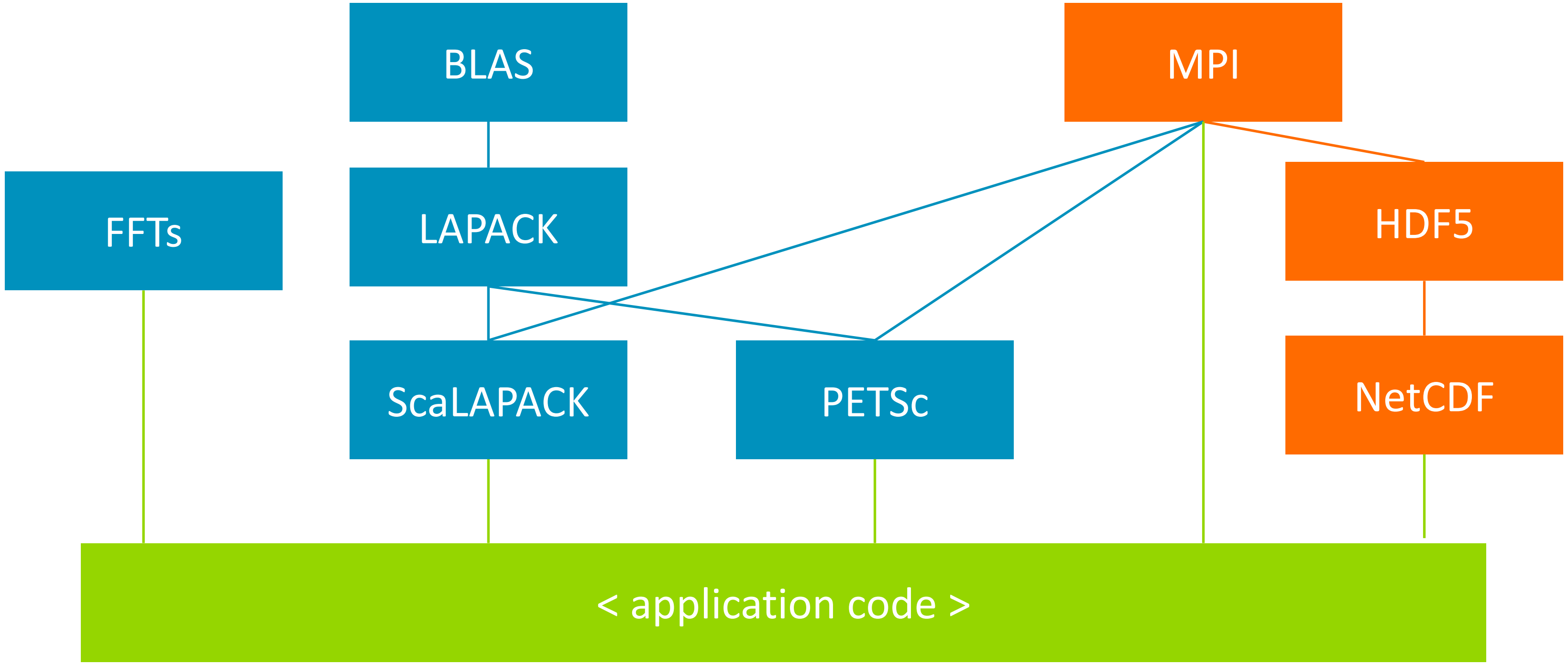
A brief introduction to open source maths libraries & Arm Performance Libraries

Chris Goodyer
July 2021

Maths libraries on Arm for HPC

- Numerical libraries are the bedrock of most scientific codes solved on HPC systems
  - Re-inventing the wheel is normally a bad idea
- There are hierarchies of tools that are built on each other
  - Today many of these work out-of-the-box
  - Those that need specific Arm optimizations are already starting to get these added
  - There are a mix of open source and proprietary libraries



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
- Sparse linear algebra and batched BLAS support
- libamath gives high-performing implementations of math.h function

Best-in-class serial and parallel performance

- Algorithmic, Armv8-A and SVE optimizations
- Tuning for specific platforms like Neoverse N1 in AWS

Validated and supported by Arm

- Validated with NAG's test suite, a de-facto standard
- Available in both commercial and free forms

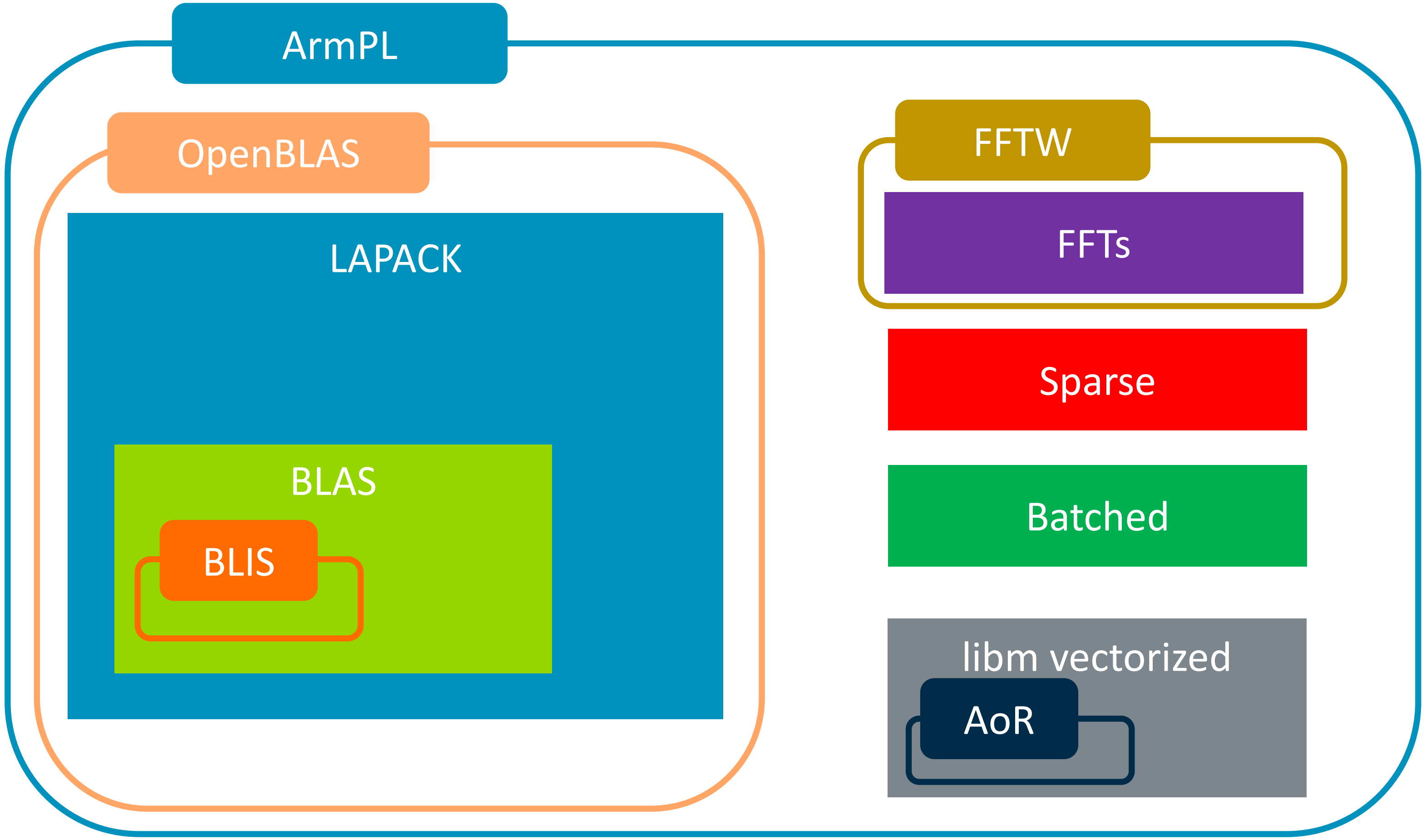
What does all this mean in practice?

Commercial 64-bit Armv8 math libraries

- These libraries are provided both as part of the paid-for *Arm Compiler for Linux* product and as a free version
  - ACfL
    - Versions compatible with both GCC and Arm Compiler for Linux
    - Microarchitectural optimizations included for Arm partner cores as well as Cortex/Neoverse ones
  - Free version
    - Only compatible with GCC
    - Microarchitectural optimizations included for only Arm-designed cores, such as Neoverse N1, as found in the AWS Graviton2 systems
- We are what is known as “the vendor maths library”
  - This means we should generally be the end user’s best option for the highest performing implementations of the functionality of the architecture
    - Open source alternatives may beat performance in some cases on some platforms
  - 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

What does Arm Performance Libraries provide?

