound to OS proc set
  {9}
OMP: Info #248: OMP_PROC_BIND: pid 24782 tid 24851 thread 2 bound to OS proc set
  {10}
OMP: Info #248: OMP_PROC_BIND: pid 24782 tid 24855 thread 3 bound to OS proc set
  {11}
OMP: Info #248: KMP_AFFINITY: pid 24783 tid 24783 thread 0 bound to OS proc set
  {12}
OMP: Info #248: OMP_PROC_BIND: pid 24783 tid 24848 thread 1 bound to OS proc set
  {13}
OMP: Info #248: OMP_PROC_BIND: pid 24783 tid 24852 thread 2 bound to OS proc set
  {14}
OMP: Info #248: OMP_PROC_BIND: pid 24783 tid 24854 thread 3 bound to OS proc set
  {15}
...
```

Where:

- Each MPI rank is still bound to a set of 4 logical cores, sequentially filling the first 16 of the available 64 logical cores.

- The Process ID (PID) shows each MPI process (0-3) receiving an OpenMP thread (four of each process) each with the OpenMP thread bound to the most spread available logical core: ranks 0 get {0},{4},{8},{12},..., ranks 3 get {3},{7},{11},{15}. The core-bound distribution of OpenMP threads is because `OMP_PROC_BIND=true` is set.

3. Similar to the OpenMP example, the distribution of th OpenMP threads can be forced to be placed closer by setting `OMP_PROC_BIND=close`.

```
KMP_AFFINITY=verbose  OMP_PROC_BIND=close OMP_NUM_THREADS=4 mpirun -np 4 --map-by
  slot:PE=16 --bind-to core --report-bindings  ./example-02
```

Giving you the output:

```
MCW rank 0 bound to socket 0[core 0[hwt 0]],
  socket 0[core 1[hwt 0]], socket 0[core 2[hwt 0]], socket 0[core 3[hwt 0]],
  socket 0[core 4[hwt 0]], socket 0[core 5[hwt 0]], socket 0[core 6[hwt 0]],
  socket 0[core 7[hwt 0]], socket 0[core 8[hwt 0]], socket 0[core 9[hwt 0]],
  socket 0[core 10[hwt 0]], socket 0[core 11[hwt 0]], socket 0[core 12[hwt 0]],
  socket 0[core 13[hwt 0]], socket 0[core 14[hwt 0]], socket 0[core 15[hwt 0]]:
  [B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/./././././././././././././././././.]
[./././././././././././././././././././././././././././././././././.]
MCW rank 1 bound to socket 0[core 16[hwt 0]],
  socket 0[core 17[hwt 0]], socket 0[core 18[hwt 0]], socket 0[core 19[hwt 0]],
  socket 0[core 20[hwt 0]], socket 0[core 21[hwt 0]], socket 0[core 22[hwt 0]],
```

```
   socket 0[core 23[hwt 0]], socket 0[core 24[hwt 0]], socket 0[core 25[hwt 0]],
   socket 0[core 26[hwt 0]], socket 0[core 27[hwt 0]], socket 0[core 28[hwt 0]],
   socket 0[core 29[hwt 0]], socket 0[core 30[hwt 0]], socket 0[core 31[hwt 0]]:
   [./././././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B]
[./././././././././././././././././././././././././././././././././.]
MCW rank 2 bound to socket 1[core 32[hwt 0]],
   socket 1[core 33[hwt 0]], socket 1[core 34[hwt 0]], socket 1[core 35[hwt 0]],
   socket 1[core 36[hwt 0]], socket 1[core 37[hwt 0]], socket 1[core 38[hwt 0]],
   socket 1[core 39[hwt 0]], socket 1[core 40[hwt 0]], socket 1[core 41[hwt 0]],
   socket 1[core 42[hwt 0]], socket 1[core 43[hwt 0]], socket 1[core 44[hwt 0]],
   socket 1[core 45[hwt 0]], socket 1[core 46[hwt 0]], socket 1[core 47[hwt 0]]:
   [./././././././././././././././././././././././././././././.][B/B/B/B/B/B/
B/B/B/B/B/B/B/B/B/B/././././././././././././.]
MCW rank 3 bound to socket 1[core 48[hwt 0]],
 socket 1[core 49[hwt 0]], socket 1[core 50[hwt 0]], socket 1[core 51[hwt 0]],
 socket 1[core 52[hwt 0]], socket 1[core 53[hwt 0]], socket 1[core 54[hwt 0]],
 socket 1[core 55[hwt 0]], socket 1[core 56[hwt 0]], socket 1[core 57[hwt 0]],
 socket 1[core 58[hwt 0]], socket 1[core 59[hwt 0]], socket 1[core 60[hwt 0]],
 socket 1[core 61[hwt 0]], socket 1[core 62[hwt 0]], socket 1[core 63[hwt 0]]:
   [./././././././././././././././././././././././././././././././.]
[./././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B]
...
OMP: Info #248: KMP_AFFINITY: pid 24878 tid 24878 thread 0 bound to OS proc set
   {0}
OMP: Info #248: OMP_PROC_BIND: pid 24878 tid 24927 thread 1 bound to OS proc set
   {1}
OMP: Info #248: OMP_PROC_BIND: pid 24878 tid 24927 thread 2 bound to OS proc set
   {2}
OMP: Info #248: OMP_PROC_BIND: pid 24878 tid 24935 thread 3 bound to OS proc set
   {3}
OMP: Info #248: KMP_AFFINITY: pid 24879 tid 24879 thread 0 bound to OS proc set
   {16}
OMP: Info #248: OMP_PROC_BIND: pid 24879 tid 24936 thread 1 bound to OS proc set
   {17}
OMP: Info #248: OMP_PROC_BIND: pid 24879 tid 24937 thread 2 bound to OS proc set
   {18}
OMP: Info #248: OMP_PROC_BIND: pid 24879 tid 24938 thread 3 bound to OS proc set
   {19}
OMP: Info #248: KMP_AFFINITY: pid 24881 tid 24881 thread 0 bound to OS proc set
   {32}
OMP: Info #248: OMP_PROC_BIND: pid 24881 tid 24928 thread 1 bound to OS proc set
   {33}
OMP: Info #248: OMP_PROC_BIND: pid 24881 tid 24930 thread 2 bound to OS proc set
   {34}
OMP: Info #248: OMP_PROC_BIND: pid 24881 tid 24931 thread 3 bound to OS proc set
   {35}
OMP: Info #248: KMP_AFFINITY: pid 24882 tid 24882 thread 0 bound to OS proc set
   {48}
OMP: Info #248: OMP_PROC_BIND: pid 24882 tid 24929 thread 1 bound to OS proc set
   {49}
OMP: Info #248: OMP_PROC_BIND: pid 24882 tid 24932 thread 2 bound to OS proc set
   {50}
OMP: Info #248: OMP_PROC_BIND: pid 24882 tid 24934 thread 3 bound to OS proc set
   {51}
...
```

Where:

- Each MPI rank is uses 16 processing elements bound to a set of 16 logical cores, sequentially filling the available 64 logical cores.

- The Process ID (PID) shows each MPI process (0-3) having four OpenMP threads (four of each process) where the OpenMP threads are bound to the closest available logical core: ranks 0 get {0},{16},{32},{48},..., ranks 3 get {3},{19},{35},{51}. The close distribution of OpenMP threads is because `OMP_PROC_BIND=close` is set. The closest logical cores are 16 apart because of the 16 processing elements.

4. Setting `OMP_PROC_BIND=spread` shows how the threads can be distributed as far away from each other:

```
KMP_AFFINITY=verbose  OMP_PROC_BIND=spread OMP_NUM_THREADS=4 mpirun -np 4 --map-
by slot:PE=16 --bind-to core --report-bindings  ./example-02
```

Giving you the output:

