
arm
SVE Compilers
and Libraries

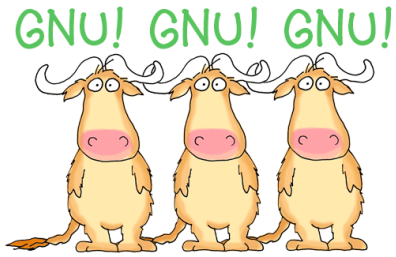
VLA Programming Approaches

Don’t panic!

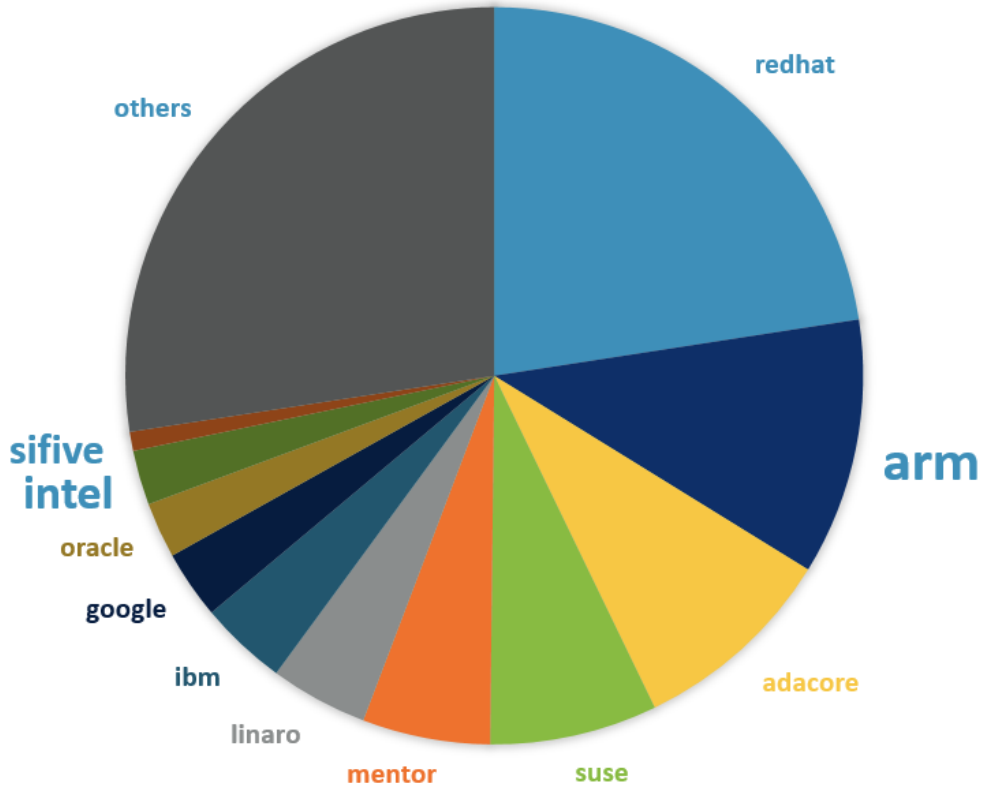
- Compilers:
  - Auto-vectorization: GCC, Arm Compiler for HPC, Cray, Fujitsu
  - Compiler directives, e.g. OpenMP
    - #pragma omp parallel for **simd**
    - #pragma vector always
- Libraries:
  - Arm Performance Library (ArmPL)
  - Cray LibSci
  - Fujitsu SSL II
- Intrinsics (ACLE):
  - Arm C Language Extensions for SVE
  - Arm Scalable Vector Extensions and Application to Machine Learning
- Assembly:
  - Full ISA Specification: The Scalable Vector Extension for Armv8-A

