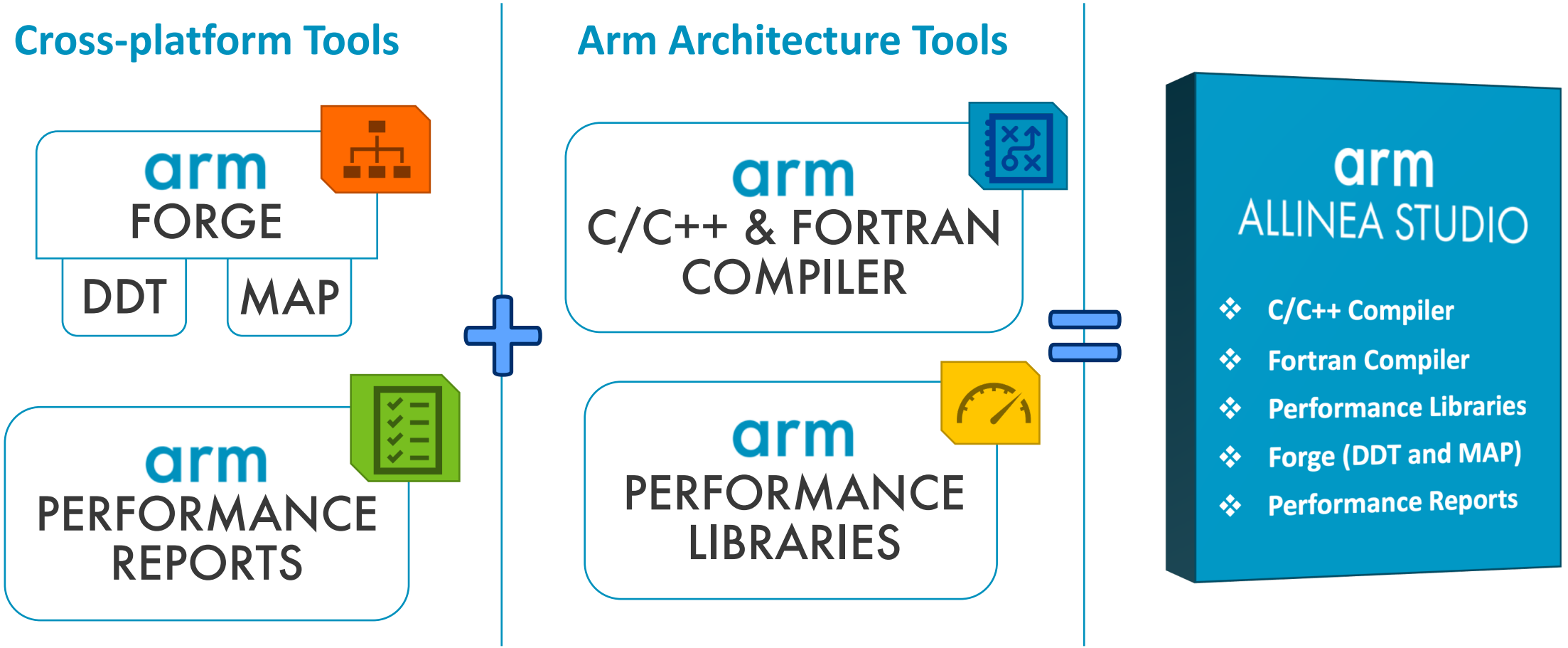
HPC Compilers and Libraries

George Steed

Senior Software Engineer, Arm Performance Libraries

Arm's solution for HPC application development and porting

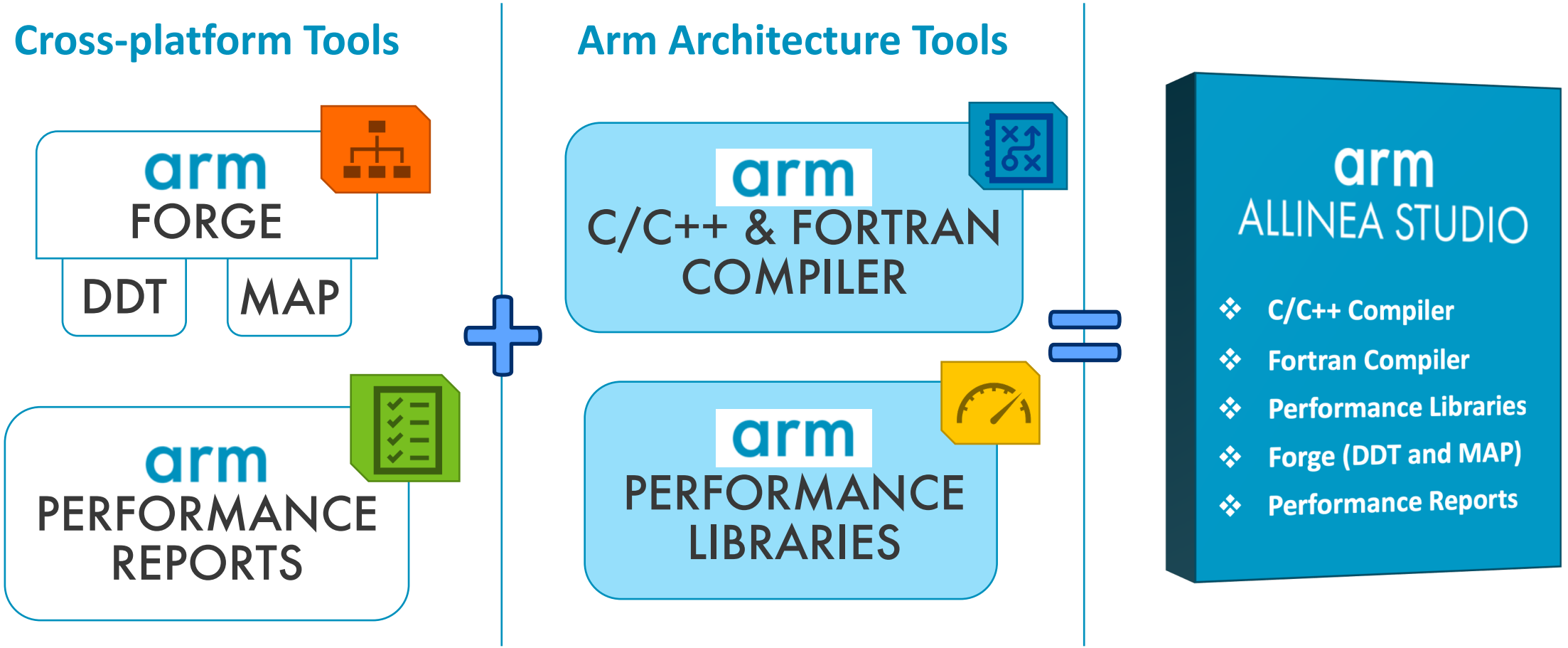
Commercial tools for aarch64, x86_64, ppc64 and accelerators



arm

Arm's solution for HPC application development and porting

Commercial tools for aarch64, x86_64, ppc64 and accelerators



arm

Arm HPC Compiler

arm

arm COMPILER

Arm's commercially-supported C/C++/Fortran compiler

Compilers tuned for Scientific Computing and HPC

Latest features and performance optimizations

Commercially supported by Arm

Tuned for Scientific Computing, HPC and Enterprise workloads

- Processor-specific optimizations for various server-class Arm-based platforms
- Optimal shared-memory parallelism using latest Arm-optimized OpenMP runtime