…and how is this different to open source libraries



Key: Blocks are technologies
Curved shapes are libs

Notes:

- OpenBLAS mainly only optimizes BLAS
- BLIS provides excellent optimizations for a subset of BLAS functions
- ArmPL is also a delivery vehicle for *Arm Optimized Routines*, but also develop implementations to be open-sourced

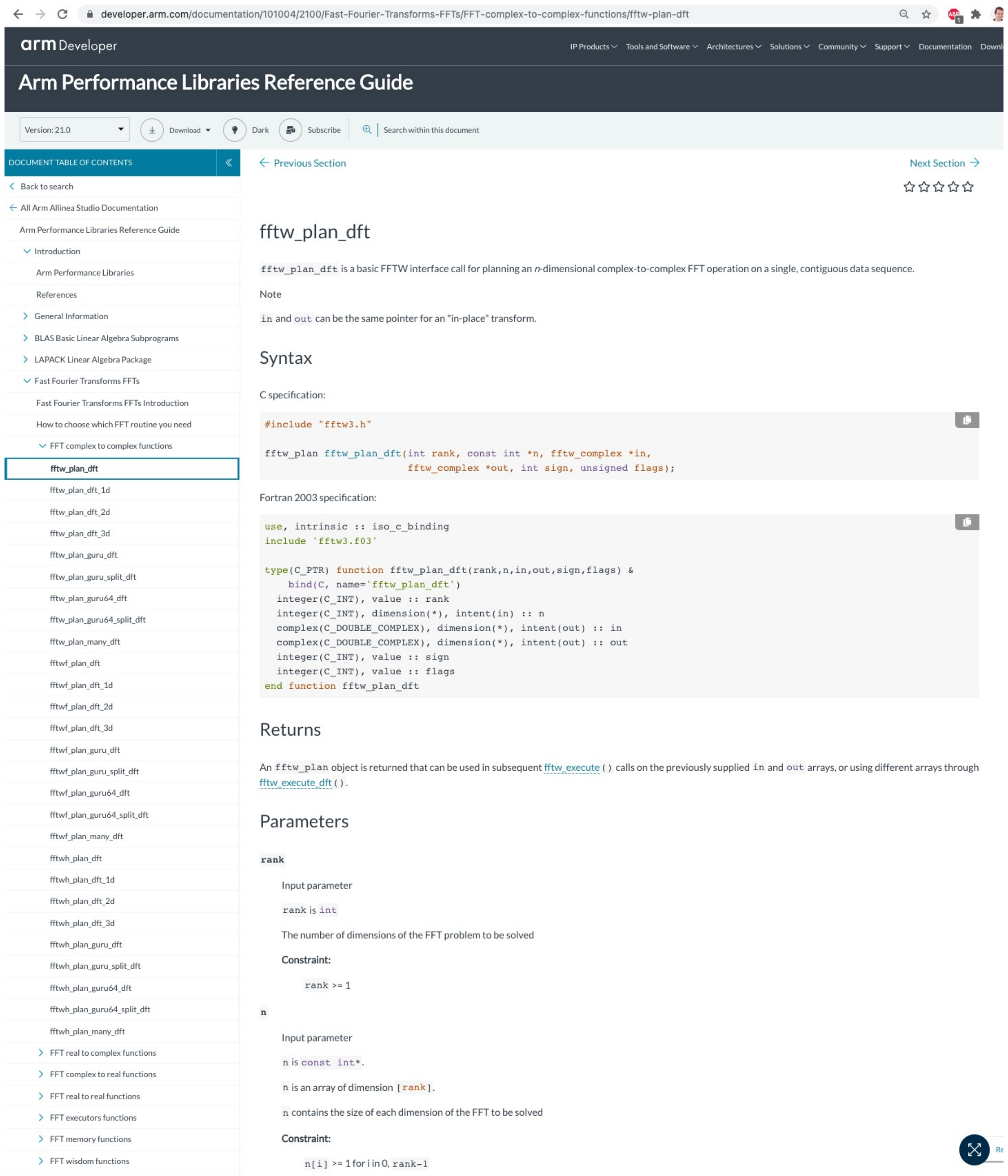
Choices in which library version users want

- 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.
    - Nested parallelism may be desired by the user
  - Having 32-bit or 64-bit integers (lp64 or ilp64)
    - 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
    - If using the full ACfL implementation, users must also choose to have the library match a GCC or an Arm Compiler build
    - Arm Compiler has added integration options that rather than having to explicitly specify the include directory and then link the library the "-armpl" option can be used in compile and link stages

- Open source libraries typically offer these same choices
  - They will be build-time options so you need to compile appropriately
- For FFTW the choice of single or double precision is also a buildtime choice

Documentation

- We have full documentation online at
  - https://developer.arm.com/documentation/101004/latest
- This includes full descriptions of the APIs we support
  - This includes FFTW calls!
  - Google for, e.g., “Arm DGEMM” normally gets our page as the first hit
- We also ship examples that give working cases illustrating how to call, compile and link
  - Both C and Fortran examples provided
- There is also a Getting Started guide online
  - https://developer.arm.com/tools-and-software/server-and-hpc/downloads/arm-performance-libraries/get-started-with-armpl-free-version



BLAS and LAPACK

Commonly used low-level math routines

- The standard linear algebra libraries used in HPC are BLAS and LAPACK
- 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 – Matrices given in GEneral format
    - MM – Matrix-Matrix multiplication is performed

Linear algebra

BLAS

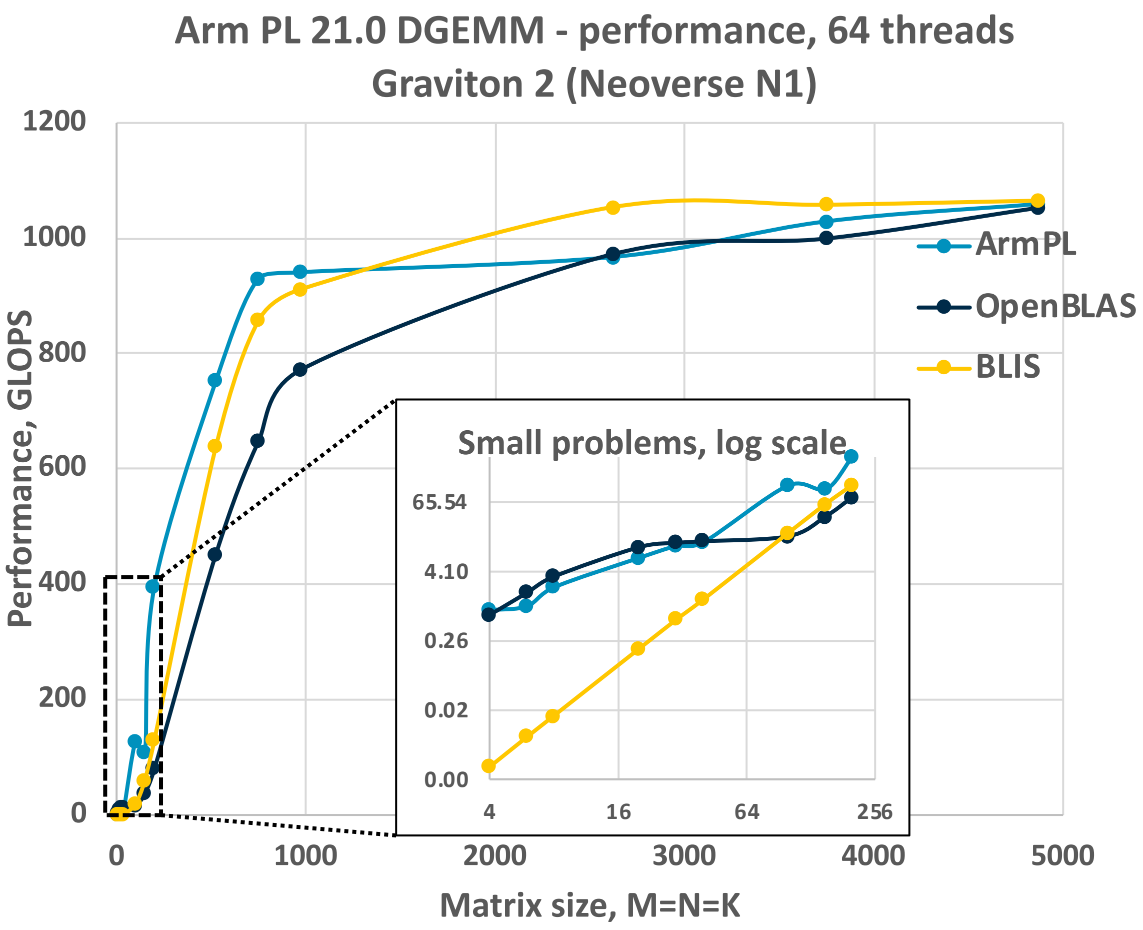
- Basic Linear Algebra Subroutines, is a standard API
  - It is provided on all systems, used by scientific codes for vector and matrix maths
  - It was designed for Fortran, but is callable from all languages
- These routines are come in three levels
  - BLAS level 1 – vector-vector operations,
    - 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
- 156 BLAS routines in total

LAPACK

- LAPACK, the Linear Algebra Package, is 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 3.9.0
- LAPACK is built on BLAS routines
- There are now around 1700 LAPACK routines