GNU compilers are a solid option

With Arm being significant contributor to upstream GNU projects



- GNU compilers are first class Arm compilers
  - Arm is one of the largest contributors to GCC
  - Focus on enablement and performance
  - Key for Arm to succeed in Cloud/Data center segment
- GNU toolchain ships with Arm Allinea Studio
  - Best effort support
  - Bug fixes and performance improvements in upcoming GNU releases



arm

GCC Optimization and Vectorization Reports

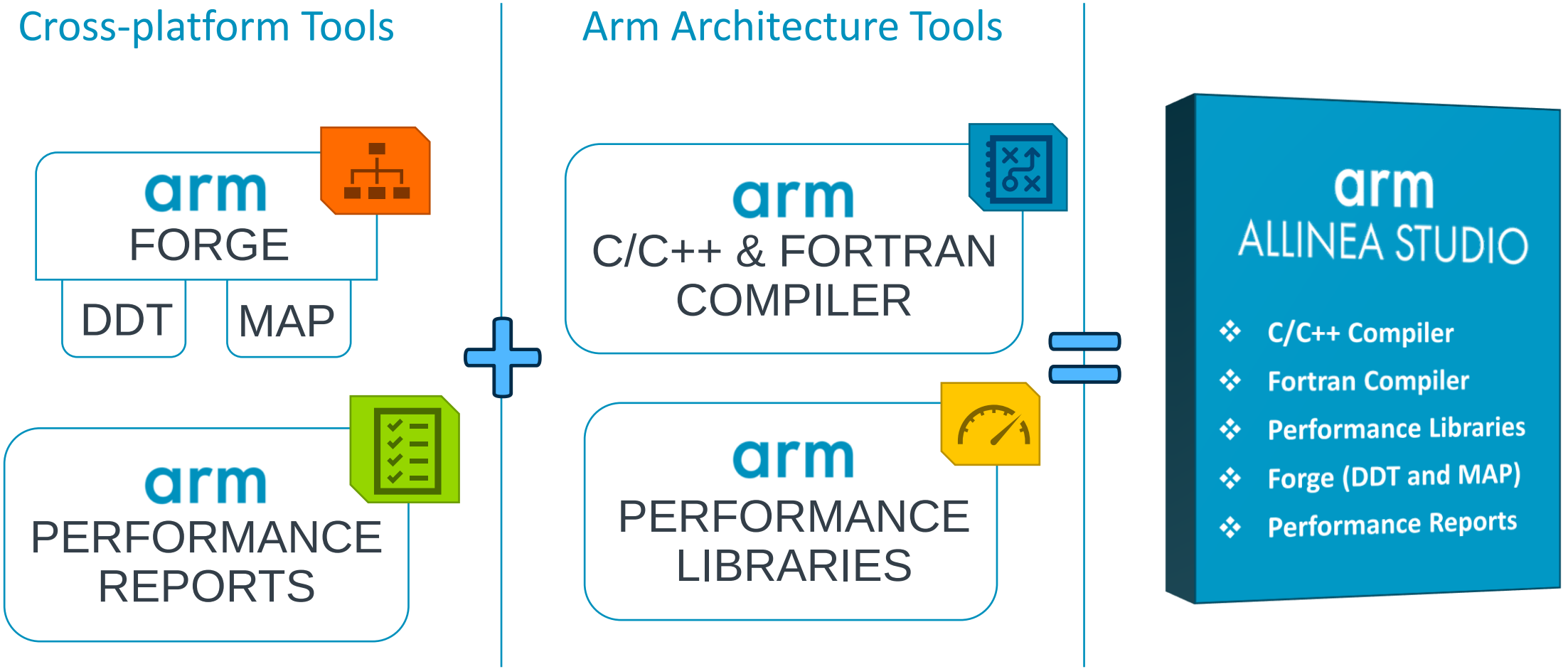
- https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html

-fopt-info	
-fopt-info-optimized	What was optimized
-fopt-info-missed	What was not optimized
-fopt-info-all	Everything
-fopt-info-all=file.out	Dump to file

```
[phirid01@co-c6-16-1 ~]$ gfortran -mcpu=thunderx2t99 -o test test.f90 -O3  -fopt-info-missed
test.f90:8:0: note: misalign = 0 bytes of ref c[_11]
test.f90:8:0: note: can't use a fully-masked loop because the target doesn't have the appropriate masked load or st
ore.
test.f90:8:0: note: cost model: the vector iteration cost = 4 divided by the scalar iteration cost = 1 is greater o
r equal to the vectorization factor = 2.
test.f90:8:0: note: not vectorized: vectorization not profitable.
test.f90:8:0: note: not vectorized: vector version will never be profitable.
test.f90:8:0: note: not vectorized: unsupported data-type
test.f90:8:0: note: can't determine vectorization factor.
test.f90:7:0: note: misalign = 0 bytes of ref b[_8]
```