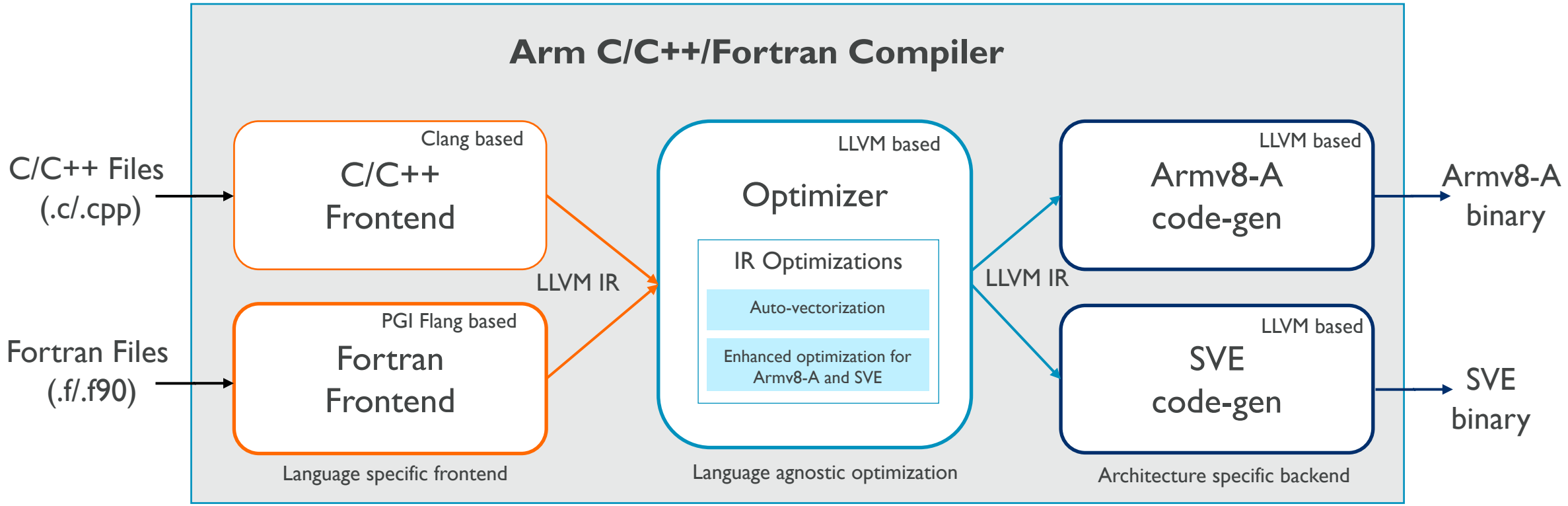
Linux user-space compiler with latest features

- C++ 14 and Fortran 2003 language support with OpenMP 4.5
- Support for Armv8-A and SVE architecture extension
- Based on LLVM and Flang, leading open-source compiler projects

Commercially supported by Arm

- Available for a wide range of Arm-based platforms running leading Linux distributions – RedHat, SUSE and Ubuntu

Arm Compiler is built on LLVM, Clang and Flang



arm

Arm Compiler for HPC: Front-end

Clang and Flang

C/C++

- Clang front-end
  - C11 including GNU11 extensions and C++17
  - Arm's 10-year roadmap for Clang is routinely reviewed and updated to respond to customers
- C11 with GNU11 extensions and C++17
- Auto-vectorization for NEON and SVE
- OpenMP 4.5

Fortran

- Flang front-end
  - Extended to support GNU Fortran (gfortran) flags
- Fortran 2003 with some 2008
- Auto-vectorization for NEON and SVE
- OpenMP 3.1
- Transition to F18
  - Extensible front-end written in C++17
  - Complete Fortran 2008 support
  - OpenMP 4.5 support
  - 4+2 engineers allocated + community

arm

Arm Compiler for HPC: Back-end

LLVM7

- Arm pulls all relevant cost models and optimizations into the downstream codebase.
  - Vendors have committed to upstreaming their cost models to LLVM.
- Auto-vectorization via LLVM **vectorizers:**
  - Use cost models to drive decisions about what code blocks can and/or should be vectorized.
  - As of October 2018, two different vectorizers from LLVM: Loop Vectorizer and SLP Vectorizer.
- Loop Vectorizer support for SVE:
  - Loops with unknown trip count
  - Runtime checks of pointers
  - Reductions
  - Inductions
  - “If” conversion
  - Pointer induction variables
  - Reverse iterators
  - Scatter / gather
  - Vectorization of mixed types
  - Global structures alias analysis

LLVM Loop Vectorizer

LLVM7 Vectorizer with SVE Optimizations

Loop Vectorizer support for SVE

- Loops with unknown trip count
- Runtime checks of pointers
- Reductions
- Inductions
- If conversion
- Pointer induction variables
- Reverse iterators
- Scatter / gather
- Vectorization of mixed types
- Global structures alias analysis

Optimizations for specific vector lengths

- Partial unrolling during vectorization
- 256-bit and 512-bit optimizations

LLVM Optimization Remarks

Let the compiler tell you how to improve vectorization

To enable optimization remarks, pass the following `–Rpass` options to `armclang`:

Flag	Description
-Rpass=<regexp>	What was optimized.
-Rpass-analysis=<regexp>	What was analyzed.
-Rpass-missed=<regexp>	What failed to optimize.

For each flag, replace <regexp> with an expression for the type of remarks you wish to view. Recommended <regexp> queries are:

- `–Rpass=\(loop–vectorize\|inline\)`
- `–Rpass–missed=\(loop–vectorize\|inline\)`
- `–Rpass–analysis=\(loop–vectorize\|inline\)`

where loop-vectorize will filter remarks regarding vectorized loops, and inline for remarks regarding inlining.

LLVM Optimization Remarks example

```
armclang -O3 -Rpass=.* -Rpass-analysis=.* example.c
```

```
example.c:8:18: remark: hoisting zext
[-Rpass=licm]
        for (int i=0;i<K; i++)
        ^
example.c:8:4: remark: vectorized loop (vectorization width: 4, interleaved count: 2)
[-Rpass=loop-vectorize]
        for (int i=0;i<K; i++)
        ^ example.c:7:1: remark: 28 instructions in function
```

```
armflang -O3 -Rpass=loop-vectorize example.F90 -gline-tables-only
```

```
example.F90:21: vectorized loop (vectorization width: 2, interleaved count: 1)
[-Rpass=loop-vectorize]
        END DO
```


arm

Arm Compiler for HPC: OpenMP Support

OpenMP 4.5 in C/C++ and OpenMP 3.1 in Fortran

Supported Features

- C/C++: OpenMP support from Clang
  - OpenMP 4.5
    - Target offloading to host is supported,
    - Target offloading to remote targets not supported.
- Fortran: OpenMP support from Flang
  - OpenMP 3.1 with some 4.0 and 4.5
    - 4.5: taskloop, if-clause, affinity query, lock hints
    - 4.0: taskgroup, affinity policies, cancellation, atomic capture/swap

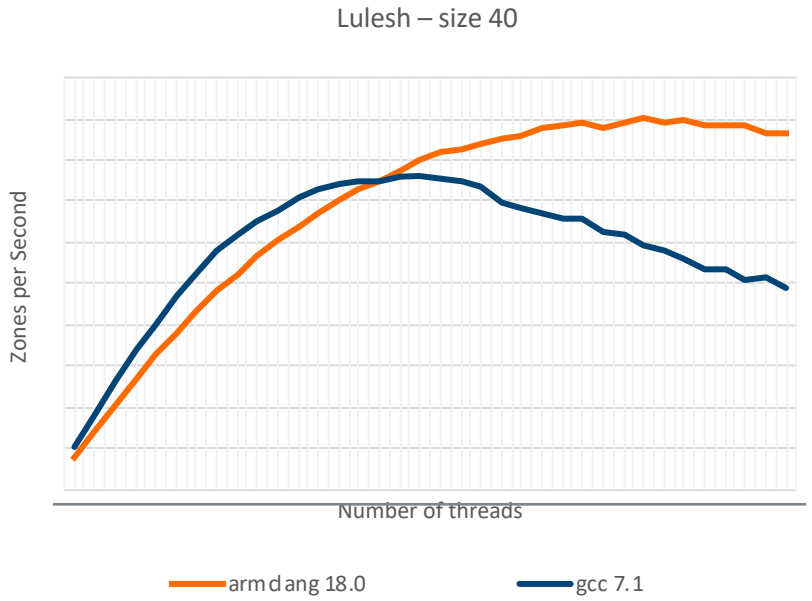
	C/C++	Fortran
OpenMP 4.0		
C/C++ Array Sections		n/a
Thread affinity policies		
"simd" construct		Partial [1]
"declare simd" construct		
Device constructs		
Task dependencies		
"taskgroup" construct		
User defined reductions		
Atomic capture swap		
Atomic seq_cst		
Cancellation		
OMP_DISPLAY_ENV		
OpenMP 4.5		
doacross loop nests with ordered		
"linear" clause on loop construct		
"simdlen" clause on simd construct		
Task priorities		
"taskloop" construct		
Extensions to device support		
"if" clause for combined constructs		
"hint" clause for critical construct		
"source" and "sink" dependence types		
C++ Reference types in data sharing attribute clauses		n/a
Reductions on C/C++ array sections		n/a
"ref", "val", "uval" modifiers for linear clause		
Thread affinity query functions		
Hints for lock API		