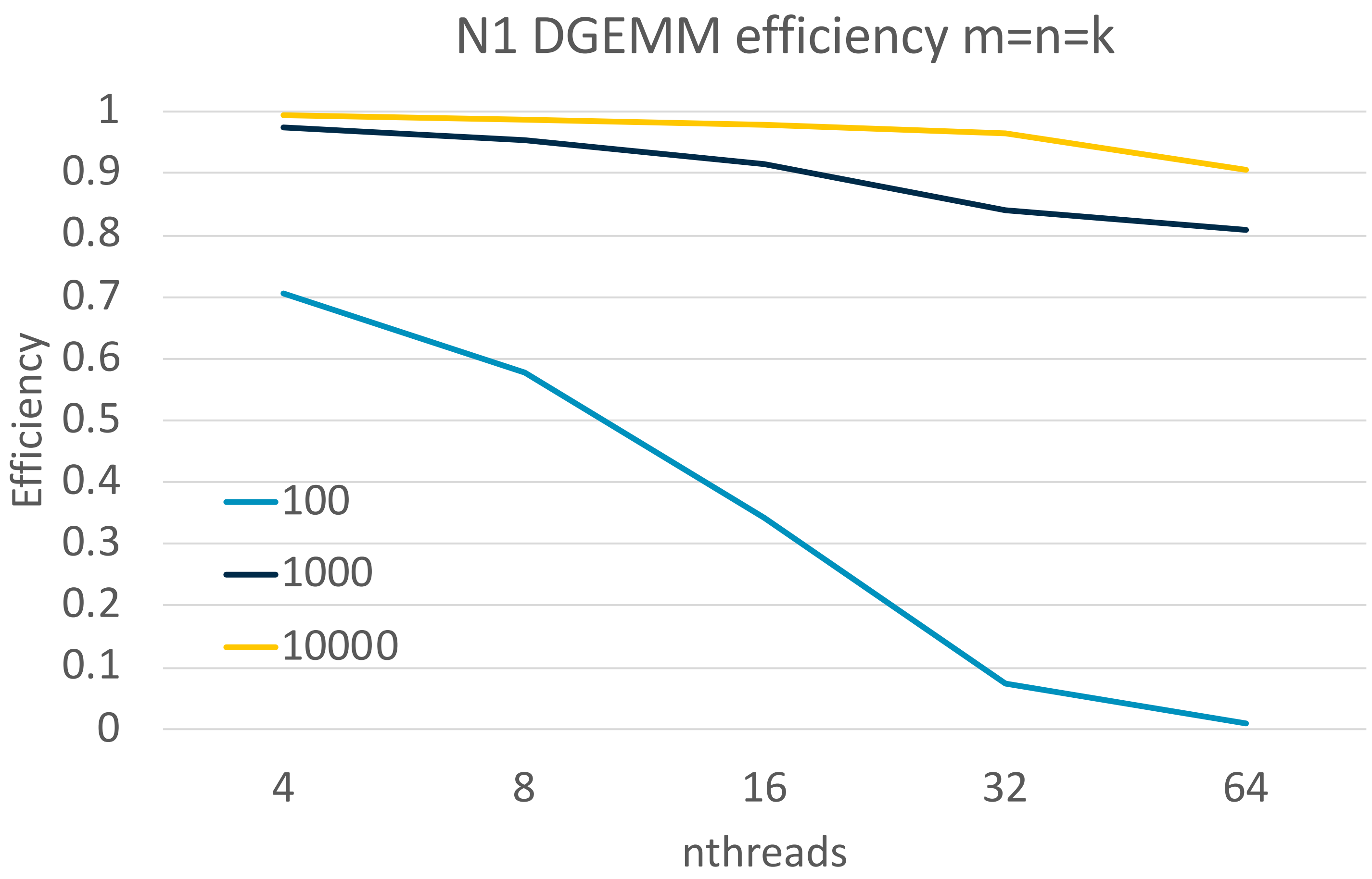
Micro-architectural tuning

- For the best performance possible micro-architectural tuning is needed
- All BLAS kernels are handwritten in *assembly code* in order to maximise overall performance
- Different micro-architectures may need differences in the instruction ordering – or even the instructions used
- In ArmPL the choice of implementation to be used is done automatically at runtime based on the hardware used
  - Open source libraries do not do this



Arm Performance Libraries: OpenMP Scaling on N1

Run on AWS Graviton2



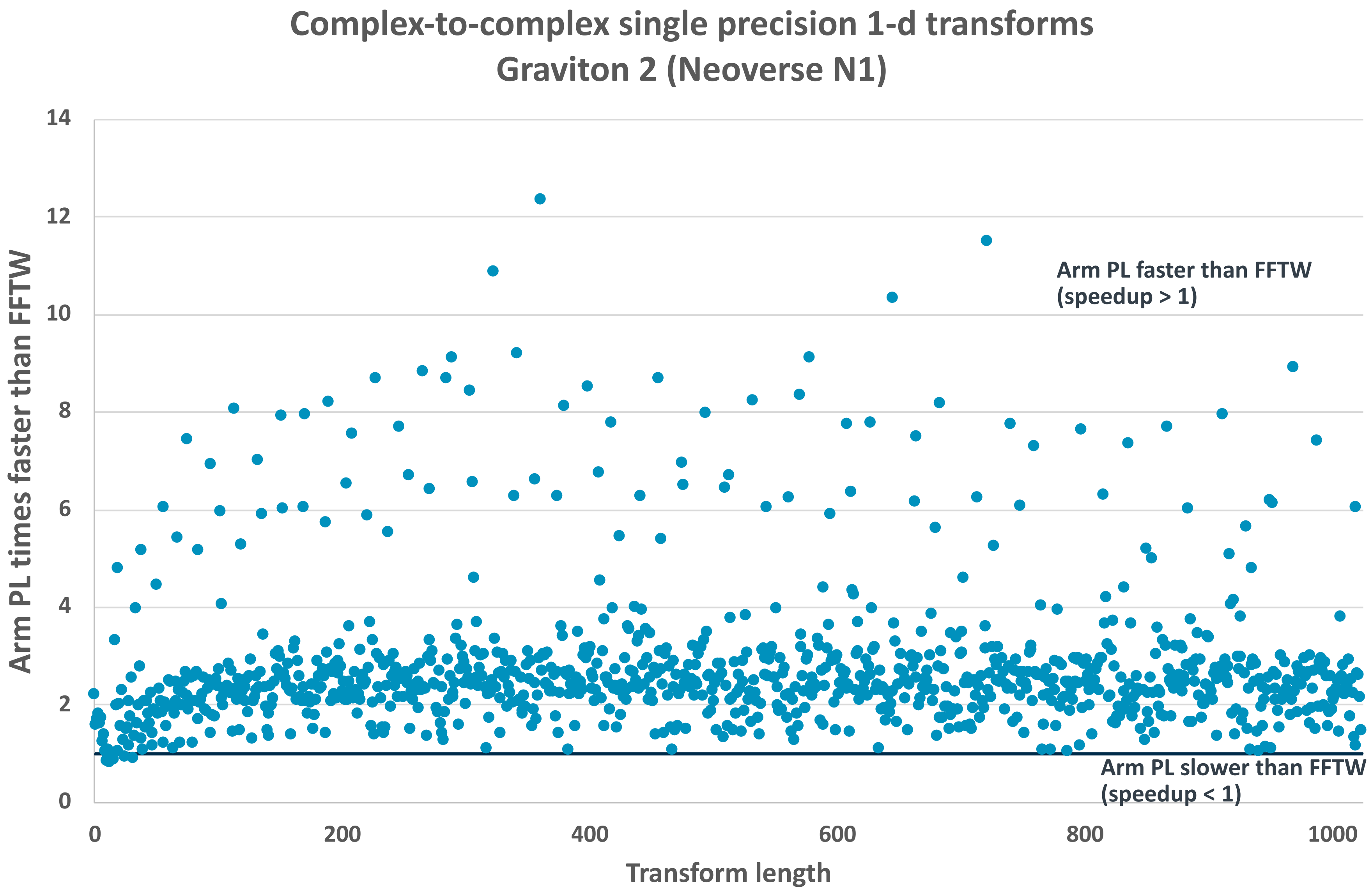
- Shown is DGEMM on square matrices using 64 threads on an AWS Graviton2
- Shown for matrix sizes of 100, 1,000 and 10,000
- Shows up to over 90% efficiency for large matrices at 64 cores