Arm's solution for HPC application development and porting

Commercial tools for aarch64, x86_64, ppc64 and accelerators



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 platforms
- Optimal shared-memory parallelism via Arm's 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 for HPC: Front-end

Clang and Flang

C/C++

- Clang front-end
  - C11 including GNU11 extensions and C++14
  - Arm's 10-year roadmap for Clang is routinely reviewed and updated to respond to customers
- C11 with GNU11 extensions and C++14
- **Auto-vectorization for SVE and NEON**
- OpenMP 4.5

Fortran

- Flang front-end
  - Extended to support gfortran flags
- Fortran 2003 with some 2008
- **Auto-vectorization for SVE and NEON**
- OpenMP 3.1
- Transition to flang "F18" in progress
  - Extensible front-end written in C++17
  - Complete Fortran 2008 support
  - OpenMP 4.5 support

arm

Arm Compiler for HPC: Back-end

LLVM9

- Arm pulls all relevant cost models and optimizations into the downstream codebase
  - Arm's si-partners are committed to upstreaming cost models for future cores to LLVM
- Auto-vectorization via LLVM **vectorizers:**
  - Use cost models to drive decisions about what code blocks can and/or should be vectorized
  - Two different vectorizers from LLVM: Loop Vectorizer and SLP Vectorizer
- Loop Vectorizer support for SVE and NEON:
  - 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

Compile and link your application on Arm

- Modify the Makefile/installation scripts to ensure compilation for aarch64 happens
- Compile the code with the **Arm Compiler for HPC**
- Link the code with the **Arm Performance Libraries**

- Examples:
- `$> armclang -c –I/path/armpl/include example.c -o example.o`
- `$> armclang example.o -armpl -o example.exe -lm`

Arm Compiler for HPC	GNU Compiler
armclang	gcc
armclang++	g++
armflang	gfortran

arm

Targeting SVE with both Arm compiler and GNU (8+)

- Compilation targets a specific architecture based on an architecture revision
  - -mcpu=native -march=armv8.1-a+lse+sve
    - Learn more: https://community.arm.com/.../compiler-flags-across-architectures-march-mtune-and-mcpu
- -march=armv8-a
  - Target V8-a
  - Will generate NEON instructions
  - No SVE
- -march=armv8-a+sve
  - Will add SVE instruction generations
- Check the assembly (-S)
  - ***armclang++ -S -o code.s -Ofast -g -march=armv8-a+sve code.cpp***
  - ***g++ -S -o code.s -Ofast -g -march=armv8-a+sve code.cpp***

Optimization Remarks for Improving Vectorization

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.

Optimization remarks example

https://developer.arm.com/products/software-development-tools/hpc/arm-fortran-compiler/optimization-remarks

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
[-Rpass-analysis=asm-printer]
   void foo(int K) {    ^
```

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: Vectorization Control

OpenMP and clang directives are supported by the Arm Compiler for HPC

C/C++	Fortran	Description
#pragma ivdep	!DIR$ IVDEP	Ignore potential memory dependencies and vectorize the loop.
#pragma vector always	!DIR$ VECTOR ALWAYS	Forces the compiler to vectorize a loop irrespective of any potential performance implications.
#pragma novector	!DIR$ NO VECTOR	Disables vectorization of the loop.

Clang compiler directives for C/C++	Description
#pragma clang loop vectorize(assume_safety)	Assume there are no aliasing issues in a loop.
#pragma clang loop unroll_count(_value_)	Force a scalar loop to unroll by a given factor.
#pragma clang loop interleave_count(_value_)	Force a vectorized loop to interleave by a factor

Arm Compiler Vectorization Reports

`-fsave-optimization-record` & `arm-opt-report file.opt.yaml`

Vectorized

4x 32-bit lanes

1-way interleaving

Fully unrolled

All 3 instances of `foo()` were inlined