```
MCW rank 0 bound to socket 0[core 0[hwt 0]],
  socket 0[core 1[hwt 0]], socket 0[core 2[hwt 0]], socket 0[core 3[hwt 0]],
  socket 0[core 4[hwt 0]], socket 0[core 5[hwt 0]], socket 0[core 6[hwt 0]],
  socket 0[core 7[hwt 0]], socket 0[core 8[hwt 0]], socket 0[core 9[hwt 0]],
  socket 0[core 10[hwt 0]], socket 0[core 11[hwt 0]], socket 0[core 12[hwt 0]],
  socket 0[core 13[hwt 0]], socket 0[core 14[hwt 0]], socket 0[core 15[hwt 0]]:
  [B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/././././././././././././././././.]
[./././././././././././././././././././././././././././././././././.]
MCW rank 1 bound to socket 0[core 16[hwt 0]],
  socket 0[core 17[hwt 0]], socket 0[core 18[hwt 0]], socket 0[core 19[hwt 0]],
  socket 0[core 20[hwt 0]], socket 0[core 21[hwt 0]], socket 0[core 22[hwt 0]],
  socket 0[core 23[hwt 0]], socket 0[core 24[hwt 0]], socket 0[core 25[hwt 0]],
  socket 0[core 26[hwt 0]], socket 0[core 27[hwt 0]], socket 0[core 28[hwt 0]],
  socket 0[core 29[hwt 0]], socket 0[core 30[hwt 0]], socket 0[core 31[hwt 0]]:
  [./././././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B]
[./././././././././././././././././././././././././././././././././.]
MCW rank 2 bound to socket 1[core 32[hwt 0]],
  socket 1[core 33[hwt 0]], socket 1[core 34[hwt 0]], socket 1[core 35[hwt 0]],
  socket 1[core 36[hwt 0]], socket 1[core 37[hwt 0]], socket 1[core 38[hwt 0]],
  socket 1[core 39[hwt 0]], socket 1[core 40[hwt 0]], socket 1[core 41[hwt 0]],
  socket 1[core 42[hwt 0]], socket 1[core 43[hwt 0]], socket 1[core 44[hwt 0]],
  socket 1[core 45[hwt 0]], socket 1[core 46[hwt 0]], socket 1[core 47[hwt 0]]:
  [./././././././././././././././././././././././././././././././././.][B/B/B/B/B/B/
B/B/B/B/B/B/B/B/B/B/././././././././././././././././.]
MCW rank 3 bound to socket 1[core 48[hwt 0]],
  socket 1[core 49[hwt 0]], socket 1[core 50[hwt 0]], socket 1[core 51[hwt 0]],
  socket 1[core 52[hwt 0]], socket 1[core 53[hwt 0]], socket 1[core 54[hwt 0]],
  socket 1[core 55[hwt 0]], socket 1[core 56[hwt 0]], socket 1[core 57[hwt 0]],
  socket 1[core 58[hwt 0]], socket 1[core 59[hwt 0]], socket 1[core 60[hwt 0]],
  socket 1[core 61[hwt 0]], socket 1[core 62[hwt 0]], socket 1[core 63[hwt 0]]:
  [./././././././././././././././././././././././././././././././././.]
[./././././././././././././././.B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B]
...
OMP: Info #248: KMP_AFFINITY: pid 24977 tid 24977 thread 0 bound to OS proc set
  {0}
OMP: Info #248: OMP_PROC_BIND: pid 24977 tid 25028 thread 1 bound to OS proc set
  {4}
OMP: Info #248: OMP_PROC_BIND: pid 24977 tid 25031 thread 2 bound to OS proc set
  {8}
OMP: Info #248: OMP_PROC_BIND: pid 24977 tid 25036 thread 3 bound to OS proc set
  {12}
OMP: Info #248: KMP_AFFINITY: pid 24978 tid 24978 thread 0 bound to OS proc set
  {16}
OMP: Info #248: OMP_PROC_BIND: pid 24978 tid 25027 thread 1 bound to OS proc set
  {20}
OMP: Info #248: OMP_PROC_BIND: pid 24978 tid 25030 thread 2 bound to OS proc set
  {24}
OMP: Info #248: OMP_PROC_BIND: pid 24978 tid 25035 thread 3 bound to OS proc set
  {28}
OMP: Info #248: KMP_AFFINITY: pid 24979 tid 24979 thread 0 bound to OS proc set
  {32}
OMP: Info #248: OMP_PROC_BIND: pid 24979 tid 25025 thread 1 bound to OS proc set
  {36}
OMP: Info #248: OMP_PROC_BIND: pid 24979 tid 25029 thread 2 bound to OS proc set
  {40}
OMP: Info #248: OMP_PROC_BIND: pid 24979 tid 25033 thread 3 bound to OS proc set
  {44}
```