FFTs

Fast Fourier Transforms

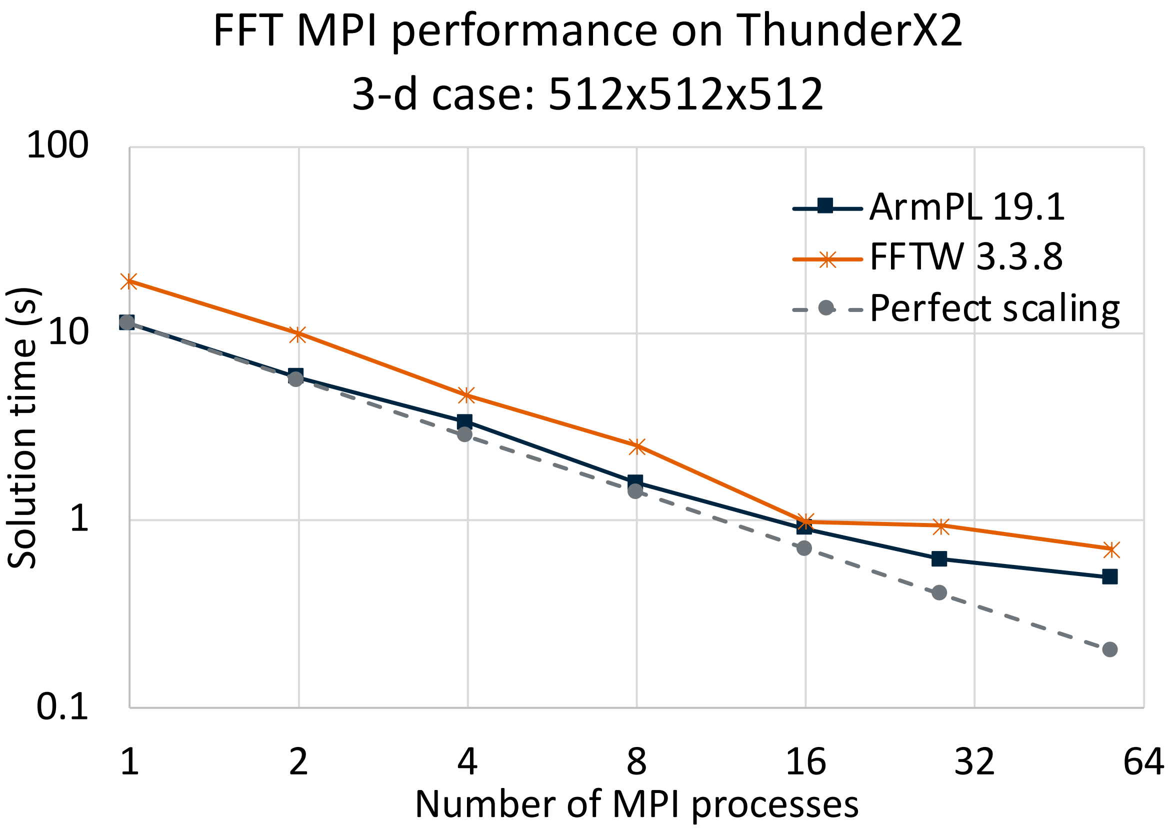
- FFTs are very commonly used in a wide variety of applications. They allow some hard problems to be transformed into a way that can be solved much more easily.
- FFTs are solved by taking time to do a "planning" stage first, before doing many fast "execute" calls of that plan
- The open-source FFTW3 interface has become the de facto standard in scientific applications
- Arm Performance Libraries includes the full set of all the Discrete Fourier Transform (DFT) functions using the FFTW3 interface
  - All functions available in:
    - 1-d, 2-d, 3-d and n-d
    - Complex-to-complex, real-to-complex and complex-to-real
    - All Discrete Cosine Transform (real-to-real) functions also supported
    - Single, double and half precision
  - FFTW's MPI interface is also supported
  - All FFTs transparently generated using a JIT compiler in the library

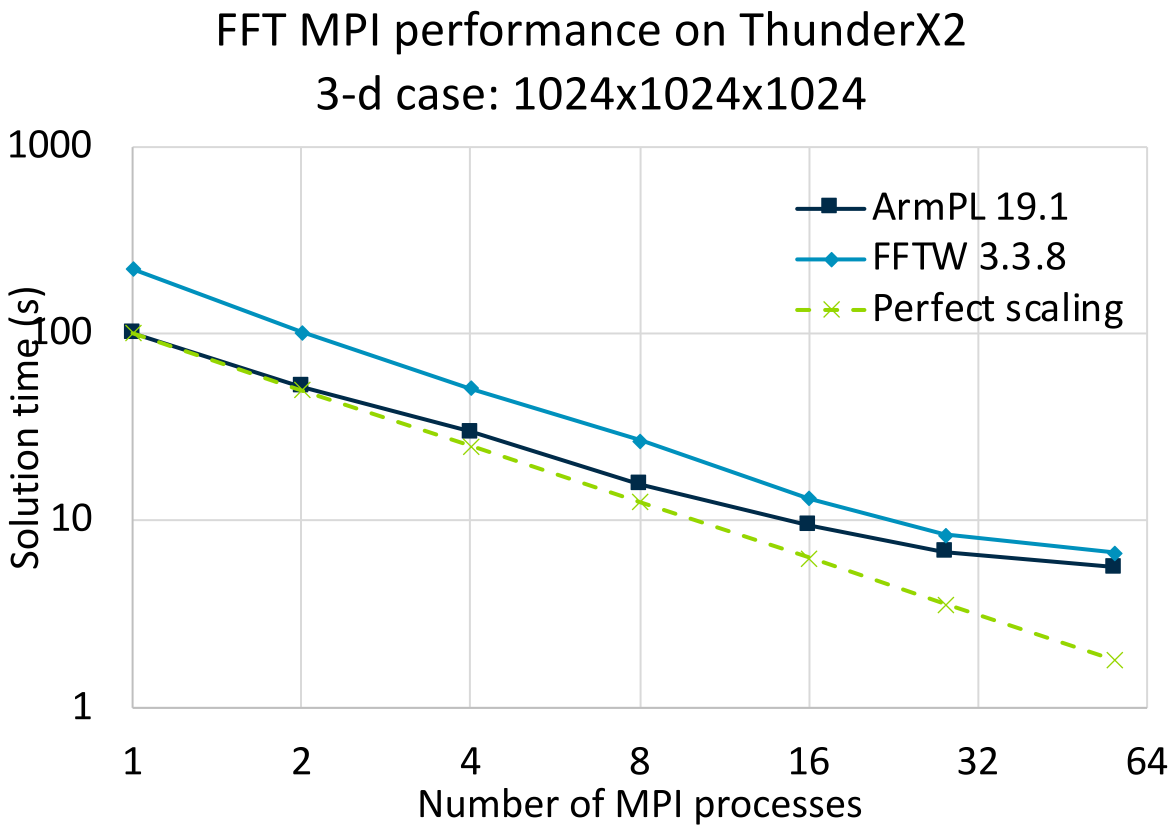
ArmPL 21.0 FFT vs FFTW 3.3.9



FFT MPI performance – 19.1

Scaling using FFTW MPI interface improved; now similar scaling to FFTW



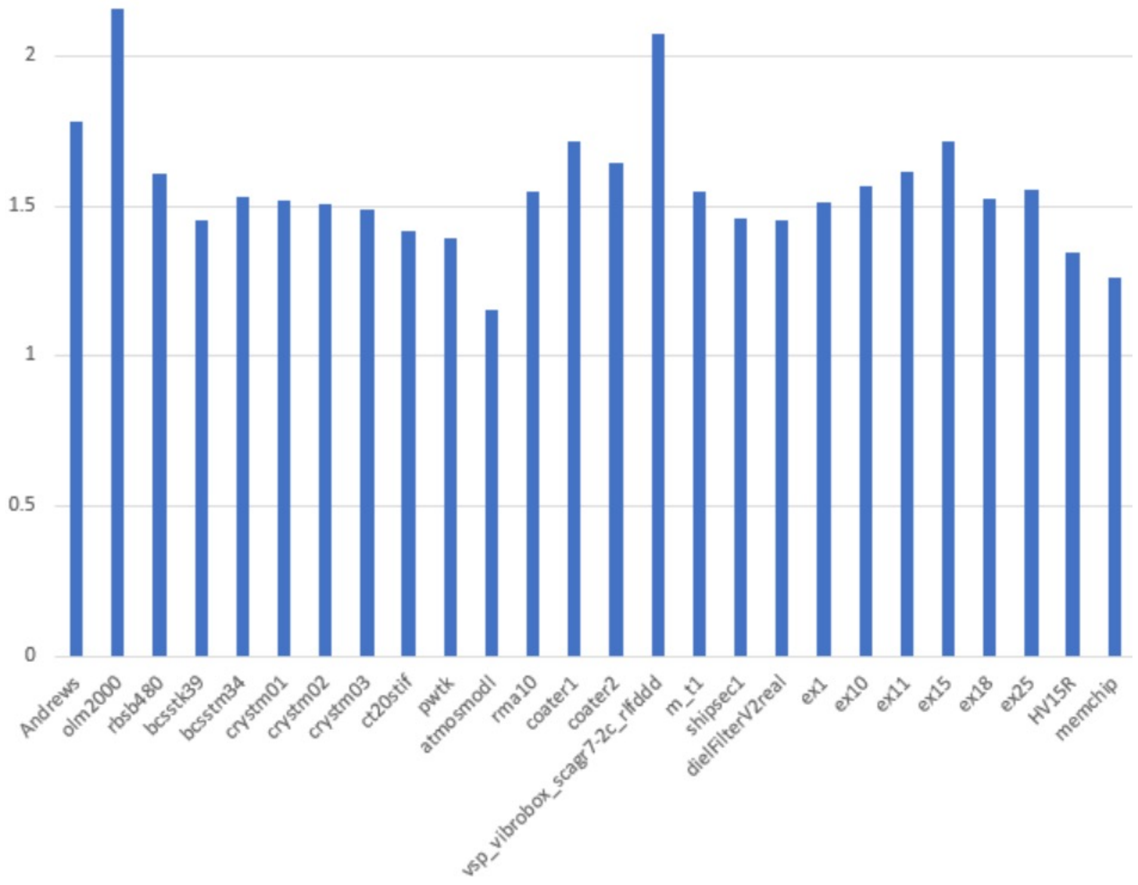


Sparse matrix support

Sparse matrix support

- Vendor maths libraries have not standard (like LAPACK) to use for high performing implementations of sparse functions
- Arm Performance Libraries has added high performing implementations of SpMV and SpMM
  - These rely on the code writer knowing about the possibility of reuse of matrices which is where the optimization potential (over CSR) comes from
- Operation schematic:
  - Create and fill sparse matrix object
  - Optimize for operation
  - Call <operation> many times