```
$ armclang -O3 or.c -c -o or.o -fsave-optimization-record -march=armv8-a+sve
$ arm-opt-report or.opt.yaml
< or.c
 1          | void bar();
 2          | void foo() { bar(); }
 3          |
 4          | void Test(int *res, int *c, int *d, int *p, int n) {
 5          |   int i;
 6          |
 7          | #pragma clang loop vectorize(assume_safety)
 8     V4,1 |   for (i = 0; i < 1600; i++) {
 9          |     res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
10          |   }
11          |
12     U16  |   for (i = 0; i < 16; i++) {
13          |     res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
14          |   }
15          |
16 I        |   foo();
17          |
18          |   foo(); bar(); foo();
   I        |   ^
   I        |               ^
19          | }
```


arm

arm

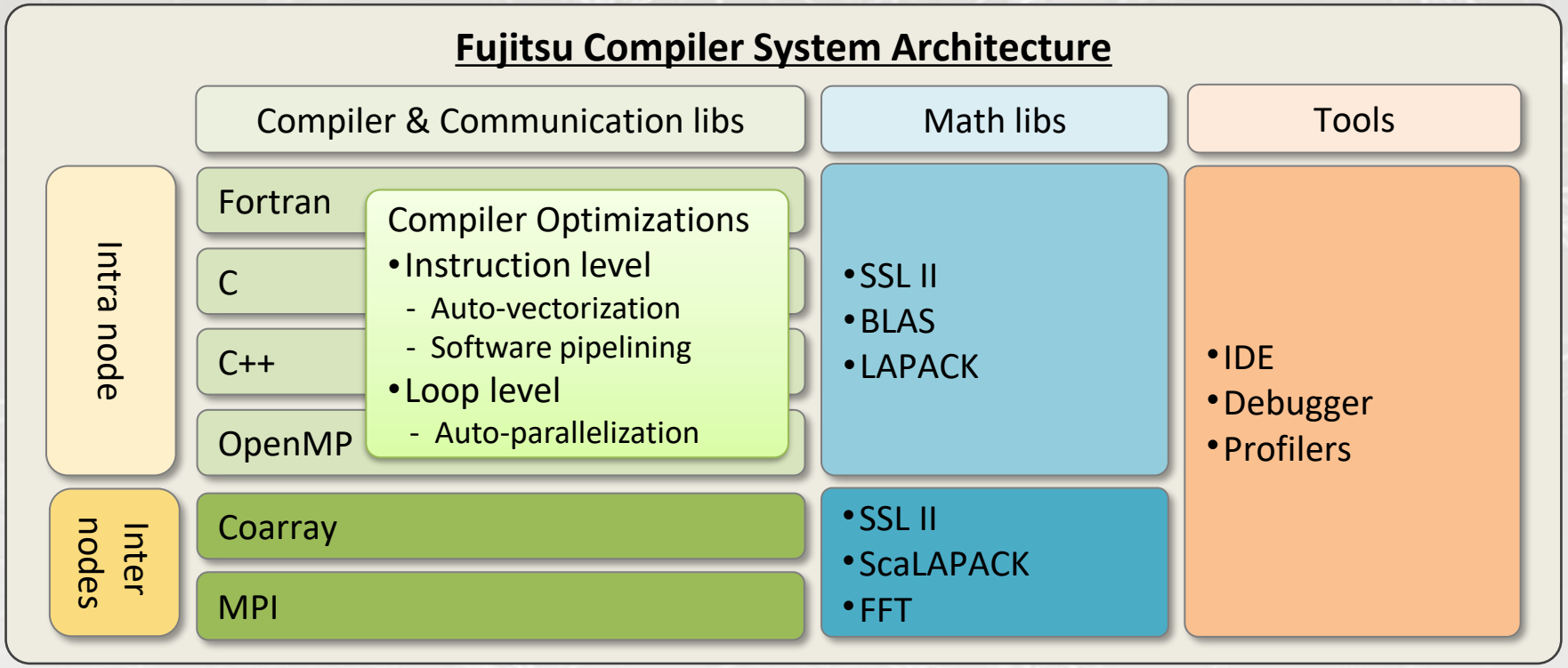
Fujitsu Compiler

FUJITSU

shaping tomorrow with you

Overview of Fujitsu compiler for A64FX systems

Fujitsu Compiler System Architecture



- Develops a variety of programming tools for various programming models
- Designs and develops Software exploiting Hardware performance

A64FX Features and Compiler Approaches

FUJITSU

- A64FX CPU Inherits features of K computer and PRIMEHPC FX100
  - Usability including options and programming models are inherited
- Compiler targeting 512-bit wide vectorization to promotes optimization, such as constant folding, by fixing vector length
  - Vectorization as VLA(vector-length-agnostic) and NEON (Advanced SIMD) is also supported

	Functions & Architecture	Fugaku	FX100	K computer
Processor	Base ISA + SIMD Extensions	ARMv8-A+SVE	SPARC V9 +HPAC-ACE2	SPARC V9 +HPC-ACE
	SIMD width [bits]	512	256	128
	Float Packed SIMD	✔ Enhanced	✔	-
	FMA	✔	✔	✔
	Reciprocal approx. inst. Math. acceleration inst.	✔	✔	✔