```
OMP: Info #248: KMP_AFFINITY: pid 24980 tid 24980 thread 0 bound to OS proc set
  {48}
OMP: Info #248: OMP_PROC_BIND: pid 24980 tid 25026 thread 1 bound to OS proc set
  {52}
OMP: Info #248: OMP_PROC_BIND: pid 24980 tid 25032 thread 2 bound to OS proc set
  {56}
OMP: Info #248: OMP_PROC_BIND: pid 24980 tid 25034 thread 3 bound to OS proc set
  {60}
...
```

Where:

- Each MPI rank is uses 16 processing elements bound to a set of 16 logical cores, sequentially filling the available 64 logical cores.

- The Process ID (PID) shows each MPI process (0-3) having four OpenMP threads (four of each process) where the OpenMP threads are bound to the furthest available logical core: ranks 0 get {0},{16},{32},{48},..., ranks 3 get {12},{28},{44},{60}. The spread distribution of OpenMP threads is because `OMP_PROC_BIND` is set to `spread`. The furthermost logical cores are 4 apart.

lstopo

`lstopo` can provide a simple representation of where the available hardware threads are located. In other words, it is convenient for deciphering the numbering scheme.

To install and run `lstopo`:

1. Install hwloc and hwloc-gui. Use a suitable package installer for your OS (you might need root permissions).

   For example, on a CentOS or RedHat system using `yum`:

   ```
   yum install hwloc hwloc-gui -y
   ```

2. Run `lstopo`:
   - To run using the GUI, use:

     ```
     lstopo -p
     ```

   - To generate a pdf, use:

     ```
     lstopo -p --output-format pdf > topology.pdf
     ```

   - To generate a text output, use:

     ```
     lstopo -p --output-format console
     ```

Tips for application porting and optimization

- If your application uses OpenMP, Arm recommends that you set key OpenMP environment variables like `OMP_NUM_THREADS` and `OMP_PROC_BIND`.

- To identify the optimum choices, run your application on a number of core counts and using various values for `OMP_PROC_BIND` and `OMP_PLACES`.

- To profile the application and identify any significant OpenMP overheads and bottlenecks, use Arm MAP.

- Set your key environment variables. Make an informed choice based on the parallel programming model and system configuration (for example `SMT={1,2,4}`, or using numactl) being used for your code.

- To understand the thread placement on your system and the impact on run-time, so that you can choose the best configuration for your application, set `KMP_AFFINITY=verbose` and use `lstopo`.

- Consider the binding options available in the MPI distribution you are using.

4 Compiler Migration Guides

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

4.1 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.

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 4-1: GNU and Arm Compiler commands

	Arm
Version details	`armflang --version`

Help and documentation	`armflang --help`
	
	`man armflang`

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` option, as detailed in the next section.

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

Arm hardware options

GCC and Arm C/C++/Fortran Compiler, have three hardware compiler options in common: `-march`, `-mtune`, and `-mcpu`:

- `-march=X`: Tells the compiler that X is the minimal architecture the binary must run on. The compiler is free to use architecture-specific instructions. This option 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 to optimize for microarchitecture X, but does not allow the compiler to change the ABI or make assumptions about available instructions. This option has the more-or-less the same meaning on Arm and x86.

- `-mcpu=X`: On Arm, this option is a combination of `-march` and `-mtune`. It simultaneously specifies the target architecture and optimizes for a given microarchitecture. On x86, this option is a deprecated synonym for `-mtune`.

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

> **Note**
>
> Arm C/C++/Fortran Compiler does not support the use of `-march=native`. To aid portability, GCC on AArch64 does support the use of `-march=native`.

These compiler options control binary code generation. Correctly using these options can greatly improve run-time 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 that is denoted by the `-On` option and similar options, as detailed in the **Commonly used options** and **Optimization compiler options** sections in each compiler guide.

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, the extended precision BLAS).

- LAPACK: A comprehensive package of higher level linear algebra routines. To find out what the latest version of LAPACK that is supported in Arm Performance Libraries is, see Arm Performance Libraries.

- FFT functions: a set of Fast Fourier Transform routines for real and complex data using the FFTW interface.

- Sparse linear algebra.

- libamath: a subset of libm, which is a set of optimized mathematical functions.

`armpl` provides a simple interface for selecting thread-parallelism and architectural tuning. Arm Performance Libraries also provides optimized versions of the C mathematical functions, tuned scalar and vector implementations of Fortran math intrinsics, and auto-vectorization of mathematical functions.

Combining `-armpl` with the `-mcpu=native|<target>` or `-march=<architecture>` armflang options enables the compiler to find the appropriate Arm Performance Libraries header files during compilation, and the compatible libraries during linking. Both `-armpl` and either the `-mcpu` or `-march` options are required to link to the correct version of Arm Performance Libraries.

Note

- If your build process compiles and links as two separate steps, ensure that you add the same `-armpl` and `-m{cpu|arch}` options to both the compile and link commands.

- If you compile (and link) on a target that is different to the target where you will run the application, compile the program so that it is suitable to run on any Armv8-A-based or Armv8-A-based SVE system, using `-march=armv8a` or `-march=armv8-a+sve`, respectively. You can also use `-march=armv8-a+sve` if you will use SVE emulation software, for example Arm Instruction Emulator, to run the program on an Armv8-A-based system.

For more information on Arm Performance Libraries, see Get started with Arm Performance Libraries.

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 4-2: 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 no support for `OMP SIMD` clauses.

PREFETCH	`!$MEM PREFETCH` `<var_1>[,<var_2>[,...]]`	Tells the compiler to generate prefetch instructions to fetch elements and load them in the data cache, ahead of their first use. Users can provide a prefetch distance. Prefetching elements can improve performance by reducing main memory latency.

Directive	Usage	Description
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 options -02 or higher.

Generating Position Independent Code (PIC) with fPIC on AArch64

The generation of Position Independent Code (PIC) is typically required for building shared libraries. Supplying the command line option -fPIC at compile time instructs armflang to generate Position Independent Code (PIC), and is generally 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 users.

However, while the use of -fpic is often interchangeable with -fPIC on x86, it is 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, -fpic is not considered to be portable between x86_64 and AArch64 architectures.

Allocating stack variables

- **Thread-safe recursion**

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

  Use the -frecursive options when compiling a procedure that:
  - Has no OpenMP elements and is not compiled using the -fopenmp option.
  - Is called from within an OpenMP parallel region in source and is compiled with the -fopenmp option.

- **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 option, 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.

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 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 C/C++/Fortran Compiler versions that are earlier than 22.0.2 limited line lengths to 264 characters. Using compiler macros in versions that are earlier than 22.0.2 can lead to the generation of source lines longer than 264 characters at compile time.

Language extensions

There are several common extensions to the Fortran language which are typically supported by many existing compilers, generally for legacy reasons, including armflang. Often, 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 4-3: 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.
`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.
`GETARG(pos, arg)`	Subroutine call which returns the `pos`-th argument that is passed on the command line when the program 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.

Extension	Purpose	Standard-compliant alternative	Notes
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.

For more information on supported language extensions, see the Fortran intrinsics chapter in the Arm Fortran Compiler Reference Guide.

Pre-defined macros

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

Table 4-4: Pre-defined macros

Macro	Value	Purpose