Speed-up for SpMV compared to CSR on a selection of sparse matrices



Batched BLAS/LAPACK interfaces

Batched BLAS

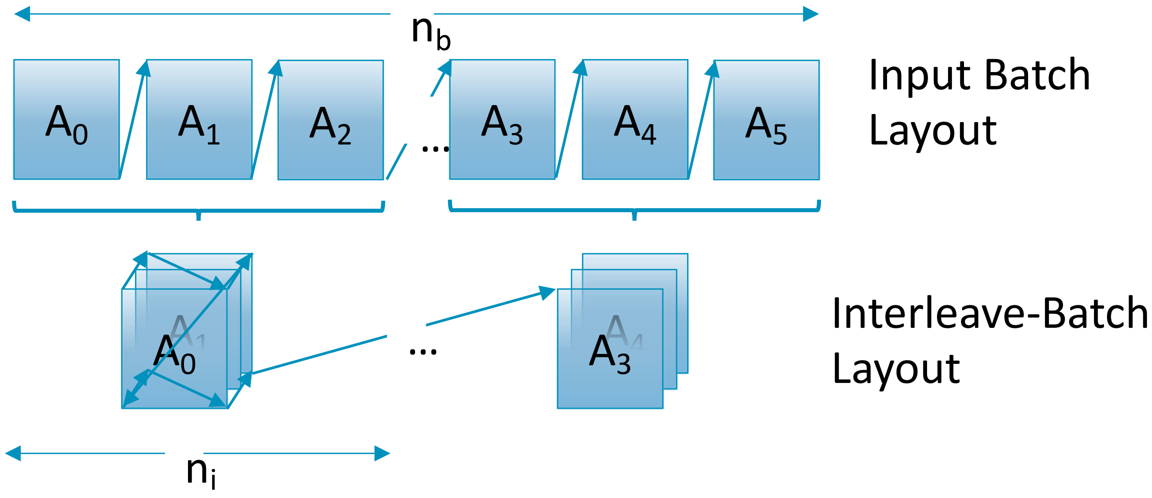
- Batched linear algebra is a growing area of importance for many real scientific codes
- This enables users to make a single library call and ask for the same operation, e.g. DGEMM, to be done on an array of input matrices
- This has the advantage of missing out the overhead of input parameter checking (which can be substantial on tiny cases) and allowing the library to do optimizations that would not be possible in individual calls
  - Parallelisation over sets of cases is an easy example of how this can be done
- Arm Performance Libraries support a batched operations using *gemm_batched()
  Even higher performance is possible if you know that data will be reused more than once, or you have more control of your input data
  - For these cases ArmPL has developed a set of functions where the data gets pre-interleaved…

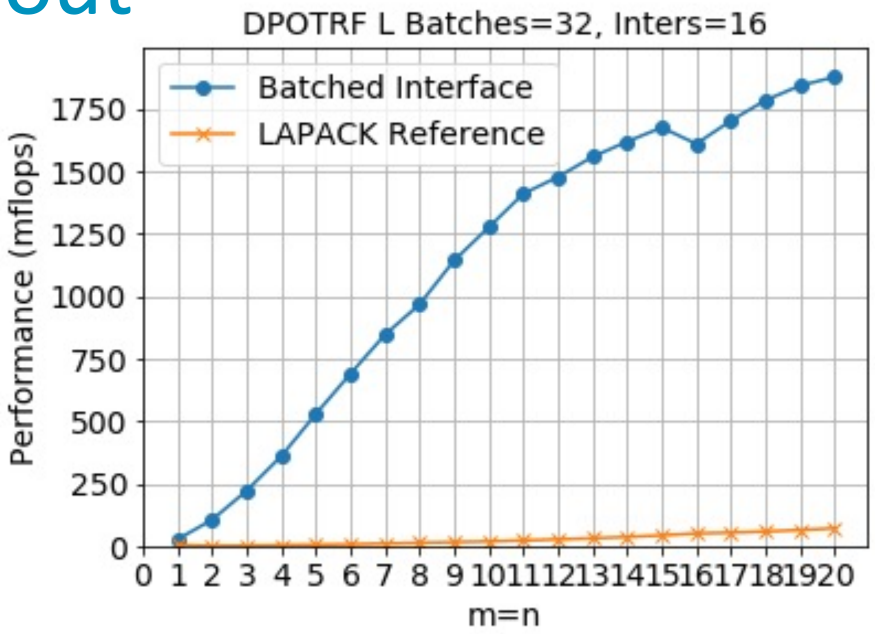
New in ArmPL 21.0: Interleave Batch layout

Interleaved batched BLAS/LAPACK functions

Using batched interfaces

- Executing a routine on a batch of n_p matrices
- Split batch of matrices into n_b times length-n_i (sub-)batches





Functions supported

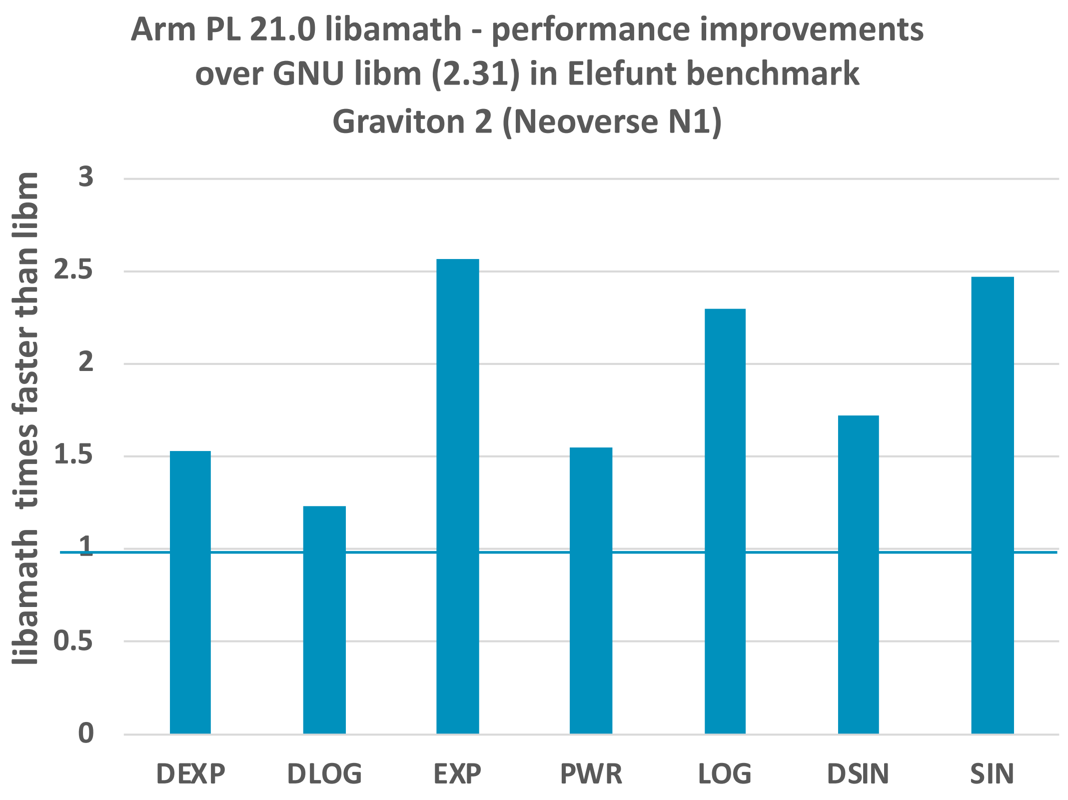
- BLAS: ddot, dger, dgemm, dgemv, dscal, dtrmm, dtrsm, dtrsv
- LAPACK: dgeqrf (QR), dgetrf (LU), dpotrf (Cholesky)
  - plus dormqr, dorgqr for multiplying and generating Q)

libamath

Optimized libm functions

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

Normalised runtime



ArmPL includes libamath and libastring

- Algorithmically better performance than standard library calls
- No loss of accuracy
- Enabled by default with Arm Compiler for Linux
- Single and double precision implementations of:
- `erf(),erfc()`
- single and double precision implementations of: `exp(),pow(),log(),log10()`
- single precision implementations of: `sin(),cos(),sincos()`
- Efficient **memory/string** functions from `string.h`
- Enable **autovectorization** of math and string routines in ACfL by adding `-armpl` or `-fsimdmath`

How to use – including with SPACK

Code changes needed to use Arm Performance Libraries

- If your code is already set up to use another BLAS, LAPACK or FFTW3 implementation then it will just work!
  - All functions use the standard APIs
  - The only exceptions are the sparse and batched interfaces
- If using the FFTW MPI interface you must ensure that the ArmPL version of "`fftw3.h`" is used rather than the FFTW version
- The C correct header file to include is "armpl.h"
  - although "`blas.h`" and "`lapack.h`" will also work
- In Fortran 90 and beyond you may need to use the module "`armpl_library`"

Arm Performance Libraries – linking

- In order to compile applications using BLAS, LAPACK and FFT routines from the Arm Performance libraries, four options are provided:
  - Serial and OpenMP builds
  - 32-bit and 64-bit integers
- These translate into four static binaries in `/opt/arm/armpl-*/lib/`
  - `libarmpl_lp64.a`
  - `libarmpl_lp64_mp.a`
  - `libarmpl_ilp64.a`
  - `libarmpl_ilp64_mp.a`
- Shared libraries (`libarmpl*.so`) are also provided
- Compile and link using, for example

```
gcc -O3 file.c -fopenmp -c file.o -I${ARMPL_DIR}/include
gcc -O3 file.o -fopenmp -o file -L${ARMPL_DIR}/lib -larmpl_mp
```

- libamath will be automatically linked in
  - Vector calls to maths functions only from Arm Compiler for Linux
    - GCC cannot automatically generate them at this time.