arm

Arm's Optimized OpenMP Runtime

Arm actively optimizes OpenMP runtime libraries for high thread counts

- Large System Extension (LSE) atomic update instructions
- Atomics dramatically reduce runtime overhead, especially at high thread counts.
  - Used extensively in the OpenMP runtime shipped with the Arm HPC Compiler.
  - Also available in GNU's runtime.
- Synchronization constructs optimized for high thread counts.
  - Designed with hundreds of threads in mind.
  - Uses hardware features whenever available.



arm

GCC is a major compiler in the Arm ecosystem

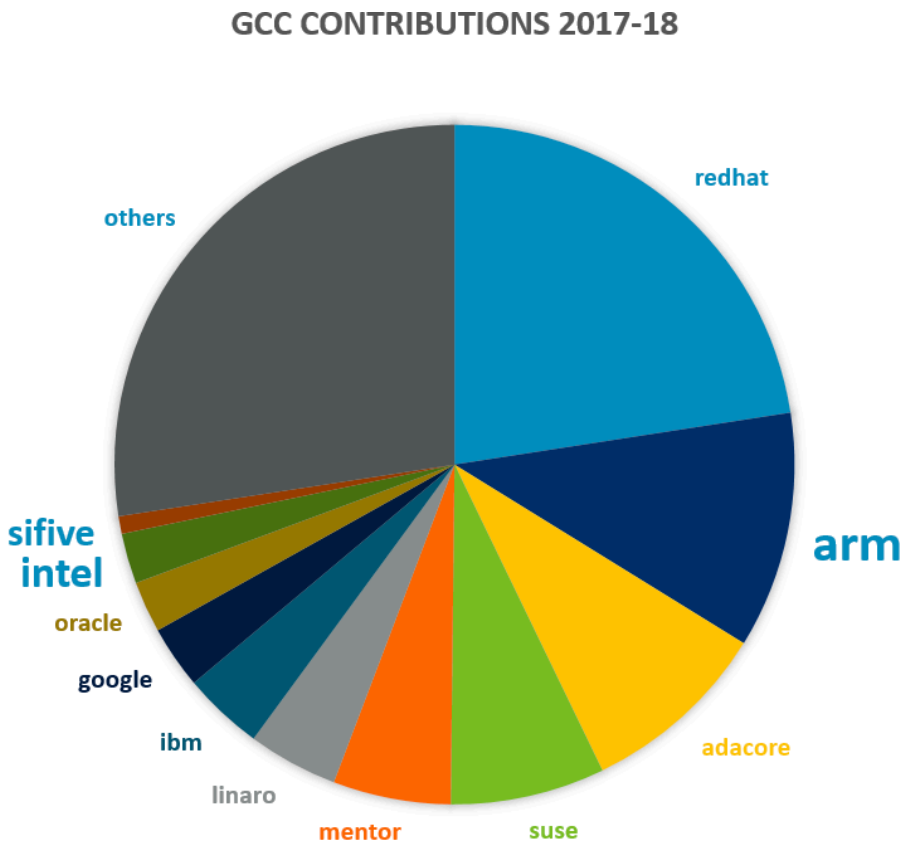
Arm the second largest contributor to the GCC project

On Arm, GCC is a first class compiler alongside commercial compilers.

- GCC ships with Arm Compiler for HPC

SVE support since GCC 8.0

NEON, LSE, and others well supported



arm

What's New in the Compiler

- Enhanced integration between ArmPL and Arm Compiler
  - -armpl flag simplifies compile/link line
- Improved performance of basic math functions through libamath
  - Linked in by default, no change to build process
- Improved performance of string.h functions (memcpy, memset) through libastring
  - Linked in by default, no change to build process
- Fortran
  - Submodules (F2008)
  - Directives: IVDEP, NOVECTOR, VECTOR ALWAYS
  - NINT and DNINT performance
- Improved documentation

Arm Performance Libraries

arm

arm PERFORMANCE LIBRARIES

Optimized BLAS, LAPACK and FFT

Commercially supported by Arm

Best in class performance

Validated with NAG test suite

Commercial 64-bit Armv8-A math libraries

- Commonly used low-level math routines - BLAS, LAPACK and FFT
- Provides FFTW compatible interface for FFT routines
- Batched BLAS support

Best-in-class serial and parallel performance

- Generic Armv8-A optimizations by Arm
- Tuning for specific platforms like Cavium ThunderX2 in collaboration with silicon vendors

Validated and supported by Arm

- Available for a wide range of server-class Arm-based platforms
- Validated with NAG's test suite, a de-facto standard
- Available for Arm compiler or GNU

What does all this mean in practice?

Commercial 64-bit Armv8 math libraries

These libraries are provided only as part of the paid-for Arm HPC compilers product

We are what is known as “the vendor maths library”

- This means we should always be the end user’s best option for the highest performing implementations of the functionality of the architecture
- Other examples are Intel MKL (on x86) and IBM’s ESSL (on POWER)
- Some systems integrators will also provide their own, e.g. Cray’s libsci

Common open source alternatives are OpenBLAS and ATLAS

Note that the Arm Performance Libraries are for 64-bit Armv8 systems only

Alternative library choices

If you weren't using Arm Performance Libraries, what can you use?

- Intel MKL (x86 only)
  - BLAS + LAPACK + FFT
  - + sparse solvers
  - + machine learning
  - + random number generators
- OpenBLAS
  - BLAS + selected LAPACK
- ATLAS
  - BLAS + LAPACK
- FFTW
  - FFT

BLIS

- BLAS + LAPACK

PLASMA

- BLAS + LAPACK

MAGMA (for nVidia GPUs)

- BLAS + LAPACK

cuBLAS (for nVidia GPUs)

- Non-standard BLAS + LAPACK

arm

Validated with NAG's test suite

NAG, the Numerical Algorithms Group are a company from Oxford, UK, specialising in developing mathematical routines

They have been around for almost 50 years and have been involved with almost all vendor maths libraries

They provided us with their validation test suite

- This enables us to test every build of the library to ensure that all changes we make still provide numerical accuracy to the end-user

NAG are also under contract with us to provide support if we discover any issues with code they have supplied

They also provide us updated code-drops when new versions of the base libraries are released

Commonly used low-level math routines

The libraries we include are known as BLAS, LAPACK and FFT

Most routines come in a four varieties (where appropriate)

- Single precision real : Routines prefixed by ‘S’
- Double precision real : Routines prefixed by ‘D’
- Single precision complex : Routines prefixed by ‘C’
- Double precision complex : Routines prefixed by ‘Z’

- The rest of the name (normally) describes something about what the routine does
  - E.g. the matrix-matrix multiplication routine DGEMM is a
    - D – Double precision
    - GE – Matricies given in GEneral format
    - MM – Matrix-Matrix multiplication is performed

BLAS

BLAS, the Basic Linear Algebra Subroutines, is a standard API

- It is provided on all systems, used by a wealth of scientific codes for vector and matrix maths
- It was designed for Fortran, but is callable from all languages

These routines are come in three levels

- BLAS level 1 – vector-vector operations, e.g. DCOPY, DAXPY, DDOT
- BLAS level 2 – matrix-vector operations, e.g. DGEMV, DTRMV, DGER
- BLAS level 3 – matrix-matrix operations, e.g. DGEMM, DTRMM, DTRSM