__ARM_ARCH	INTEGER	Defined as an integer value and expands to 8 to indicate that the system implements v8 of the Armv8-A architecture. Selection of architecture-dependent source at compile time.
__ARM_LINUX_COMPILER__	1	Defined as an integer value and expands to 1 to indicate Arm Compiler for Linux.
__ARM_LINUX_COMPILER_BUILD__	INTEGER	Defined as an integer value and expands to the Arm Compiler for Linux build number.
__armclang_major__	INTEGER	Defined as an integer value and expands to the Arm Compiler for Linux major version number.
__armclang_minor__	INTEGER	Defined as an integer value and expands to the Arm Compiler for Linux minor version number.
__armclang_version__	STRING	Defined as a string value and expands to the full Arm Compiler for Linux version number.
__FLANG	1	Defined as an integer value and expands to 1 to indicate a FLANG-derived compiler. __FLANG is often included in Makefiles that support flang compilers.

> **Note**
>
> The preceding list is not exhaustive, to read about more predefined macros that are available to armflang, see: https://developer.arm.com/documentation/101380/latest/Fortran-language-reference/Predefined-macro-support.

Detailed compiler options

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

Understand the optimization choices the compiler makes

Arm C/C++/Fortran Compiler incorporates two tools to help you better understand the optimization decisions that it makes:

Arm Optimization Report

Arm Optimization Report is a new feature of Arm Compiler for Linux version 20.0 that builds upon the llvm-opt-report tool available in open-source LLVM. The new Arm Optimization Report feature makes it easier to see what optimization decisions the compiler is making about unrolling, vectorizing, and interleaving, in-line with your source code.

To enable Arm Optimization Report:

1. At compile time, add the `-fsave-optimization-record` to the command line.

   A `<filename>.opt.yaml` report is generated by the compiler, where `<filename>` is the name of the binary.

2. Use Arm Optimization Report (`arm-opt-report`) to inspect the `<filename>.opt.yaml` report as augmented source code:

   ```
   arm-opt-report <filename>.opt.yaml
   ```

   The annotated source code appears in the terminal.

For more information, refer to the Arm Optimization Reports documentation in the Arm Fortran Compiler reference guide.

Optimization Remarks

Optimization Remarks can be used to see which code has been inlined or to understand why a loop has not been vectorized.

To enable Optimization Remarks, pass one or more of the following `-Rpass` options at compile time:

Table 4-5: Options to enable Optimization Remarks

`-Rpass` options	Description
`-Rpass=<regexp>`	To request information about what Arm C/C++/Fortran Compiler has optimized.
`-Rpass-analysis=<regexp>`	To request information about what Arm C/C++/Fortran Compiler has analyzed.
`-Rpass-missed=<regexp>`	To request information about what Arm C/C++/Fortran 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.

For example, to get actionable information on which loops can and cannot be vectorized at compile time, pass:

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

> **Note**
> - Optimization Remarks are piped to `stdout` at compile time.

For more information on optimization remarks, see the Fortran compiler reference guides.

Further resources

Standards compliance

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

Additional information

- Arm Fortran Compiler Developer and Reference Guide
- Statement support in Arm Fortran Compiler
- Intrinsic support in Arm Fortran Compiler
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

4.2 armflang for gfortran users

The reference versions used in this guide are:

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

Invoking the compiler

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

Table 4-6: Invoking the compiler

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

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

Table 4-7: Accessing version information and documentation

	GCC	Arm
Version details	`gfortran --version`	`armflang --version`

Help and documentation	`gfortran --help`
	
	`man gfortran`
		`armflang --help`
		
		`man armflang`

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 4-8: 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 the Compiler options for the Arm Fortran Compiler.

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 allocated (or reallocated) 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: allocation (or reallocation) 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.

`-D`*macro*=*value*	`-D`*macro*=*value*	Set macro to *value*.
`-L`*directory*	`-L`*directory*	Add *directory* to the include search path.
`-l`*lib*	`-l`*lib*	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 64-bit (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 64-bit (KIND=8).
`-frecord-marker=n`	N/A	Length of record markers for unformatted files. n 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.

GCC	Arm	Description

`-fpic`
 	
`-fPIC`
	`-fpic`
	
	`-fPIC`
		Generate Position Independent Code (PIC).
		
		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 use a Global Offset Table (GOT) limited to 32kiB.
		
		Arm C/C++/Fortran Compiler treats `-fpic` as equivalent to `-fPIC`. To increase code portability, Arm recommends using `-fPIC` when compiling with Arm C/C++/Fortran Compiler.
		
		For more information on the use of `-fpic` and `-fPIC` on AArch64, see the Note about building Position Independent Code PIC on AArch64.

Optimization compiler options

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

Table 4-9: 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.

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	`-ffinite-maths-only`	Enable optimizations that ignore the possibility of NaNs and Infs.

Language extensions

For information on supported language extensions, see the Arm Fortran Compiler Reference Guide.

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.

Further resources

Standards compliance

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

Additional information

- Arm Fortran Compiler Developer and Reference Guide
- Statement support in Arm Fortran Compiler
- Intrinsic support in Arm Fortran Compiler
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

4.3 armflang for ifort users

The reference versions used in this guide are:

- Intel Fortran Compiler 17.0.1
- Arm® Fortran Compiler

Invoking the compiler

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

Table 4-10: Invoking the compiler

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

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

Table 4-11: Accessing version information and documentation

	Intel	Arm
Version details	`ifort --version`	`armflang --version`

Help and documentation	`ifort --help`
	
	`man ifort`
		`armflang --help`
		
		`man armflang`

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 4-12: 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.

`-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 the Compiler options for 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.

Intel	Arm	Description

`-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.

`-D`*macro=value*	`-D`*macro=value*	Set *macro* to value.
`-L`*directory*	`-L`*directory*	Add *directory* to the include search path.
`-l`*lib*	`-l`*lib*	Search for the library *lib* when linking.

`-real-size 64`
 	
`-r8`
	`-r8`	Set the default KIND for real and complex declarations, constants, functions, and intrinsics to 64-bit (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 64-bit (KIND=8).

`-fpic`
 	
`-fPIC`
	`-fpic`
	
	`-fPIC`
		Generate Position Independent Code (PIC).
		
		Arm C/C++/Fortran Compiler and ICC both treat `-fpic` as equivalent to `-fPIC`. To increase code portability, Arm recommends using `-fPIC` when compiling with Arm C/C++/Fortran Compiler.
		
		For more information on the use of `-fpic` and `-fPIC` on AArch64, see the Note about building Position Independent Code PIC on AArch64.

Optimization compiler options

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

Table 4-13: 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`

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.

Description	Syntax	Notes
Finite maths	-ffinite-maths-only	Enable optimizations that ignore the possibility of NaNs and Infs.

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`.

Further resources

Standards compliance

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

Additional information

- Arm Fortran Compiler Developer and Reference Guide
- Statement support in Arm Fortran Compiler
- Intrinsic support in Arm Fortran Compiler
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

4.4 armflang for pgfortran users

The reference versions used in this guide are:

- PGI Fortran Compiler 18.5
- Arm® Fortran Compiler

Invoking the compiler

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

Table 4-14: Invoking the compiler

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

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

Table 4-15: Accessing version information and documentation

	PGI	Arm
Version details	`pgfortran --version`	`armflang --version`

Help and documentation	`pgfortran --help`
	
	`man pgFortran`
		`armflang --help`
		
		`man armflang`

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 4-16: 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 the Compiler options for the Arm Fortran Compiler.

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 remains supported in the armflang, but from versions 22.0.2+ 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.

`-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 64-bit (such as real (KIND=8)). Unspecified real kinds are evaluated as KIND=8.