Spack virtual packages

- Spack provides the concept of a virtual package
  - A package depends upon an implementation, but doesn’t necessarily care whose
  - Used for MPI, BLAS, LAPACK, FFTW-API
- Applications depend on the virtual package
- Different packages can provide those implementations
  - Choice is made at installation time – user can override

App Using Virtual Package (SW4LITE)	Library Providing Implementation (ArmPL)	Library Providing Implementation (Open MPI)
depends_on(‘blas’)	provides(‘blas’)	provides(‘mpi’)
depends_on(‘lapack’)	provides(‘lapack’)	provides('mpi@:3.1', when='@2.0.0:')
depends_on(‘mpi’)	provides(‘fftw-api@3’)	

Spack specs for virtual packages

Ask Spack to fulfil with defaults: Results in ArmPL and OpenMPI

```
$ spack spec sw4lite%gcc
…
sw4lite@1.1%gcc@10.3.0~ckernel+openmp precision=double arch=linux-amzn2-graviton2
    ^armpl@21.0.0%gcc@10.3.0~ilp64~shared threads=none arch=linux-amzn2-graviton2
    ^openmpi@4.1.0%gcc@10.3.0~atomics~cuda~cxx~cxx_exceptions ……
```

Ask Spack to fulfil with ‘openblas’: Results in OpenBLAS and OpenMPI

```
$ spack spec sw4lite%gcc ^openblas
…
sw4lite@1.1%gcc@10.3.0~ckernel+openmp precision=double arch=linux-amzn2-graviton2
    ^openblas@0.3.15%gcc@10.3.0~bignuma~consistent_fpcsr~ilp64+locking+pic+shared
           threads=none arch=linux-amzn2-graviton2
    ^openmpi@4.1.0%gcc@10.3.0~atomics~cuda~cxx~cxx_exceptions ……
```

Who provides the implementations?

BLAS	LAPACK	FFTW-API
armpl	armpl	amdfftw
atlas	atlas	armpl
blis	cray-libsci	cray-fftw
cray-libsci	flexiblas	fftw
essl	fujitsu-ssl2	fujitsu-fftw
fujitsu-ssl2	intel-mkl	intel-mkl
intel-mkl	intel-oneapi-mkl	intel-oneapi-mkl
intel-oneapi-mkl	intel-parallel-studio	intel-parallel-studio
intel-parallel-studio	libflame	
netlib-lapack	netlib-lapack	
netlib-xblas	nvhpc	
nvhpc	openblas	
openblas	veclibfort	
veclibfort		

Note: Not all available on all platforms

How to use the virtual packages?

- Each implementation provides a ‘libs’
  - Listing which libraries to use, and their paths
  - Wrapped up as `'blas_libs'`, `'lapack_libs'` and `'fftw_libs'`

- Using package can then just reference the virtual package:

```
lapack_blas = spec['lapack'].libs + spec['blas'].libs
targets.append('EXTRA_LINK_FLAGS={0}'.format(lapack_blas.ld_flags))
```

- Results in: (for `sw4lite%gcc ^armpl`)

```
EXTRA_LINK_FLAGS= \
  -L/software/ACFL/21.0/armpl-21.0.0_AArch64_RHEL-7_gcc_aarch64-linux/lib \
  -L/lib64 \
  -L/software/gcc/10.3.0/lib64 \
  -larmpl -lamath -lastring -lm -lgfortran -lgfortran
```

How are variants handled? … It's complicated

- Variants
  - OpenMP support (`'threads=openmp'`), 64-bit int support (`'+ilp64'`)
  - Normally get passed through as a dependency

- Different packages handle variants differently
  - So, we need to specify them explicitly

```
depends_on('blas')
depends_on('lapack')
depends_on('armpl+ilp64 threads=openmp', when='^armpl')
depends_on('openblas+ilp64 threads=openmp', when='^openblas')
```

arm

Logging library calls: perf-libs-tools

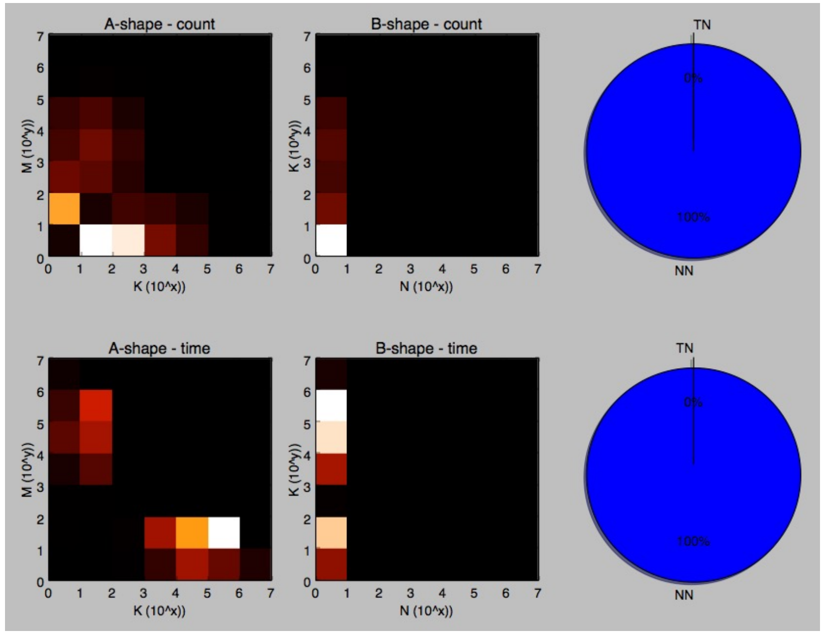
Logging Library : perf-libs-tools

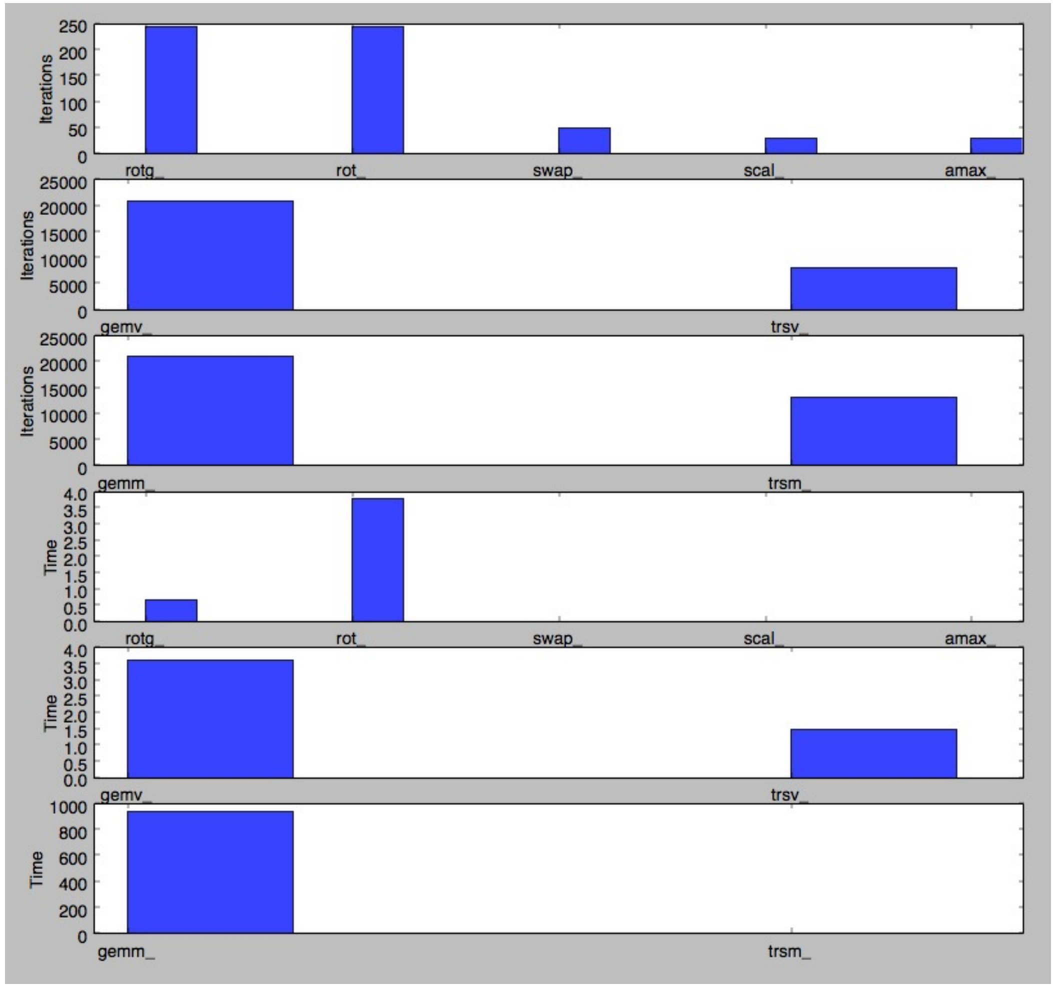
https://github.com/ARM-software/perf-libs-tools