42 BLAS routines in total

Providing incredibly high performing versions of these routines is the team's main work

LAPACK

LAPACK, the Linear Algebra Package, is a another standard API

- It is provided on all systems, used by a wealth of scientific codes for solving equation systems
- It was designed for Fortran, but is callable from all languages
- We currently support LAPACK 3.7.0

The routines in LAPACK are normally build on BLAS routines so work we do on BLAS routines increases performance of particular LAPACK routines, too

There are now around 1700 LAPACK routines

- Most we do not touch, just using the reference version from Netlib
- Certain ones are **very widely used**, and these are where we focus our attention
- The key names to look out for are:
  - Cholesky factorization : ?POTRF
  - LU factorization : ?GETRF
  - QR factorization : ?GETQR

Fast Fourier Transforms

- FFTs are very commonly used in a wide variety of applications. They allow some hard problems to be transformed into a way that can be solved much more easily.
- We ship an implementation compatible with FFTW3, the most popular interface currently available.
  - Including our own version of the FFTW3 header
  - Includes half-precision support as of 19.2.
  - Support for Basic, Advanced, Guru and MPI interfaces

Sparse Matrix-Vector Multiply

- No standard interface for SpMV, but closely matches Intel's inspector-executor model.
- Support CSC, CSR, COO matrix formats.
- Much faster than alternatives on AArch64 (e.g. Eigen)

Choices in which library version users want

Unfortunately there are a few fundamental choices that users can make which mean that we cannot just ship a single library.

- OpenMP
  - Users may want the maths library they are using to take advantage of all cores available, or they may not want any OpenMP done by our library.
- Having 32-bit or 64-bit integers
  - This issue confuses everyone at some stage. If you need to run a code with very large counts of something then it is necessary to use 64-bit integers
- Compiler choice
  - Users must now also choose to have the library match a GCC or and Arm Compiler build
    - This is necessary as the Fortran ABI differs for complex functions (of which we have two) and there are also different levels of OpenMP support

Micro-architectural tuning

In order to achieve the best performance possible on all partner systems we need to do different micro-architectural tuning

All BLAS kernels are handwritten in *assembly code* in order to maximise overall performance

Different micro-architectures sometimes need fundamental differences in the instruction ordering – or even the instructions used

At run-time this work should all be transparent to the user

However multiple packages are typically available for users to choose from, and they need to load the appropriate module to set up their paths

Currently available are versions for:

- A72
- Cavium ThunderX2
- Generic AArch64

Benchmarking

- Included with our NAG validation suite is a less comprehensive timing suite
- In total we benchmark around 140 routines
- For a full coverage every routine needs testing at a variety of :
  - problem sizes
  - matrix shapes
  - thread counts
- We compare our results to open source alternatives on AArch64
- …as well as some comparisons against MKL

arm

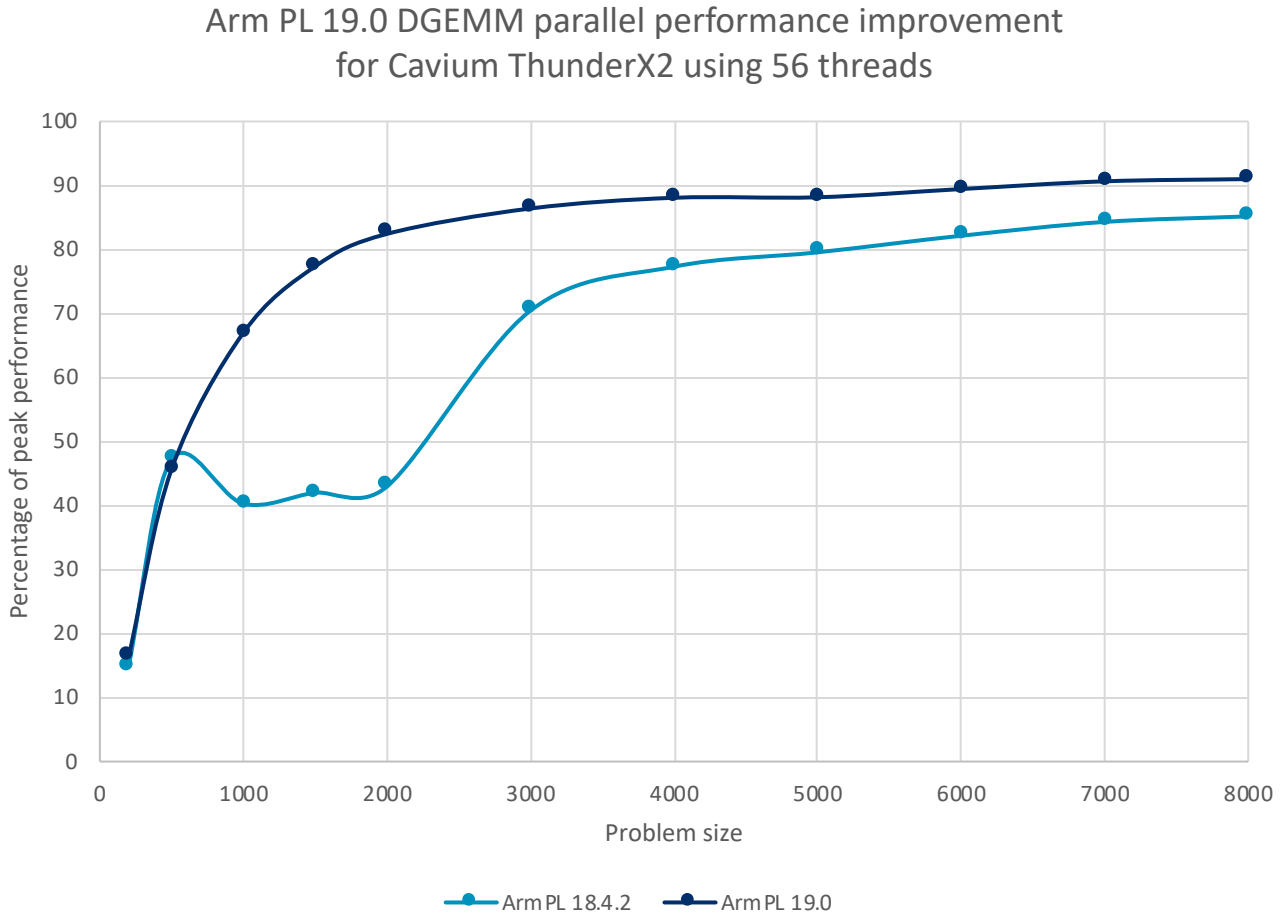
DGEMM performance

Excellent serial and parallel performance

Achieving very high performance at the node level leveraging high core counts and large memory bandwidth

Single core performance at
95% of peak for DGEMM (not shown)

Parallel performance at 90% of peak
Improved parallel performance for small problem sizes