`-i8`	`-i8`	Set the default kind for INTEGER and LOGICAL to 64-bit (KIND=8).

PGI	Arm	Description
`-fpic`	`-fpic`	Generate Position Independent Code (PIC).
`-fPIC`	`-fPIC`	Arm C/C++/Fortran Compiler and PGI both treat `-fpic` as equivalent to `-fPIC`. To increase code portability, Arm recommends using `-fPIC` when compiling with Arm C/C++/Fortran Compiler. For more information on the use of `-fpic` and `-fPIC` on AArch64, see the Note about building Position Independent Code PIC on AArch64.

Optimization compiler options

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

Table 4-17: 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.

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	`-ffinite-maths-only`	Enable optimizations that ignore the possibility of NaNs and Infs.

Further resources

Standards compliance

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

Additional information

- Arm Fortran Compiler Developer and Reference Guide
- Statement support in Arm Fortran Compiler
- Intrinsic support in Arm Fortran Compiler
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

5 Coding for Neon

The topics in this chapter describe coding for Neon®.

5.1 Introducing Neon for Armv8-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 extra 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 increase the speed of applications, for example audio and video processing, voice and facial recognition, computer vision, and deep learning.

As an application developer, there are several ways you can use Neon technology:

- Neon-enabled libraries, for example 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.

If you are 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 Armv8.

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 useful. For example, see the Arm Cortex-A75 Technical Reference Manual and the Arm Cortex-A75 Software Optimization Guide.

Data processing methodologies

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

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, for example 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 working 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 must be loaded into a separate 64-bit register. Performing large numbers of individual operations on small data sizes does not use hardware resources efficiently because processor, registers, and data paths are all designed for 64-bit calculations. In addition, when you add two 8-bit values and get a 9-bit result, you must add extra instructions to cope with the overflow.

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 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:

Figure 5-1: Neon SIMD add example
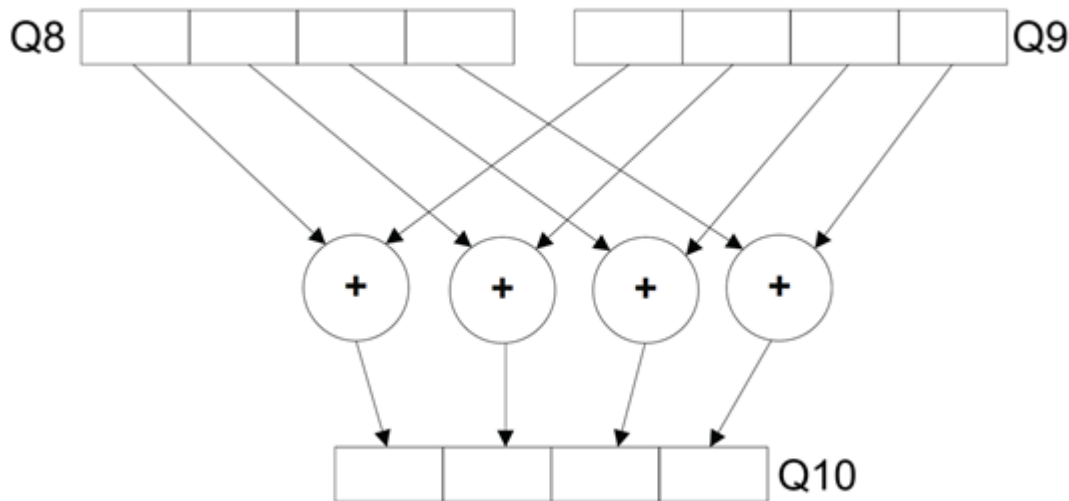
Performing the four operations with a single SIMD instruction is faster than with four separate SISD instructions. The diagram shows 128-bit registers each holding four 32-bit values, but other combinations are possible for Neon registers:

- 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, for example those used in mobile devices, often divide each full data register into multiple sub-registers and perform computations on the sub-registers in parallel. If the processing for the data sets is simple and repeated many times, SIMD can give considerable performance improvements for these processors. SIMDS 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.

Fundamentals of Armv8 Neon technology

Armv8 includes both 32-bit execution and 64-bit execution states, each with their own instruction sets:

- AArch64 is the name that is used to describe the 64-bit Execution state of the Armv8 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 Armv8 architecture, which is almost identical to Armv7.

> **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 Armv7, including Neon instructions.

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 reduces the area and power cost that is 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 that is 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`.

Figure 5-2: Neon lanes and elements
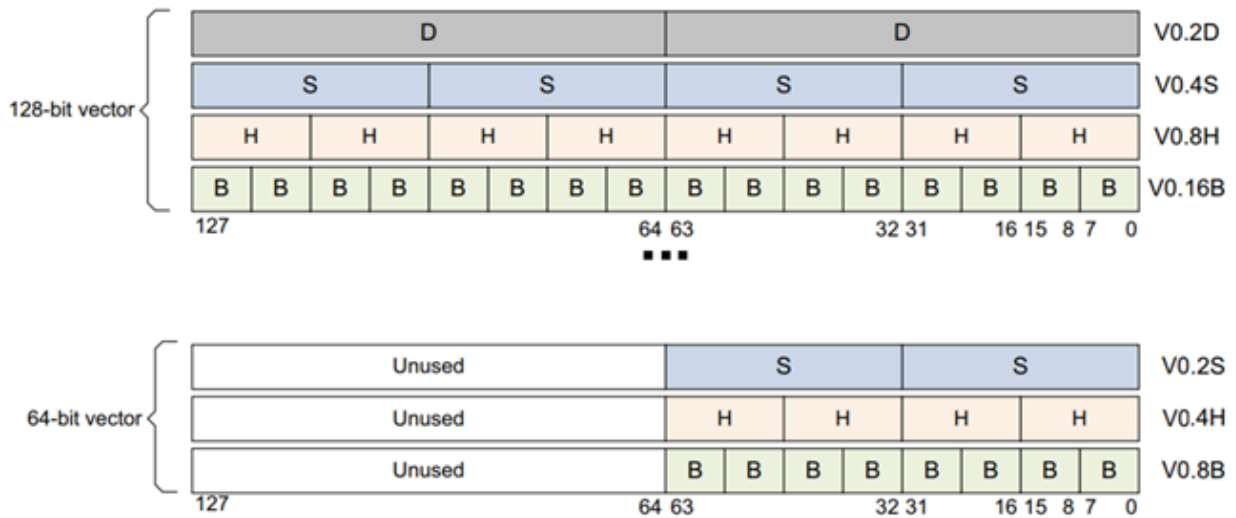
Elements in a vector are ordered from the least significant bit. In other words, element `0` uses the least significant bits of the register. Looking at an example of a Neon instruction, the instruction `ADD V0.8H, V1.8H, V2.8H` performs a parallel addition of eight lanes of 16-bit (8 x 16 = 128) integer elements from vectors in `V1` and `V2`, storing the result in `V0`:

Figure 5-3: Neon 8-bit by 16-bit add example
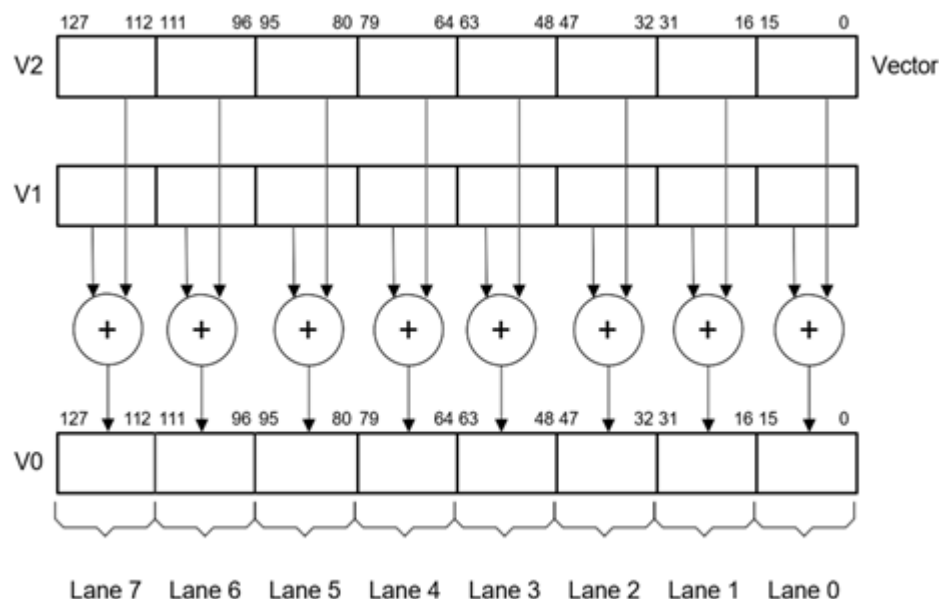
The Optimizing C code with Neon intrinsics topic provides a useful introduction to Neon programming. The tutorial describes how to use Neon intrinsics by examining an example which processes a matrix multiplication.

Related information

Optimizing C code with Neon intrinsics on page 89
Arm Architecture Reference Manual for the Armv8-A architecture profile
ISA exploration tools
Neon Intrinsics Reference
Arm Community

5.2 Compile for Neon with auto-vectorization

This guide shows how to use the auto-vectorization features in Arm® Compiler for Linux to automatically generate code that contains Armv8 Advanced SIMD instructions. It contains a number of examples to explore Neon® code generation and highlights coding best practices that help the compiler produce the best results.

As a programmer, there are a number of ways you can make use of Neon technology:

- Neon-enabled math libraries such as the Arm Performance Libraries 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. This gives you direct, low-level access to the exact Neon instructions you want, all from C/C++ code.

- For very high performance, hand-coded Neon assembler can be an alternative approach for experienced programmers.

> **Note**
>
> The examples in this guide use Arm Compiler for Linux which is a Linux user space, C/C++, and Fortran compiler, tuned for scientific computing, HPC, and enterprise workloads.
>
> You can adapt the examples for other compilers, however, you will need to consult your compiler documentation to find out the equivalent options to use in the examples.

Why rely on the compiler for auto-vectorization?

Writing hand-optimized assembly kernels or C code containing Neon intrinsics provides a high level of control over the Neon code in your software. However, these methods might cause portability and engineering complexity.

In many cases, a high quality compiler can generate code which is just as good, but requires significantly less design time. The process of allowing the compiler to automatically identify opportunities in your code to use Advanced SIMD instructions is called auto-vectorization.

In terms of specific compilation techniques, auto-vectorization includes:

- Loop vectorization: unrolling loops to reduce the number of iterations, while performing more operations in each iteration.

- Superword-Level Parallelism (SLP) vectorization: bundling scalar operations together to make use of full width Advanced SIMD instructions.

The benefits of relying on compiler auto-vectorization are:

- Programs implemented in high level languages are portable, so long as there are no architecture-specific code elements such as inline assembly or intrinsics.

- Modern compilers are capable of performing advanced optimizations automatically.

- Targeting a given micro-architecture can be as easy as setting a single compiler option. Optimizing an assembly program requires deep knowledge of the target hardware.

Auto-vectorization might not be the right choice in all situations, however:

- While source code can be architecture agnostic, to get the best code-generation source code might have to be compiler-specific.

- Small changes in a high-level language or the compiler options can result in significant and unpredictable changes in generated code.

- Using the compiler to generate Neon code is appropriate for most projects. Other methods for exploiting Neon are only necessary when the generated code does not deliver the necessary performance, or when particular hardware features are not supported by high-level languages.

For example, configuring system registers to control floating-point functionality must be performed in assembly code.

Compile for Neon with Arm Compiler for Linux

To enable automatic vectorization you must specify an optimization level that includes auto-vectorization.

Arm Compiler for Linux supports some useful tools to analyze what the compiler has vectorized:

- C/C++ Arm Optimization Remarks or Fortran Arm Optimization Remarks: Specifying the `-Rpass=loop` compiler option displays useful diagnostic information from the compiler about how it optimized particular loops. This information includes vectorization width and interleave count. Specifying the `-Rpass-analysis=loop` compiler option provides more information about what the compiler has analyzed.

- C/C++ Arm Optimization Report or Fortran Arm Optimization Report: A new feature of Arm Compiler for Linux that builds upon the `llvm-opt-report` tool available in open-source LLVM. Arm Optimzation Report allows you to see what the compiler has unrolled, vectorized, and interleaved as annotated source code in the terminal. To use Arm Optimization Report, add the option `-fsave-optimization-record` to your compile line. A `<filename>.opt.yaml` file is created. Read the resulting yaml file using:

```
arm-opt-report <filename>.opt.yaml
```

Specify a Neon target

Neon is supported in all standard Armv8-A implementations. Targeting any Armv8-A architecture or processor allows the generation of Neon code.

To run on a wide range of processors, you can target an architecture. Generated code runs on any processor implementation of that target architecture, but performance might be impacted.

To compile for any Armv8-A AArch64 target, pass the `-march=armv8-a` option at compile time:

```
armclang -march=armv8-a
```

To run on specific microarchitectures or processor, you can target the microarchitecture or processor. Generated code runs only on processors supporting that microarchitectural implementation, or the specified processor.

To run code on a specific microarchitecture, pass the `-mtune=<microarch>` option at compile time:

```
armclang -mtune=<microarch>
```

To run code on a particular processor, pass the `-mcpu=<cpu>` option at compile time:

```
armclang -mcpu=native
```

On Arm, the `-mcpu` option is a combination of `-march` and `-mtune`. It simultaneously specifies the target architecture and optimizes for a given microarchitecture.

The simplest and easiest way to get the best performance on Arm with Arm Compiler for Linux, GNU compilers, or LLVM-compilers is to only use the `-mcpu=native` option, and avoid using `-mtune` or `-march`. For more information about architectural compiler flags and data to support this recommendation, see the Compiler flags across architectures: -march, -mtune, and -mcpu blog.

Specify an auto-vectorizing optimization level

Arm Compiler for Linux provides a wide range of optimization levels, selected with the `-o` option:

Table 5-1: Some Arm C/C++ Compiler optimization options

Option	Description	Auto-vectorization
`-O0`	Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this might result in a significantly larger image. This is the default optimization level.	Never
`-O1`	Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug.	Disabled by default.
`-O2`	High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information.	Enabled by default.
`-O3`	Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.	Enabled by default.
`-Ofast`	Enable all the optimizations from level 3, including those performed with the -ffp-mode=fast armclang option. This level also performs other aggressive optimizations that might violate strict compliance with language standards.	Enabled by default.

For more information about these options, see the Arm C/C++ Compiler compiler options or Arm Fortran Compiler compiler options.

Auto-vectorization is enabled by default at optimization level `-O2` and higher. The `-fno-vectorize` option lets you disable auto-vectorization.

At optimization level `-O1`, auto-vectorization is disabled by default. The `-fvectorize` option lets you enable auto-vectorization.

At optimization level `-O0`, auto-vectorization is always disabled. If you specify the `-fvectorize` option, the compiler ignores it.

Example: Vector addition

This example shows how you can use compiler options to auto-vectorize and optimize a simple C program.

1. Create a new file `vec_add.c` containing the following `vec_add(float *vec_A, float *vec_B, float *vec_C, int len_vec)` function which adds two arrays of 32-bit floating-point values.

```c
/*
 * Copyright (C) Arm Limited, 2022 All rights reserved.
 *
 * The example code is provided to you as an aid to learning when working
 * with Arm-based technology, including but not limited to programming tutorials.
 * Arm hereby grants to you, subject to the terms and conditions of this Licence,
 * a non-exclusive, non-transferable, non-sub-licensable, free-of-charge licence,
 * to use and copy the Software solely for the purpose of demonstration and
 * evaluation.
 *
 * You accept that the Software has not been tested by Arm therefore the Software
 * is provided "as is", without warranty of any kind, express or implied. In no
 * event shall the authors or copyright holders be liable for any claim, damages
 * or other liability, whether in action or contract, tort or otherwise, arising
 * from, out of or in connection with the Software or the use of Software.
 */
#include <stdio.h>
void vec_init(float *vec, int len_vec, float init_val) {
        int i;
        for (i=0; i<len_vec; i++) {
                vec[i] = init_val;
        }
}
void vec_print(float *vec, int len_vec) {
        int i;
        for (i=0; i<len_vec; i++) {
                printf("%f, ", vec[i]);
        }
        printf("\n");
}
void vec_add(float *vec_A, float *vec_B, float *vec_C, int len_vec) {
        int i;
        for (i=0; i<len_vec; i++) {
                vec_C[i] = vec_A[i] + vec_B[i];
        }
}
int main() {
        int N = 10;
        float A[N];
        float B[N];
        float C[N];
        vec_init(A, N, 1.0);
        vec_init(B, N, 1.0);
        vec_add(A, B, C, N);
        vec_print(A, N);
        vec_print(B, N);
        vec_print(C, N);
        return 0;
}
```

2. Compile the code without using auto-vectorization:

```
armclang -march=armv8-a -g -c -O1 vec_add.c
```

3. To see the generated instructions, disassemble the resulting object file:

```
fromelf --disassemble vec_add.o -o disassembly_vec_off.txt
```

The disassembled code looks similar to:

```
vec_add                         ; Alternate entry point
        CMP       w3,#1
        B.LT      |L3.36|
        MOV       w8,w3
|L3.12|
        LDR       s0,[x0],#4
        LDR       s1,[x1],#4
        SUBS      x8,x8,#1
        FADD      s0,s0,s1
        STR       s0,[x2],#4
        B.NE      |L3.12|
|L3.36|
        RET
```

You can see the label name `vec_add` for the function, followed by the generated assembly instructions that make up the function. The `FADD` instruction performs the core part of the operation, but the code is not making use of Neon as only one addition operation is performed at a time because the `FADD` instruction is operating on the scalar registers `s0` and `s1`.

4. Re-compile the code using auto-vectorization:

```
armclang -march=armv8-a -g -c -O1 vec_add.c -fvectorize
```

5. To see the generated instructions, disassemble the resulting object file:

```
fromelf --disassemble vec_add.o -o disassembly_vec_on.txt
```

The disassembled code looks similar to:

```
vec_add                         ; Alternate entry point
        CMP       w3,#1
        B.LT      |L3.184|
        CMP       w3,#4
        MOV       w8,w3
        MOV       x9,xzr
        B.CC      |L3.140|
        LSL       x10,x8,#2
        ADD       x12,x0,x10
        ADD       x11,x2,x10
        CMP       x12,x2
        ADD       x10,x1,x10
        CSET      w12,HI
        CMP       x11,x0
        CSET      w13,HI
        CMP       x10,x2
        CSET      w10,HI
        CMP       x11,x1
        AND       w12,w12,w13
        CSET      w11,HI
        TBNZ      w12,#0,|L3.140|
        AND       w10,w10,w11
        TBNZ      w10,#0,|L3.140|
        AND       x9,x8,#0xfffffffc
```

```
            MOV       x10,x9
            MOV       x11,x2
            MOV       x12,x1
            MOV       x13,x0
|L3.108|
            LDR       q0,[x13],#0x10
            LDR       q1,[x12],#0x10
            SUBS      x10,x10,#4
            FADD      v0.4S,v0.4S,v1.4S
            STR       q0,[x11],#0x10
            B.NE      |L3.108|
            CMP       x9,x8
            B.EQ      |L3.184|
|L3.140|
            LSL       x12,x9,#2
            ADD       x10,x2,x12
            ADD       x11,x1,x12
            ADD       x12,x0,x12
            SUB       x8,x8,x9
|L3.160|
            LDR       s0,[x12],#4
            LDR       s1,[x11],#4
            SUBS      x8,x8,#1
            FADD      s0,s0,s1
            STR       s0,[x10],#4
            B.NE      |L3.160|
|L3.184|
            RET
```

SLP auto-vectorization has been successful, as you can see from the instruction `FADD v0.4s,v0.4s,v1.4s`, which performs an addition on four 32-bit floats packed into a SIMD register. However, the auto-vectorization has come at significant cost to code size because it must detect cases where the SIMD width is not a divisor of the array length. This increase in code size might or might not be acceptable depending on the project and target hardware.

Example: function in a loop

If you want to use particular optimization features of the compiler, sometimes changes to source code are unavoidable. These unavoidable changes can occur when the code is too complex for the compiler to auto-vectorize, or when you want to override the decision the compiler makes about how to optimize a particular piece of code.

1. Create a new file called `cubed.c` containing the `cubed` and `vec_cubed` functions, which calculate the cubes of an array of values.

```
/*
 * Copyright (C) Arm Limited, 2022 All rights reserved.
 *
 * The example code is provided to you as an aid to learning when working
 * with Arm-based technology, including but not limited to programming tutorials.
 * Arm hereby grants to you, subject to the terms and conditions of this Licence,
 * a non-exclusive, non-transferable, non-sub-licensable, free-of-charge licence,
 * to use and copy the Software solely for the purpose of demonstration and
 * evaluation.
 *
 * You accept that the Software has not been tested by Arm therefore the Software
 * is provided "as is", without warranty of any kind, express or implied. In no
 * event shall the authors or copyright holders be liable for any claim, damages
 * or other liability, whether in action or contract, tort or otherwise, arising
 * from, out of or in connection with the Software or the use of Software.
 */
#include <stdio.h>
void vec_init(double *vec, int len_vec, double init_val) {
        int i;
```