	Inter-core hardware barrier	✔	✔	✔
	Sector cache	✔ Enhanced	✔	✔
	Hardware “prefetch” assist	✔ Enhanced	✔	✔

Fujitsu Compiler: Language Standard Support

FUJITSU

Languages	Specification	Support Level
C	C11 (ISO/IEC 9899:2011)	fully supported
C++	C++14 (ISO/IEC 14882:2014) C++17C++17 (ISO/IEC 14882:2017)	fully supported partially supported
Fortran	Fortran 2008 (ISO/IEC 1539-1:2010) Fortran 2018 (ISO/IEC 1539-1:2018)	fully supported partially supported
OpenMP	OpenMP 4.0 (released in July 2013) OpenMP 4.5 (released in Nov. 2015) OpenMP 5.0 (released in Nov. 2018)	fully supported partially supported partially supported

Promotes object-oriented programming and accelerates high performance by supporting latest language standards

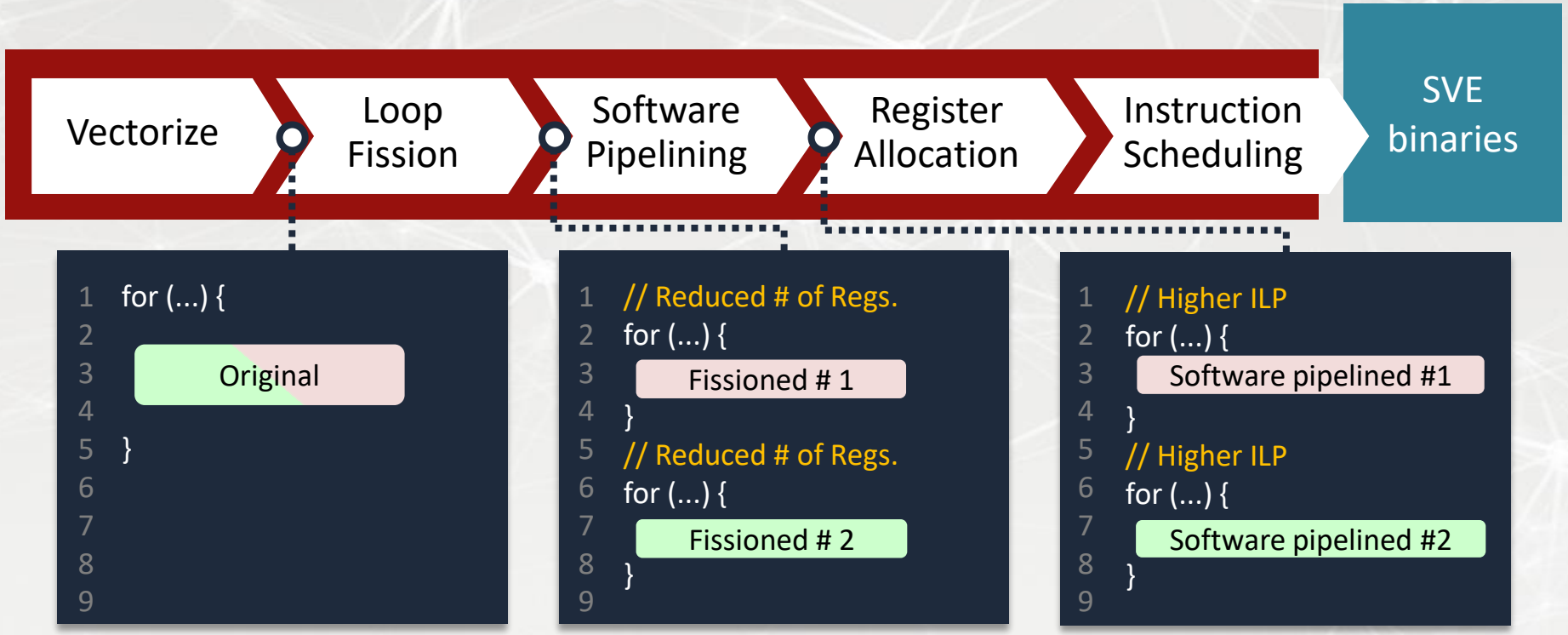
Important Optimizations to accelerate performance

FUJITSU

- Vectorization
  - Automatic vectorization is enhanced to utilize SVE
  - OpenMP SIMD directives and ACLE are availabe
- Software-pipelining
  - Improves instruction-level parallelism of loops
- Loop fission
  - Reduces necessary registers in order to promote software-pipelining

Fujitsu Compiler Optimization Flow

- 1. Vectorizes loops with SVE instructions
- 2. Loop Fission reduces required resources if needed
- 3. Software-pipelining is performed
- 4. Register allocated with optimizations
- 5. Pre/Post-RA instruction scheduling is performed



Fujitsu Compiler Commands

FUJITSU

- Cross-compiler, which works on x86 server

Tips: cross compiler commands have **px** suffix, which means cross-platform.

Language	Command
Fortran	`frtpx [option list] [file list]`
C	`fccpx [options list] [file list]`
C++	`FCCpx [options list] [file list]`

- Own-compiler, which works on Arm server

Language	Command
Fortran	`frt [option list] [file list]`