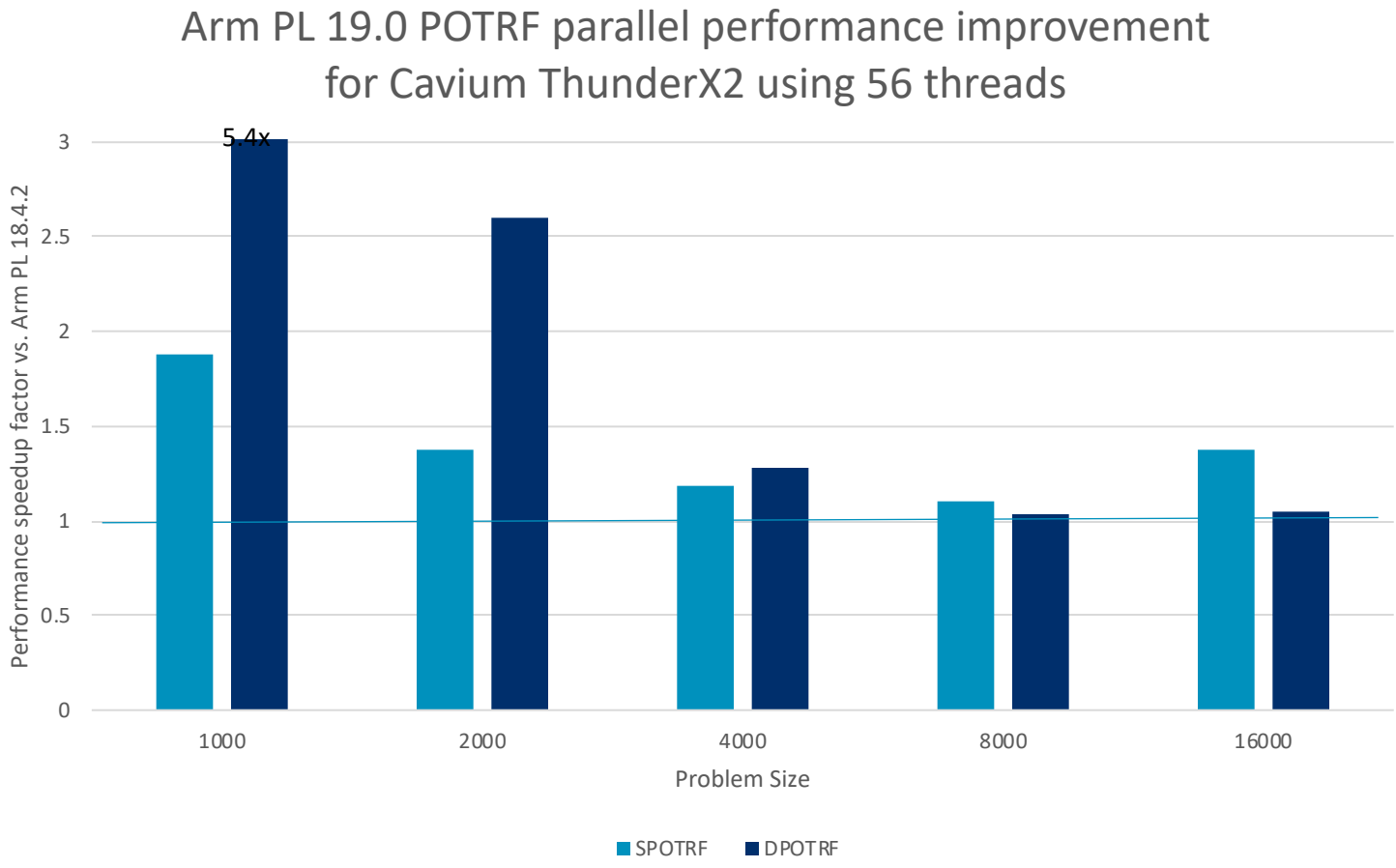


arm

LAPACK performance

Improved load balance for xPOTRF, xGEQR and xGETRF (Cholesky, QR and LU factorization, respectively) when using many threads.

Chart shows Cholesky factorization routines SPOTRF and DPOTRF.