```
            for (i=0; i<len_vec; i++) {
                    vec[i] = init_val*i - len_vec/2;
            }
}
void vec_print(double *vec, int len_vec) {
        int i;
        for (i=0; i<len_vec; i++) {
                printf("%f, ", vec[i]);
        }
        printf("\n");
}
double cubed(double x) {
        return x*x*x;
}
void vec_cubed(double *x_vec, double *y_vec, int len_vec) {
        int i;
        for (i=0; i<len_vec; i++) {
                y_vec[i] = cubed(x_vec[i]);
        }
}
__attribute__((always_inline)) double cubed_i(double x) {
        return x*x*x;
}
void vec_cubed_opt(double *restrict x_vec, double *restrict y_vec, int len_vec) {
        int i;
        #pragma clang loop interleave_count(1)
        for (i=0; i<len_vec; i++) {
                y_vec[i] = cubed_i(x_vec[i]);
        }
}
int main() {
        int N = 10;
        double X[N];
        double Y[N];
        vec_init(X, N, 1);
        vec_print(X, N);
        vec_cubed(X, Y, 10);
        vec_print(Y, N);
        vec_cubed_opt(X, Y, 10);
        vec_print(Y, N);
        return 0;
}
```

2. Compile the code using auto-vectorization:

```
armclang -march=armv8-a -g -c -O1 -fvectorize cubed.c
```

3. To see the generated instructions, disassemble the resulting object file:

```
fromelf --disassemble cubed.o -o disassembly.txt
```

The disassembled code looks similar to this:

```
cubed                       ; Alternate entry point
        FMUL     d1,d0,d0
        FMUL     d0,d1,d0
        RET
        AREA ||.text.vec_cubed||, CODE, READONLY, ALIGN=2
vec_cubed                        ; Alternate entry point
        STP      x21,x20,[sp,#-0x20]!
        STP      x19,x30,[sp,#0x10]
        CMP      w2,#1
        B.LT     |L4.48|
        MOV      x19,x1
        MOV      x20,x0
```

```
        MOV        w21,w2
|L4.28|
        LDR        d0,[x20],#8
        BL         cubed
        SUBS       x21,x21,#1
        STR        d0,[x19],#8
        B.NE       |L4.28|
|L4.48|
        LDP        x19,x30,[sp,#0x10]
        LDP        x21,x20,[sp],#0x20
        RET
```

There are a number of issues in this code:

- The compiler has not performed loop or SLP vectorization, or inlined our cubed function.

- The code needs to perform checks on the input pointers to verify that the arrays do not overlap.

One solution to address these issues is to compile at a higher optimization level. However, the next steps show how to optimize the code without altering the compiler options.

4. To override some of the decisions of the compiler, add the following macros and qualifiers to the code:

- `__attribute__((always_inline))` is an Arm Compiler for Linux extension which indicates that the compiler always attempts to inline the function. In this example, not only is the function inlined, but the compiler can also perform SLP vectorization.

  Before inlining, the `cubed` function works with scalar doubles only, so there is no need or way of performing SLP vectorization on this function by itself.

  When the `cubed` function is inlined, the compiler can detect that its operations are performed on arrays and vectorize the code with the available ASIMD instructions.

- `restrict` is a standard C/C++ keyword that indicates to the compiler that a given array corresponds to a unique region of memory. `restrict` eliminates the need for run-time checks for overlapping arrays.

- `#pragma clang loop interleave_count(X)` is a Clang language extension that lets you control auto-vectorization by specifying a vector width and interleaving count. This pragma is a [COMMUNITY] feature of Arm Compiler.

```
__always_inline double cubed(double x) {
        return x*x*x;
}
void vec_cubed(double *restrict x_vec, double *restrict y_vec, int len_vec) {
        int i;
        #pragma clang loop interleave_count(2)
        for (i=0; i<len_vec; i++) {
                y_vec[i] = cubed(x_vec[i]);
        }
}
```

5. Compile and disassemble with the same commands used before.

   The resulting code is:

```
vec_cubed                        ; Alternate entry point
```

```
        CMP       w2,#1
        B.LT      |L4.132|
        CMP       w2,#4
        MOV       w8,w2
        B.CS      |L4.28|
        MOV       x9,xzr
        B         |L4.92|
|L4.28|
        AND       x9,x8,#0xfffffffc
        ADD       x10,x0,#0x10
        ADD       x11,x1,#0x10
        MOV       x12,x9
|L4.44|
        LDP       q0,q1,[x10,#-0x10]
        ADD       x10,x10,#0x20
        SUBS      x12,x12,#4
        FMUL      v2.2D,v0.2D,v0.2D
        FMUL      v3.2D,v1.2D,v1.2D
        FMUL      v0.2D,v0.2D,v2.2D
        FMUL      v1.2D,v1.2D,v3.2D
        STP       q0,q1,[x11,#-0x10]
        ADD       x11,x11,#0x20
        B.NE      |L4.44|
        CMP       x9,x8
        B.EQ      |L4.132|
|L4.92|
        LSL       x11,x9,#3
        ADD       x10,x1,x11
        ADD       x11,x0,x11
        SUB       x8,x8,x9
|L4.108|
        LDR       d0,[x11],#8
        SUBS      x8,x8,#1
        FMUL      d1,d0,d0
        FMUL      d0,d0,d1
        STR       d0,[x10],#8
        B.NE      |L4.108|
|L4.132|
        RET
```

This disassembly shows that the inlining, SLP vectorization, and loop vectorization have been successful. Using the restrict pointers has eliminated run-time overlap checks.

The code size has increased slightly because of the loop tail which handles any remaining iterations when the total loop count is not a multiple of four (the effective unroll depth). The loop unroll depth is two and is the SLP width is two, so the effective unroll depth is four. The next step shows an optimization you can make if you know the loop count is always a multiple of four.

6. Assuming the loop count is always a multiple of four, tell the compiler to mask off the lower bits of the loop counter:

```
void vec_cubed(double *restrict x_vec, double *restrict y_vec, int len_vec) {
        int i;
        #pragma clang loop interleave_count(1)
        for (i=0; i<(len_vec & ~3); i++) {
                y_vec[i] = cubed_i(x_vec[i]);
        }
}
Compile and disassemble with the same commands we used earlier. This produces the
 following code:
vec_cubed                          ; Alternate entry point
        AND       w8,w2,#0xfffffffc
        CMP       w8,#1
        B.LT      |L13.40|
```

```
        MOV       w8,w8
|L13.16|
        LDR       q0,[x0],#0x10
        SUBS      x8,x8,#2
        FMUL      v1.2D,v0.2D,v0.2D
        FMUL      v0.2D,v0.2D,v1.2D
        STR       q0,[x1],#0x10
        B.NE      |L13.16|
|L13.40|
        RET
```

The code size is reduced because the compiler knows it no longer has to test for and deal with any remaining iterations that were not a multiple of four. Informing the compiler that the data supplied will always be a multiple of the vector length has produced more optimized code.

This example is simple enough that compiling at -o2 will perform all of these optimizations with no code changes, but more complex pieces of code might require this type of tuning to get the most from the compiler.

Coding best practices for auto-vectorization

As an implementation becomes more complicated the likelihood that the compiler can auto-vectorize the code decreases. For example, loops with the following characteristics are particularly difficult (or impossible) to vectorize:

- Loops with interdependencies between different loop iterations.

- Loops with break clauses.

- Loops with complex conditions.

Arm recommends modifying your source code implementation to eliminate these situations.

For example, a necessary condition for auto-vectorization is that the number of iterations in the loop size must be known at the start of the loop. Break conditions mean the loop size might not be knowable at the start of the loop, which prevents auto-vectorization. If it is not possible to completely avoid a break condition, it might be worthwhile breaking up the loops into multiple vectorizable and non-vectorizable parts.

A full discussion of the compiler directives used to control vectorization of loops for can be found in the LLVM-Clang documentation, but the two most important are:

- #pragma clang loop vectorize(enable)

- #pragma clang loop interleave(enable)

These pragmas are hints to the compiler to perform SLP and Loop vectorization respectively.

More detailed guides covering auto-vectorization are available in the Arm C/C++ Compiler and Arm Fortran Compiler Reference guides:

- Coding best practice for auto-vectorization

- Using pragmas to control auto-vectorization

Related information

Arm C/C++ Compiler documentation
GCC documentation
Architecture Exploration Tools
Arm Architecture Reference Manual Armv8

5.3 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 explains 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.

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 several ways you can use 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.

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, avoiding many of the difficulties often seen when developing assembly code.

For 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 several 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 rewritten 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:

- More learning is required to use Neon intrinsics, than to import a library or to rely on a compiler.

- Hand-optimized assembly code might offer the greatest scope for performance improvement even if it is more difficult to write.

Example: Matrix multiplication

This example implements some C functions using Neon intrinsics. The example chosen does not demonstrate the full complexity of the 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 and consists of groups of arithmetic operations which are repeated in a simple way:

Figure 5-4: Matrix multiplication diagram
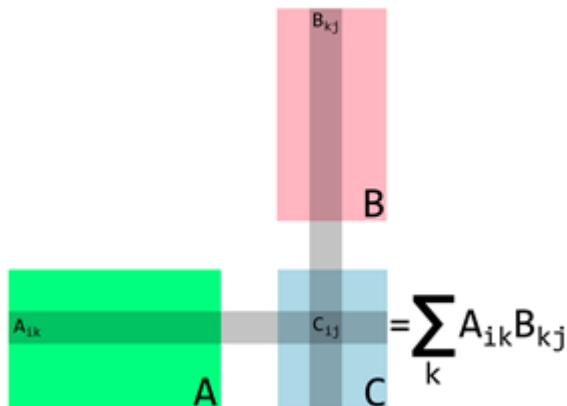
The matrix multiplication process is as follows:

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];
            }
        }
    }
}
```

The preceding code uses a column-major layout of the matrices in memory. That is, an $n \times m$ 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.