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

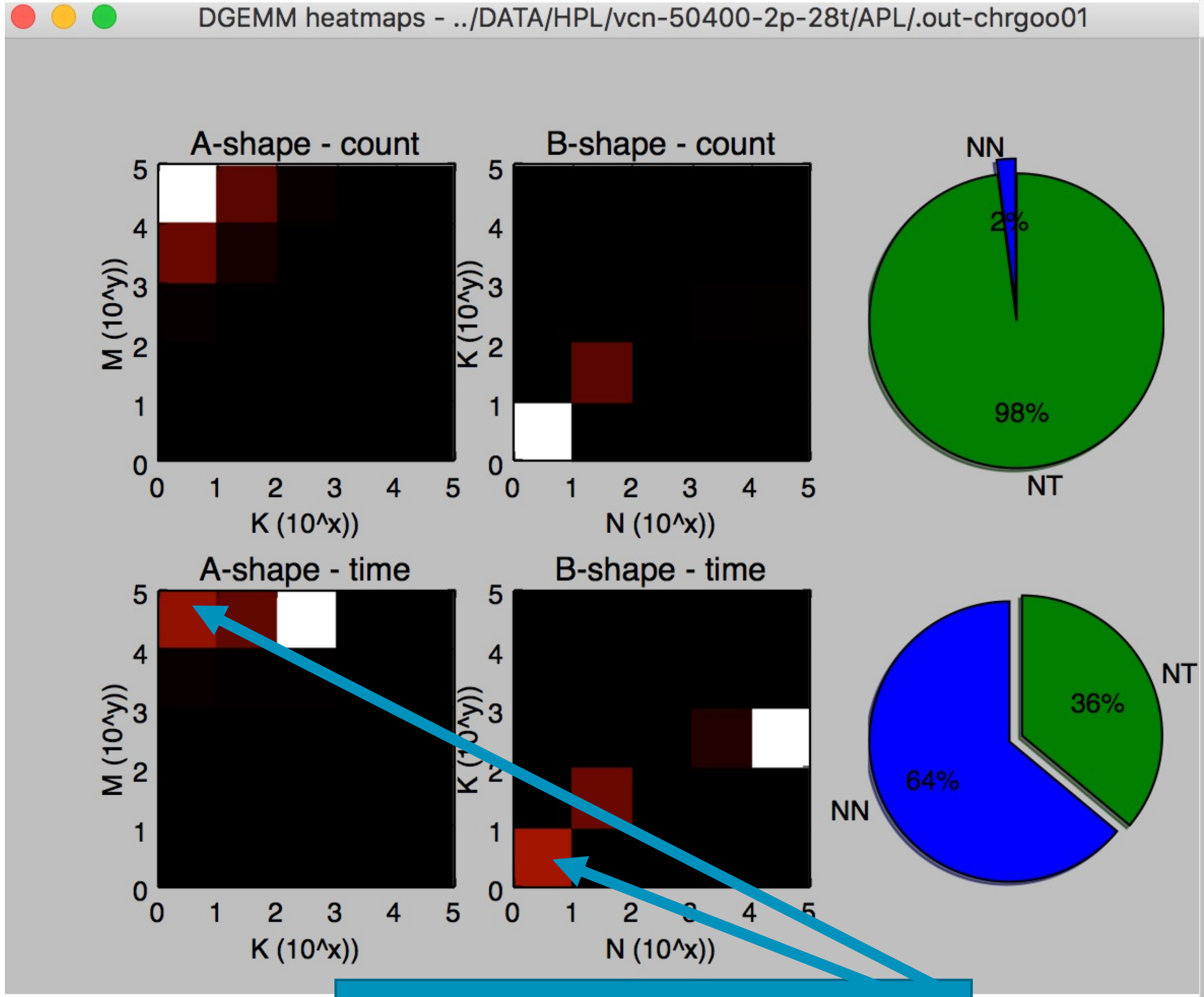
- Used in conjunction with Arm Performance Libraries can generate logging info to help profile applications for specific case breakdowns

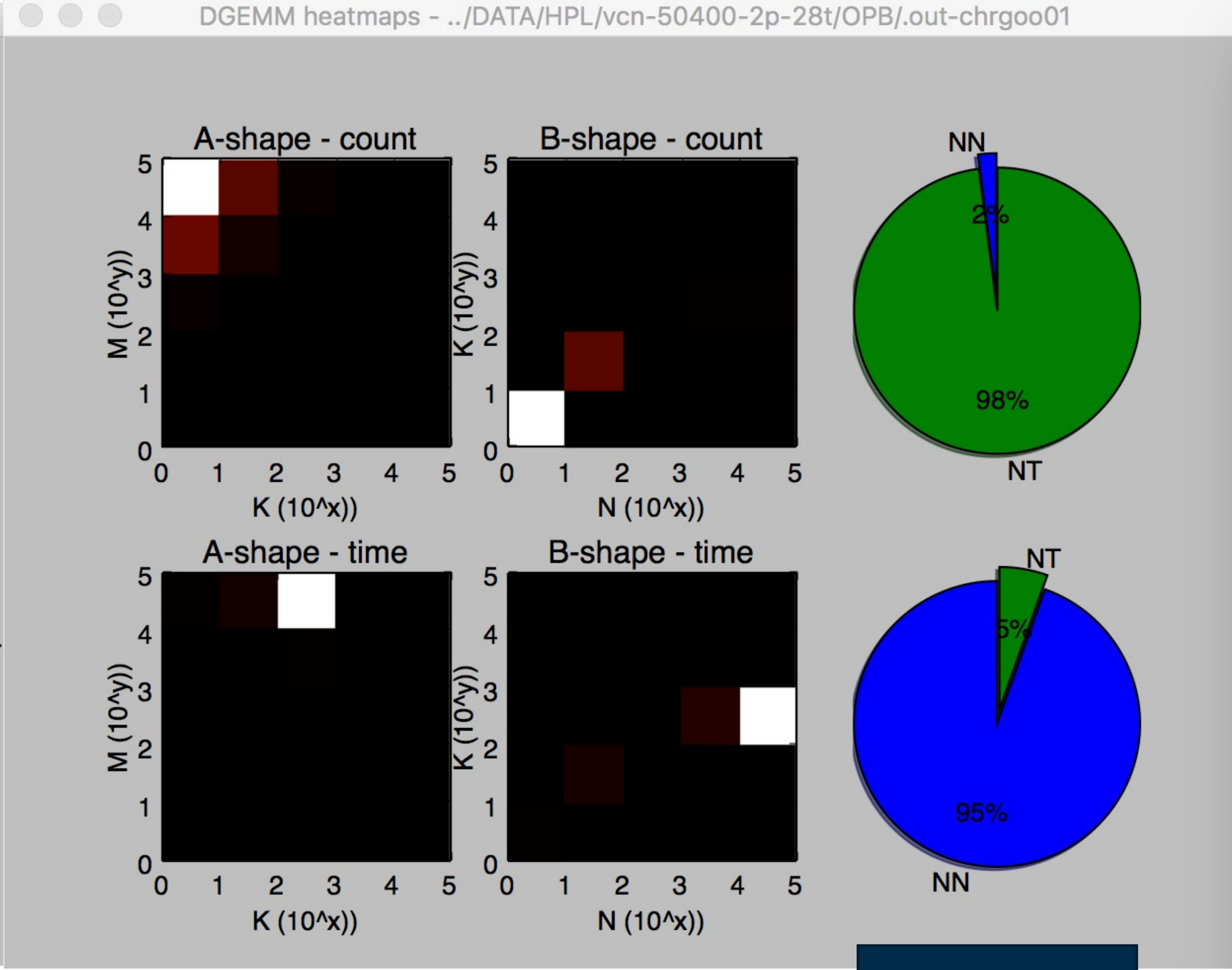
- Allows us to identify where time is spent in real applications.





Ages ago OpenBLAS did HPL better. Why?





Arm Performance Libraries

Huge time overhead for tiny cases. Now fixed!