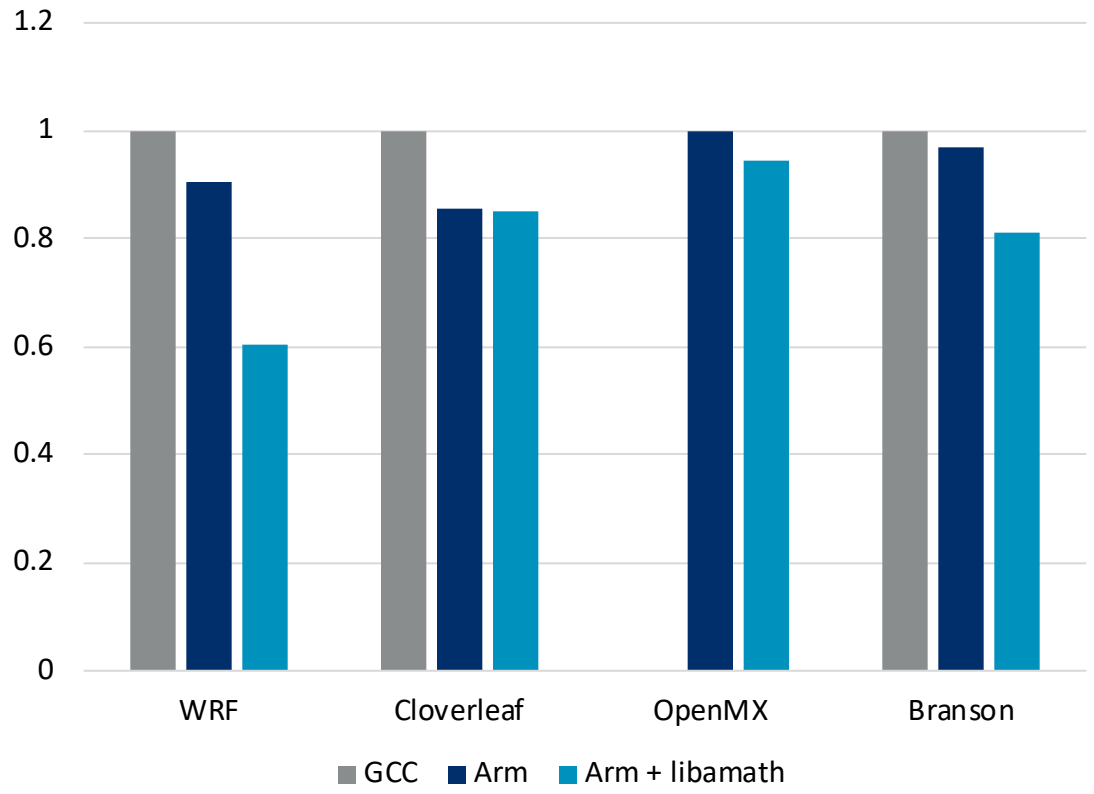


arm

Math Routine Performance

Distribution of https://github.com/ARM-software/optimized-routines

Normalised runtime



Arm PL provides libamath

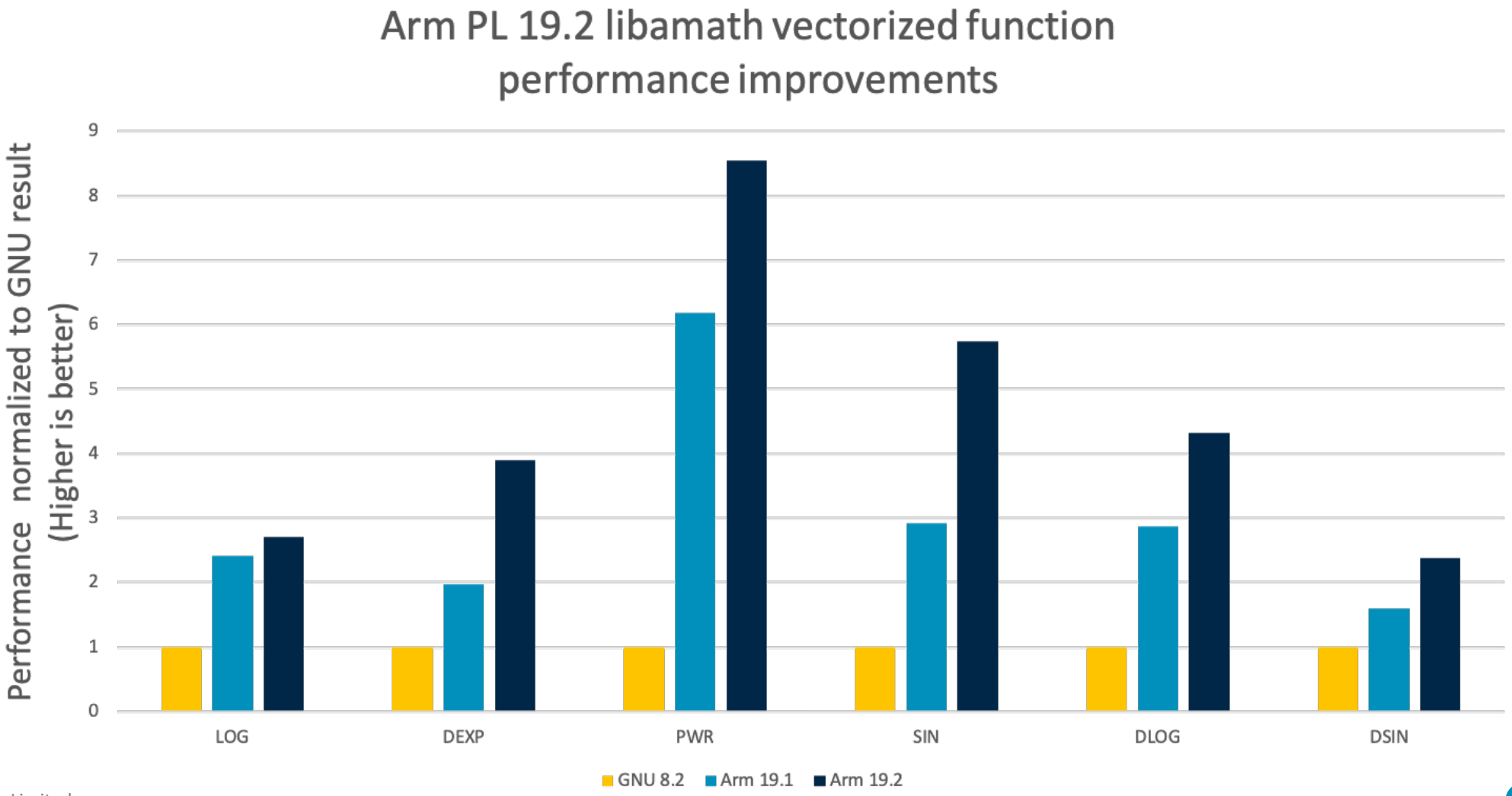
- With Arm PL module loaded, include `-lamath` in the link line.
- Algorithmically better performance than standard library calls
- No loss of accuracy
- single and double precision implementations of: `exp()`, `pow()`, and `log()`
- single precision implementations of: `sin()`, `cos()`, `sincos()`, `tan()`

arm

Improvements in 19.2

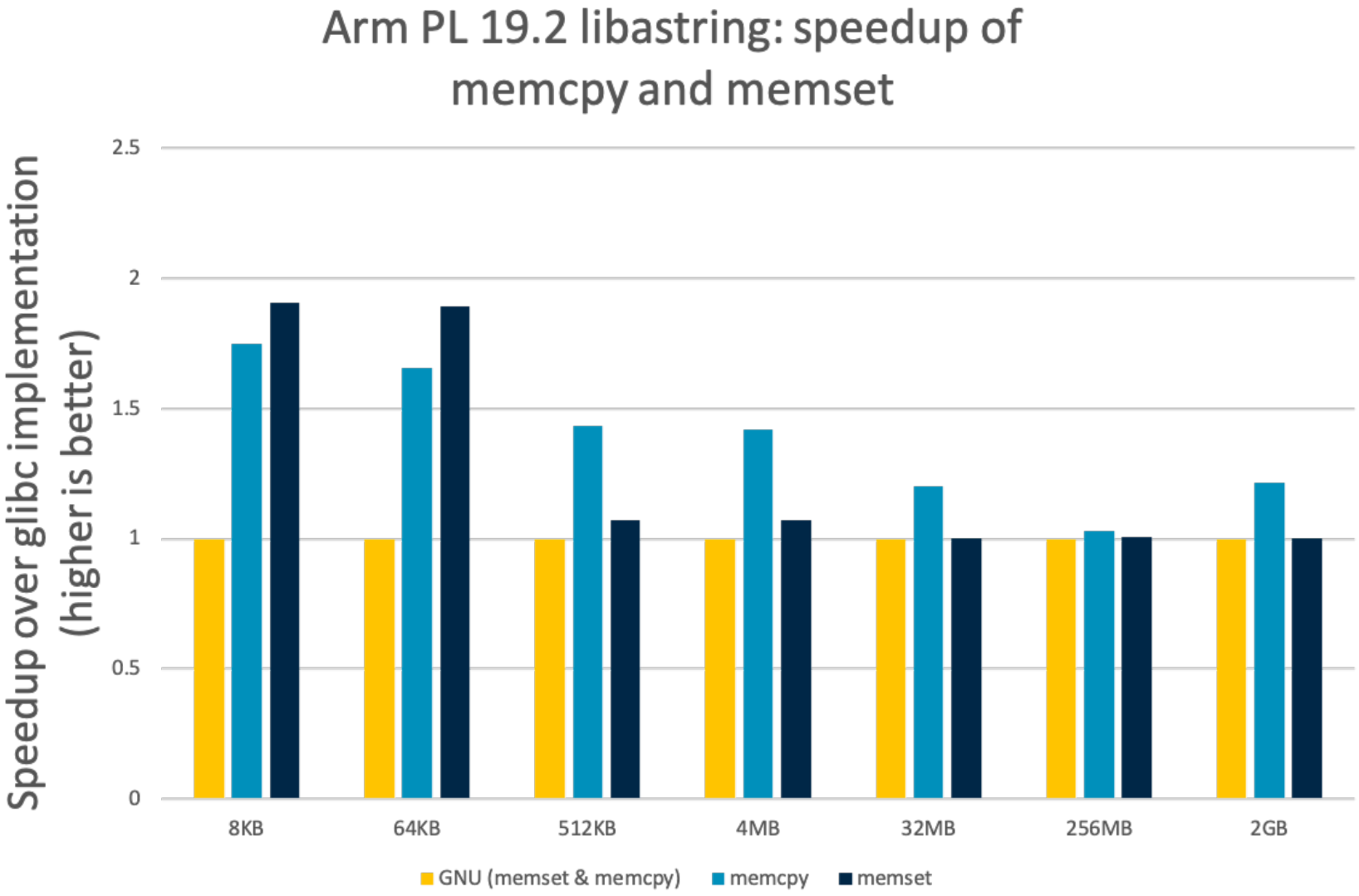
- Faster libamath
  - vectorized versions of sin, cos, exp and log in both single and double precision.
- Introduction of libastring
  - Optimised versions of common string.h functions (memcpy, memset).
- Faster FFTs
- Half precision FFTs and GEMM