C	`fcc [options list] [file list]`
C++	`FCC [options list] [file list]`

Note: own-compiler is also called native-compiler or self-compiler.

Cross-compiler and own-compiler have the same ability.
Differences are their command names and where they work.

Recommended Options

FUJITSU

- -Kfast option is recommend for higher performance
  - Turns on the following options internally
  - Some Options cause side-effects in execution result such as precision

Option	Feature
-O3	Compile at highest optimization level 3.
-Kdalign	Assume alignment on a double-word boundary.
-Keval	Apply optimization to change the method of mathematical evaluation
-Kfp_contract	Optimize by using FMA arithmetic instructions.
-Kfp_relaxed	Execute reciprocal approximation operations.
-Kfz	**[New for Armv8]** Enable flush-to-zero mode to treat denormal numbers as zero
-Kilfunc	Inline expand intrinsic math functions into approximate instructions
-Kmfunc	Apply multi-operation functionalization to promote vectorizing
-Komitfp	Omit the frame pointer register for a procedure call.
-Ksimd_packed_promotion	**[New for SVE]** Promote packed simd for SVE instructions to optimize address calculation
-Klib	[C/C++ only] Optimize with recognizing C standard libraries functions as built-in
-Krdconv	[C/C++ only] Optimize with assuming that 4-byte int loop variants does not overflow
-x-	[C/C++ only] Inline expand user-defined functions

Other Notable Options for Performance

FUJITSU

Options to promote or control optimizations

Option	Feature
-Kpreex	Evaluate codes which is invariant through loops before entering loops. This may cause a execution time error such as segmentation fault.
-Ksimd=2	Vectorizes loops which contains IF-constructs with predication.
-Kocl	Enable Optimization Control Lines (OCL), FUJITSU-specific directives to control optimization.