OpenBLAS

perf-libs-tools – Usage guide

Building and collecting data is very straightforward:

```
# git clone https://github.com/ARM-software/perf-libs-tools
# cd perf-libs-tools
# make
# export LD_PRELOAD=$PWD/lib/libarmpl_summarylog.so
# ./application
<output>
Arm Performance Libraries output summary stored in
/tmp/armplsummary_16776.apl
```

Using perf-libs-tools then running the “process_summary.py” script gives lots of useful information like:

```
# ~/perf-libs-tools/tools/process_summary.py /tmp/armplsummary_*
Process full dataset for BLAS, LAPACK and FFT function usage.
…
BLAS level 1      : count   88218776    total time      48.6955
BLAS level 2      : count  334145452    total time    1672.3763
BLAS level 3      : count    7473286    total time     151.5514
LAPACK            : count     325027    total time       5.2572
FFT               : count   60446086    total time     481.7579
```

perf-libs-tools: Summary output per function

```
BLAS cases:
----------
BLAS level 1:
  dcopy_     cnt=   17440189  totTime=       24.0269
  dscal_     cnt=   67608016  totTime=       20.9392
  zcopy_     cnt=    2232038  totTime=        3.0867
  zdotc_     cnt=     175374  totTime=        0.2762
 zdscal_     cnt=     322549  totTime=        0.2116
  zscal_     cnt=     230973  totTime=        0.0696
  zaxpy_     cnt=      50006  totTime=        0.0201
 idamax_     cnt=      48546  totTime=        0.0188
  zswap_     cnt=      39103  totTime=        0.0116
 dznrm2_     cnt=      15485  totTime=        0.0097
   ddot_     cnt=      14859  totTime=        0.0064
 dzasum_     cnt=      15924  totTime=        0.0056
  daxpy_     cnt=      14859  totTime=        0.0055
  dasum_     cnt=       6216  totTime=        0.0031
  dnrm2_     cnt=       2072  totTime=        0.0021
 izamax_     cnt=       1067  totTime=        0.0017
  dswap_     cnt=       1500  totTime=        0.0006

BLAS level 2:
  dgemv_     cnt=  333911120  totTime=     1671.7741
  zgemv_     cnt=     137302  totTime=        0.4562
  ztrmv_     cnt=      27594  totTime=        0.0466
  ztrsv_     cnt=      23587  totTime=        0.0315
  zhemv_     cnt=      13104  totTime=        0.0292
  zher2_     cnt=      15081  totTime=        0.0279
  dtrsv_     cnt=      11840  totTime=        0.0072
  zgerc_     cnt=       5824  totTime=        0.0036
```

```
BLAS level 3:
  dgemm_     cnt=    7150124  totTime=       90.5322
  zgemm_     cnt=     201115  totTime=       59.9991
  ztrmm_     cnt=       3562  totTime=        0.7432
  ztrsm_     cnt=      42949  totTime=        0.1444
  zherk_     cnt=      43304  totTime=        0.0813
  dtrsm_     cnt=      31008  totTime=        0.0358
  zher2k_    cnt=        928  totTime=        0.0103
  zhemm_     cnt=        296  totTime=        0.0050

LAPACK cases:
----------
zhegv_     cnt=        843  totTime=        1.2420
zheev_     cnt=        843  totTime=        0.7431
zpotrf_    cnt=       1750  totTime=        0.4506
dgetrf_    cnt=       5920  totTime=        0.4397
zhegst_    cnt=        843  totTime=        0.3478
zhegs2_    cnt=        991  totTime=        0.3318
zsteqr_    cnt=        843  totTime=        0.2665
zhetrd_    cnt=        843  totTime=        0.2627
zungtr_    cnt=        843  totTime=        0.1859
ztrtri_    cnt=         64  totTime=        0.1818
zpocon_    cnt=        843  totTime=        0.1629
```

Case Study: (1) Report from a user

A user reported that the 18.4 libraries release was slower than FFTW for a key application

Compiler	BLAS	FFT	Runtime (s)
GCC 7.2	ArmPL 18.4	Cray FFTW	195
GCC 7.2	OpenBLAS 0.2	Cray FFTW	210
GCC 7.2	ArmPL 18.4	ArmPL 18.4	237
GCC 7.2	OpenBLAS 0.2	ArmPL 18.4	252

We were confused since we were confident that we could solve the sized FFT cases encountered quicker…

Case Study: (2) Using perf-libs-tools

The user was then was able to provide full perf-libs-tools for the whole solution run.

The FFT summaries looked like:

```
FFT cases:
----------
 FFTW calls:
    fftw_plan_many_dft len=  [54]   plan-cnt=      20092   plan-Time=     5.735552 exec-cnt=      24126    exec-Time=    20.338287
    fftw_plan_many_dft len=  [60]   plan-cnt=      20151   plan-Time=     5.380509 exec-cnt=      18043    exec-Time=    15.108546
    fftw_plan_many_dft len=  [80]   plan-cnt=      20151   plan-Time=     2.144119 exec-cnt=      18262    exec-Time=     4.805128
    fftw_plan_many_dft len= [120]   plan-cnt=        404   plan-Time=     0.038137 exec-cnt=        444    exec-Time=     0.081618
    fftw_plan_many_dft len= [160]   plan-cnt=        404   plan-Time=     0.058282 exec-cnt=        465    exec-Time=     0.115087
fftw_plan_many_dft_c2r len=  [54]   plan-cnt=         30   plan-Time=     0.001394 exec-cnt=          0    exec-Time=     0.000000
fftw_plan_many_dft_c2r len= [108]   plan-cnt=        188   plan-Time=     0.042434 exec-cnt=        139    exec-Time=     0.041167
fftw_plan_many_dft_r2c len=  [54]   plan-cnt=         29   plan-Time=     0.001645 exec-cnt=          0    exec-Time=     0.000000
fftw_plan_many_dft_r2c len= [108]   plan-cnt=        216   plan-Time=     0.043146 exec-cnt=        121    exec-Time=     0.027025
```

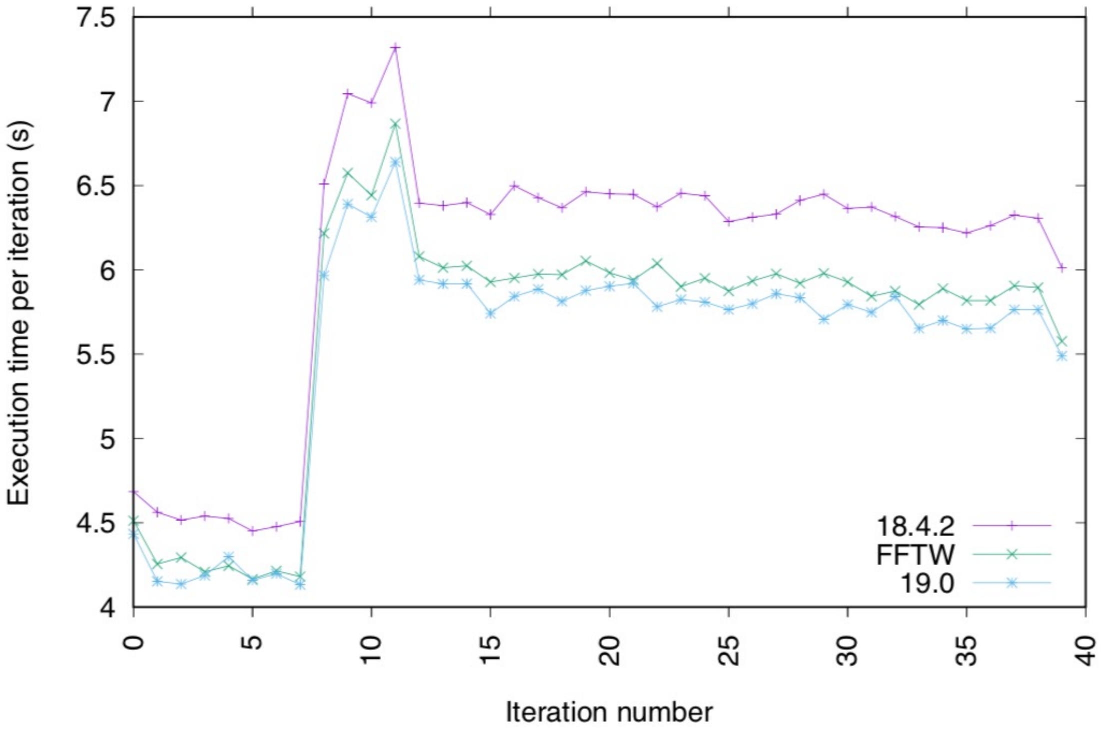
Library	Planning time	Execution time
ArmPL 18.4.0	13.66s	41.24s
FFTW	4.82s	49.72s

Case Study: (3) Fixing the issue

Back to the innards of the libraries

- It was clear that the code was creating one FFT plan per execution – a use-case we had never expected
- When we plan our FFTs various data-structures are set up.
- This includes allocating certain working arrays, even if we don't need to do auditioning as the case has been put into 'wisdom' previously
- It turned out we were unnecessarily zeroing an array which meant that if planning wasn't a rare event, there was an overhead
- Fixing this was very quick to implement and in the released 19.0 a week later

Final performance



Getting more information

We want you to be as successful as possible!

- If you have a question about anything to do with linking or running the various options do ask on the *'#help-maths-libraries'* channel on the A-HUG Cloud Hackathon Slack
  - We've got lots of experience of these packages. Don't suffer in silence!
- ArmPL:
  - Documentation:
    - https://developer.arm.com/documentation/101004/latest
  - Getting Started guide online
    - https://developer.arm.com/tools-and-software/server-and-hpc/downloads/arm-performance-libraries/get-started-with-armpl-free-version
- Perf-libs-tools:
    - https://github.com/ARM-software/perf-libs-tools
- OpenBLAS:
    - https://github.com/xianyi/OpenBLAS
- BLIS:
    - https://github.com/flame/blis
- FFTW
    - http://www.fftw.org/

Please do share your perf-libs-tools outputs with us. This will help prioritization of future work.