Improvements in 19.2: Faster libamath



arm

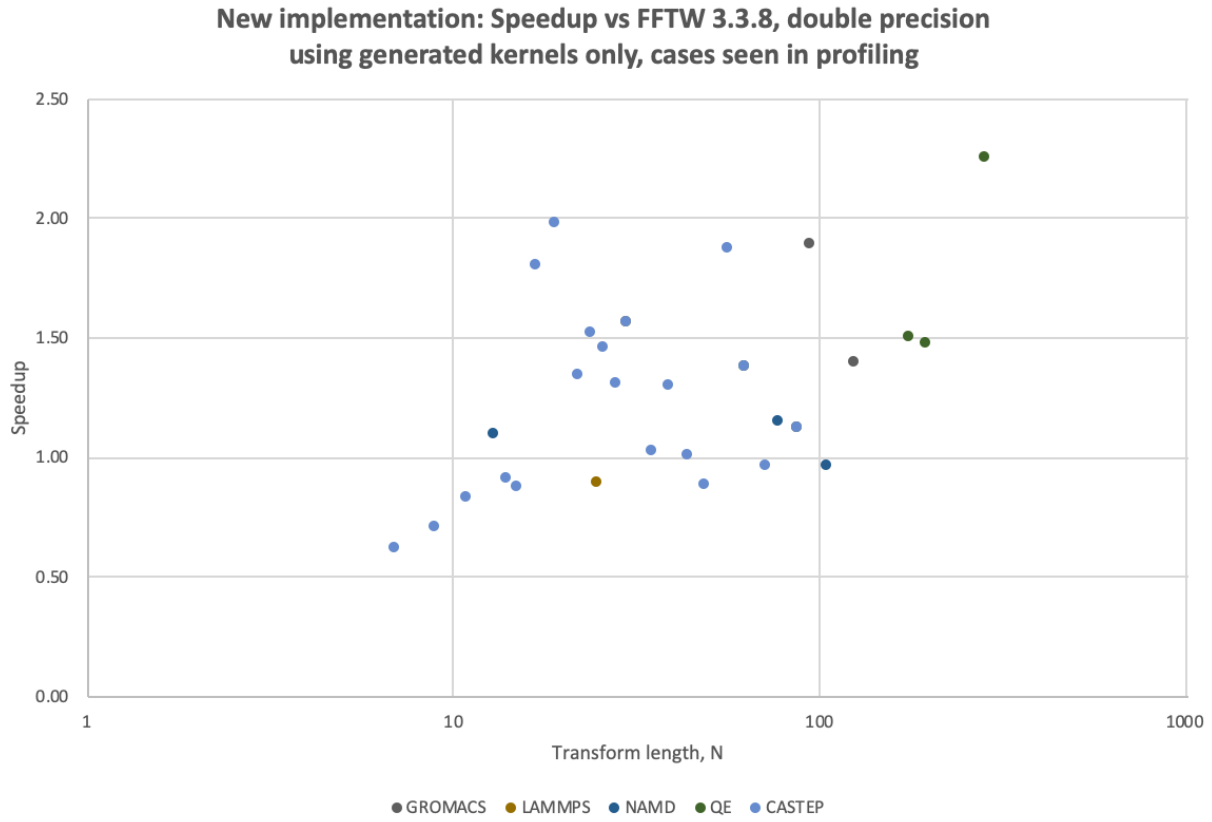
Improvements in 19.2: Libastring

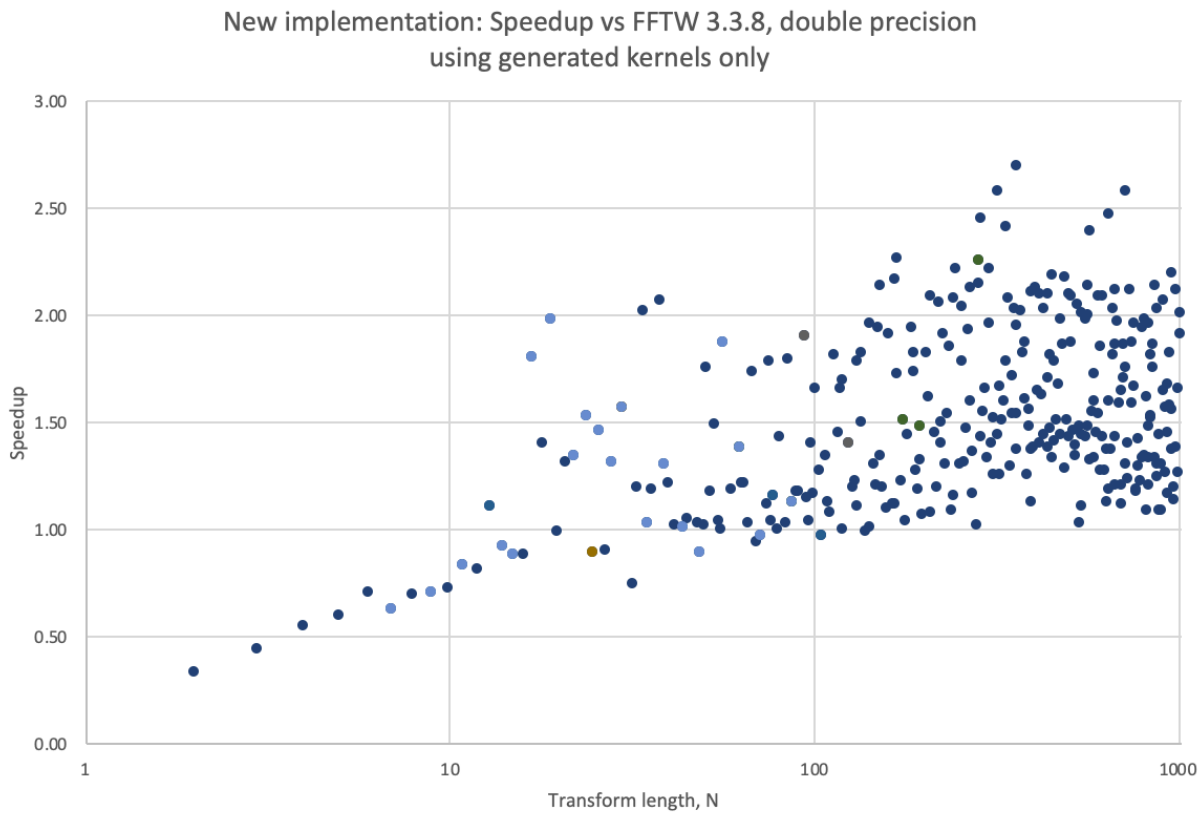


arm

Improvements in 19.2: Faster FFTs

- Competitive with FFTW for key workloads, faster in many cases.





arm

Improvements in 19.2: Half-precision FFTs and GEMM

- Will transparently use half-precision instructions if available (v8.2), else fall back to upcasting to single-precision.
- Similar to the existing FFT3 and GEMM function signatures.

```
 1  /* Include Arm Performance Libraries FFT interface. */
 2  #include "fftw3.h"
 3
 4  /* Declare half-precision arrays to be used */
 5  __fp16 *in = ...;
 6  fftwh_complex *out = ...;
 7
 8  /* Plan, execute and destroy */
 9  fftwh_plan plan = fftwh_plan_many_dft_r2c(...);
10  fftwh_execute(plan);
11  fftwh_destroy_plan(plan);
```

```
void hgemm_(const char *transa, const char *transb,
            const armpl_int_t *m, const armpl_int_t *n, const armpl_int_t *k,
            const __fp16 *alpha, const __fp16 *a, const armpl_int_t *lda,
            const __fp16 *b, const armpl_int_t *ldb,
            const __fp16 *beta, __fp16 *c, const armpl_int_t *ldc);
```


arm

Open source libraries for improved performance

Arm Optimized Routines

https://github.com/ARM-software/optimized-routines

These routines provide high performing versions of many math.h functions

- Algorithmically better performance than standard library calls
- No loss of accuracy

SLEEF library

https://github.com/shibatch/sleef/

Vectorized math.h functions

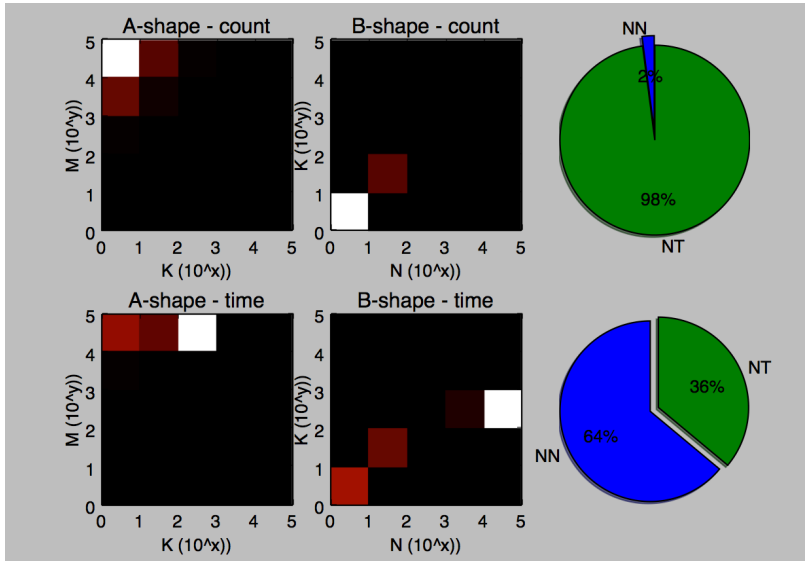
- Provided as an option for use in Arm Compiler

Perf-libs-tools

https://github.com/ARM-software/perf-libs-tools

Understanding an application's needs for BLAS, LAPACK and FFT calls

- Used in conjunction with Arm Performance Libraries can generate logging info to help profile applications for specific case breakdowns



Example visualization: DGEMM cases called

arm

Community Libraries

https://gitlab.com/arm-hpc/packages/wikis/

Package Support

- Over 54 packages in Arm's Package Wiki
- Trilinos, PETSc, Hypre, SuperLU, ScaLAPACK, NetCDF, HDF5, etc.
  - Tested; works well with Arm and GNU compilers
  - Soon tested at scale on *Astra*

Testing and Development

Multiple partners making ThunderX2 available to open source projects for CI/CD

- packet.net, Verne Global

Resourcing

Arm support:

- Part of broader NRE and commercial projects
- Currently providing reactive support to users at over 40 HPC sites worldwide

Arm HPC Ecosystem Guides

https://developer.arm.com/solutions/hpc

More in depth guides on how to build some packages, e.g. FFTW.