This example examines small, fixed-size matrices before moving on to larger matrices.

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

```
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;
        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, that is 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?	Why are they used?
float32x4_t	An array of four 32-bit floats.	One uint32x4_t fits into a 128-bit register and ensures that 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 needed from A and B.

Code element	What is it?	Why are they used?
`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`, adds the result to a third `float32x4_t`, and then returns the result.	Since the matrix row-on-column dot products are a set of multiplications and additions, this operation fits 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 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 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 result 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){
                    // 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, that is 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,Cl);
    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 it utilizes the Advanced SIMD technology with full register packing. Typical C code, generally, does not.

- `FMLA` instead of `FMUL` instructions. As specified by the intrinsics.

- Fewer loop iterations. When used properly, intrinsics allow loops to be unrolled easily.

In the disassembled function there are unnecessary loads and stores because memory allocation and data type initialization (for example, `float32x4_t`) was not addressed in the C code.

Full source code of Matrix multiplication example

```
/*
 * Copyright (C) Arm Limited, 2019 All rights reserved.
 *
 * The example code is provided to you as an aid to learning when working
 * with Arm-based technology, including but not limited to programming tutorials.
 * Arm hereby grants to you, subject to the terms and conditions of this Licence,
 * a non-exclusive, non-transferable, non-sub-licensable, free-of-charge licence,
 * to use and copy the Software solely for the purpose of demonstration and
 * evaluation.
 *
 * You accept that the Software has not been tested by Arm therefore the Software
 * is provided "as is", without warranty of any kind, express or implied. In no
 * event shall the authors or copyright holders be liable for any claim, damages
 * or other liability, whether in action or contract, tort or otherwise, arising
 * from, out of or in connection with the Software or the use of Software.
 */
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
```

```c
#include <arm_neon.h>
#define BLOCK_SIZE 4
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];
      }
    }
  }
}
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 result in C.
  * It is the user's 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) {
      // 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 in 4x1 blocks, i.e. each column in 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, B2, 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;
      vst1q_f32(C+C_idx, C0);
      vst1q_f32(C+C_idx+n, C1);
      vst1q_f32(C+C_idx+2*n, C2);
      vst1q_f32(C+C_idx+3*n, C3);
    }
  }
}
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;
  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);
}
void print_matrix(float32_t *M, uint32_t cols, uint32_t rows) {
  for (int i=0; i<rows; i++) {
    for (int j=0; j<cols; j++) {
      printf("%f ", M[j*rows + i]);
    }
    printf("\n");
  }
```

```c
    printf("\n");
}
void matrix_init_rand(float32_t *M, uint32_t numvals) {
    for (int i=0; i<numvals; i++) {
      M[i] = (float)rand()/(float)(RAND_MAX);
    }
}
void matrix_init(float32_t *M, uint32_t cols, uint32_t rows, float32_t val) {
    for (int i=0; i<rows; i++) {
      for (int j=0; j<cols; j++) {
        M[j*rows + i] = val;
      }
    }
}
bool f32comp_noteq(float32_t a, float32_t b) {
    if (fabs(a-b) < 0.000001) {
      return false;
    }
    return true;
}
bool matrix_comp(float32_t *A, float32_t *B, uint32_t rows, uint32_t cols) {
    float32_t a;
    float32_t b;
    for (int i=0; i<rows; i++) {
      for (int j=0; j<cols; j++) {
        a = A[rows*j + i];
        b = B[rows*j + i];
        if (f32comp_noteq(a, b)) {
          printf("i=%d, j=%d, A=%f, B=%f\n", i, j, a, b);
          return false;
        }
      }
    }
    return true;
}
int main() {
  uint32_t n = 2*BLOCK_SIZE; // rows in A
  uint32_t m = 2*BLOCK_SIZE; // cols in B
  uint32_t k = 2*BLOCK_SIZE; // cols in a and rows in b
  float32_t A[n*k];
  float32_t B[k*m];
  float32_t C[n*m];
  float32_t D[n*m];
  float32_t E[n*m];
  bool c_eq_asm;
  bool c_eq_neon;
  matrix_init_rand(A, n*k);
  matrix_init_rand(B, k*m);
  matrix_init(C, n, m, 0);
  print_matrix(A, k, n);
  print_matrix(B, m, k);
  //print_matrix(C, n, m);
  matrix_multiply_c(A, B, E, n, m, k);
  printf("C\n");
  print_matrix(E, n, m);
  printf("===============================\n");
  matrix_multiply_neon(A, B, D, n, m, k);
  printf("Neon\n");
  print_matrix(D, n, m);
  c_eq_neon = matrix_comp(E, D, n, m);
  printf("Neon equal to C? %d\n", c_eq_neon);
  printf("===============================\n");
}
```

Program conventions

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.

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 values where the vector data types are 64 bits or 128 bits long, and therefore fit completely into a Neon register. `N` corresponds with those instructions which operate on multiple registers at once.

Functions

Similar to 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` might be present).

q

> Indicates a saturating operation (except for `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 that is 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.

Quick reference

What is Neon?

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

All processors that are compliant with the Armv8-A architecture (for example, the Cortex-A76 or Cortex-A57) include Neon. Neon provides an extra 32 128-bit registers with instructions that operate on 8, 16, 32, or 64 bit lanes in 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.

Related information

Arm C Language Extensions (ACLE)

Neon Intrinsics Reference

Architecture Exploration Tools

Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile

5.4 Useful Neon Resources

Some useful resources when coding for Neon®, are:

Developer resources

- Arm Neon web page

- Neon Programmer's Guide for Armv8-A

- Neon Programmer's Guide for Armv8-A: Coding for Neon

- Neon Intrinsics

Blogs

- Arm Neon optimization blog

- Arm Neon Programming Quick Reference

- Coding for Neon: Matrix Multiplication

- Coding for Neon: Rearranging Vectors

6 Porting to Arm resources

This guide supplements the other resources which Arm provides to help you start porting your applications to Arm, with a focus on optimizing for the Arm Advanced SIMD architectural extension (called Neon® in an Arm implementation). This chapter provides more information about additional porting resources available from Arm.

6.1 Porting resources

To help you port your applications to AArch64, Arm provides a variety of resources:

- Community-driven porting recipes on the Arm HPC Packages Wiki. The GitLab repository contains a broader list of porting and tuning recipes to which the community contribute and maintain.

- White papers. A collection of White Papers and externally-published papers that are relevant to High Performance Computing (HPC) on Arm.

- Conference presentations. Arm frequently presents and hosts events at HPC industry conferences. This page collects the presentations delivered at these events.

- Arm Compiler for Linux documentation. Arm® Compiler for Linux is composed of Arm Fortran Compiler, Arm C/C++ Compiler, and Arm Performance Libraries.

- Arm Forge documentation.

- HPC blogs and forums on Arm HPC Community. Arm frequently publishes information about new releases, interesting research, conference reviews, and industry news through blogs. The forum is available for the Arm HPC community to ask questions and find solutions in a forum context.

To learn more about porting your application to SVE-enabled Arm-based systems, see the SVE-variant of this porting guide: Porting and Optimizing HPC Applications for Arm SVE guide. This guide includes some introductory information about SVE, illustrates some example code, and discusses the tooling Arm provides to build code for SVE hardware.