- Run the `configure` script:

```
./configure --enable-neon --enable-shared --enable-openmp
--enable-mpi CC=armclang CXX=armclang++ FC=armflang
F77=armflang --prefix=$INSTALL_DIR
```

Where:

  - `--enable-neon`: enables the set of NEON SIMD instructions.
  - `--enable-shared`: builds shared libraries.
  - `--enable-openmp`: builds threaded, OpenMP libraries.
  - `--enable-mpi`: builds MPI-enabled libraries.

- To ensure `libtool` uses the correct `linker` flags with armflang, it must be patched, post-configure, using:

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

- Make, test, and install the double precision build:

```
make -j
make check
make install
```

Practical usage

arm

Read the docs

https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/arm-allinea-studio

- Getting started
- Fortran and C/C++ language reference guides
- Usage guides for users of other compilers (GNU, Intel, PGI)
- Best practice (mostly pragmas and compiler flags) to encourage auto-vectorization
- Optimization remarks
- Fortran and OpenMP support levels

How to use ArmPL

Three decisions

Decision 1: which microarchitecture?

Decision 2: OpenMP?

Decision 3: 32-bit or 64-bit integers?

What this gives

- ArmPL
  - BLAS
  - LAPACK
  - FFTs
  - Sparse
- Libamath
  - Optimised scalar math
  - Optimised vector math
- Libastring
  - Optimised string.h routines

How to use ArmPL

Basic usage (Native architecture, 32-bit integers, serial ArmPL)

Arm Compiler for HPC

-mcpu=native -armpl

GNU

module load <Architecture>

-mcpu=native

-I $ARMPL_INCLUDES

-larmpl

arm

How to use ArmPL

Decision 1: Which micro-architecture?

Arm Compiler for HPC

-mcpu=native

GNU

-mcpu=native

arm

How to use ArmPL

Decision 2: OpenMP?

Arm Compiler for HPC

To use serial ArmPL

-armpl (defaults to no OpenMP)

(or -armpl=nomp)

To use parallel ArmPL

-armpl –fopenmp

(or -armpl=mp)

GNU

To use serial ArmPL

-I $ARMPL_INCLUDES

–larmpl

To use parallel ArmPL

-I $ARMPL_INCLUDES_MP

–larmpl_mp

How to use ArmPL

Decision 3: 32-bit or 64-bit integers?

Arm Compiler for HPC

To use 32-bit integers

-armpl (defaults to 32-bit)

(or -armpl=lp64)

To use 64-bit integers

-armpl –i8

(or -armpl=ilp64)

GNU

To use 32-bit integers

-I $ARMPL_INCLUDES

–larmpl

To use 64-bit integers

-I $ARMPL_INCLUDES_ILP64 –DINTEGER64

–larmpl_ilp64

Compiler options - general

armclang --help

`-o <file>`	Write output to <file>.
`-c`	Only run preprocess, compile, and assemble steps.
`-g`	Generate source-level debug information.
`-Wall`	Enable all warnings.
`-w`	Suppress all warnings.
`-fopenmp`	Enable OpenMP.
`-O`*n*	Level of optimization to use (0, 1, 2, 3).
`-Ofast`	Enables aggressive optimization of floating point operations.
`-ffp-contract=(fast|on|off)`	Allow fused floating point operations (eg FMAs).

`-Rpass=\(loop-vectorize|inline\)`
`-Rpass-missed=\(loop-vectorize|inline\)`
`-Rpass-analysis=\(loop-vectorize|inline\)`
	Optimization Remarks is a feature of LLVM compilers that provides you with information about the choices made by the compiler.

Compiler options - Fortran

armclang --help

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

Pragmas to control vectorization

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

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

#pragma clang loop unroll_count(_value_)

- Forces a scalar loop to unroll by a given factor

#pragma clang loop interleave_count(_value_)

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

Fortran directives to control vectorization

!dir$ ivdep

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

!dir$ vector always

- Always vectorize, if it’s safe to do so

!dir$ novector

- Disable vectorization

OpenMP

OpenMP is a shared memory parallelism paradigm

It enables a program to use multiple computational cores in order to complete a calculation quicker

Arm Performance Libraries users can control the number of threads used by setting the environment variable `OMP_NUM_THREADS`

- This normally defaults to the number of cores on your system

We strongly recommend users to also set `OMP_PROC_BIND` to "`true`", "`close`" or "`spread`" depending on their needs

- If unset the threads are not pinned to cores and may migrate around the system

Users may also choose to do "nested parallelism" wanting multi-threaded functions from within an already parallel environment

- Use of `OMP_DYNAMIC=true` and `OMP_NESTED=true` is necessary

arm

Porting to Arm Performance Libraries

Most users should be able to port without issues since all calls should be the same as they have used on previous systems

The only change that may be necessary will be including "`armpl.h`"
(as opposed to e.g. "`mkl.h`")

BLAS and LAPACK guidance

We still do not have BLAS and LAPACK routines documented on developer.arm.com but recommend www.netlib.org/lapack/explore-html

All users should take note of the required inputs and their sizes

- Also of note is any values that may be overwritten during computation

```
subroutine dgemm     (  character                    TRANSA,
                        character                    TRANSB,
                        integer                      M,
                        integer                      N,
                        integer                      K,
                        double precision             ALPHA,
                        double precision,
                        dimension(lda,*)             A,
                        integer                      LDA,
                        double precision,
                        dimension(ldb,*)             B,
                        integer                      LDB,
                        double precision             BETA,
                        double precision,
                        dimension(ldc,*)             C,
                        integer                      LDC
                     )
```


arm

BLAS and LAPACK guidance

We still do not have BLAS and LAPACK routines documented on developer.arm.com but recommend www.netlib.org/lapack/explore-html

All users should take note of the required inputs and their sizes

- Also of note is any values that may be overwritten during computation

[in]	**K**	K is INTEGER On entry, K specifies the number of columns of the matrix op(A) and the number of rows of the matrix op(B). K must be at least zero.
[in]	**ALPHA**	ALPHA is DOUBLE PRECISION. On entry, ALPHA specifies the scalar alpha.
[in]	**A**	A is DOUBLE PRECISION array, dimension (LDA, ka), where ka is k when TRANSA = 'N' or 'n', and is m otherwise. Before entry with TRANSA = 'N' or 'n', the leading m by k part of the array A must contain the matrix A, otherwise the leading k by m part of the array A must contain the matrix A.
[in]	**LDA**	LDA is INTEGER On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, m), otherwise LDA must be at least max(1, k).
[in]	**B**	B is DOUBLE PRECISION array, dimension (LDB, kb), where kb is n when TRANSB = 'N' or 'n', and is k otherwise.

arm

FFT instructions

We implement the majority of the FFTW3 interface, so the best place to start is their own documentation: http://www.fftw.org/fftw3_doc/FFTW-Reference.html

For half-precision FFTs, make sure you include the correct fftw3.h header!

4.3.1 Complex DFTs

```
fftw_plan fftw_plan_dft_1d(int n0,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_2d(int n0, int n1,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_3d(int n0, int n1, int n2,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft(int rank, const int *n,
                        fftw_complex *in, fftw_complex *out,
                        int sign, unsigned flags);
```

Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!