Parallelization options

Option	Feature
-Kparallel	Apply automatic parallelization.
-Kopenmp	Enable OpenMP directives
-Kopenmp_simd	Enable OpenMP simd directives

under development

Useful options to know applied optimizations

Option	Feature
-Nlst=t	Output Compilation Optimization information in list file (*.lst)
-Koptmsg=2	Output optimization messages

arm

Cray Compiler

Hewlett Packard
Enterprise

HPE Cray Programming Environment for ARM

Compiler for Apollo 80 and Legacy Cray XC Systems

- Supports native compilation (no cross-compiling) for Marvell TX2 and Fujitsu A64FX ARM processors
- Offers compiler feedback through loopmark: -hlist=a (cce-sve) or –fsave-loopmark (cce clang)
- cce-sve
  - Fortran and C/C++ compiler generates ARM SVE code
  - C/C++ compiler based on Cray classic compiler (EDG front-end)
    - Not as strong C++ support
- cce
  - Added to ARM platforms in Sept to provide stronger compiler for C++ code
  - Fortran and C/C++ compiler generates ARM Neon vector code
  - C/C++ compiler based on new clang compiler (LLVM)
    - ARM SVE code generation coming in fall of 2021
    - Will move to SVE code generation for Fortran and C/C++
- Usage guidance
  - Choose compiler based on dominant/most important code (C++ strength vs Fortran SVE strength)
  - Cannot mix use between two compiling environments

HPE Cray Compiling Environment (CCE)

- Fortran compiler
  - Proprietary front-end and optimizer; HPE-modified LLVM
  - Fortran 2018 support (including coarray teams)

- C and C++ compiler
  - HPE-modified closed-source build of Clang+LLVM complier
  - C11 and C++17 support
  - UPC support

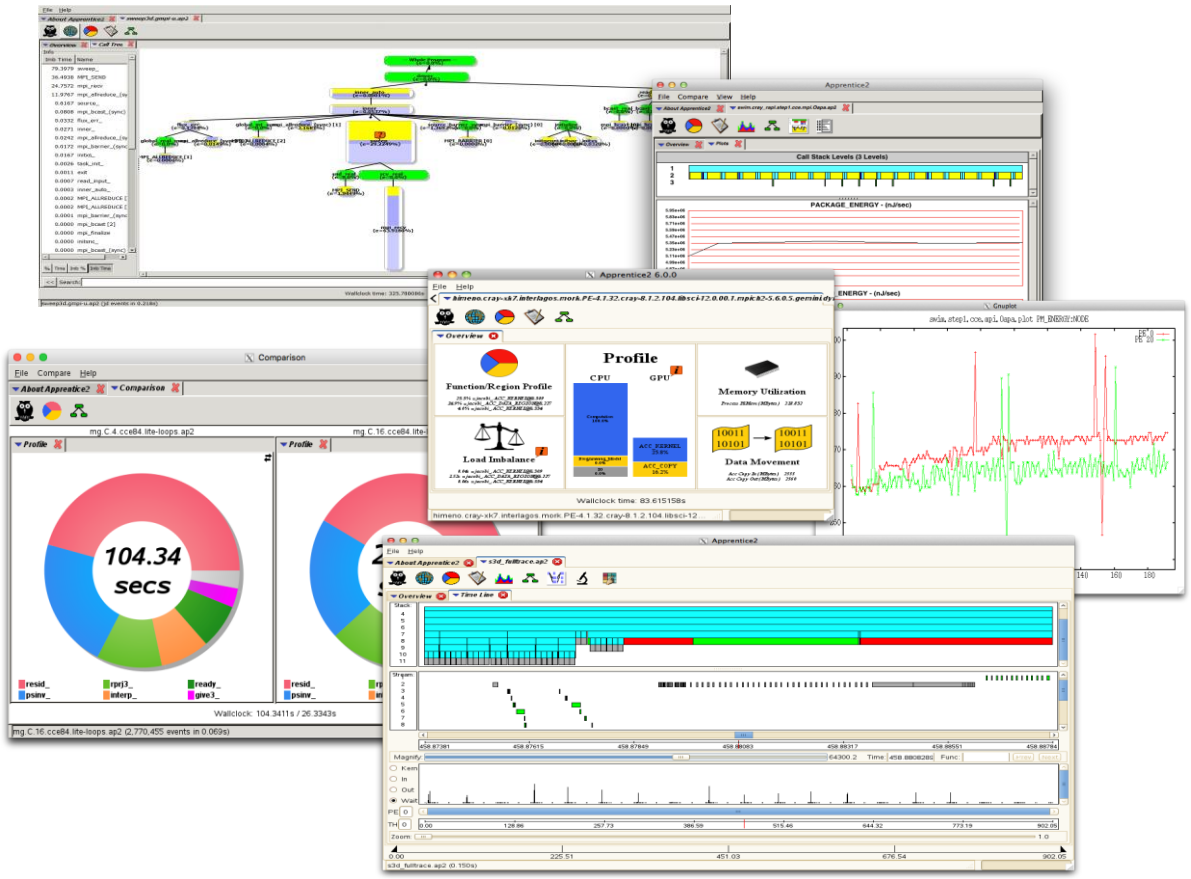
- PGAS support is functional (not performant across IB) on Apollo 80

- OpenMP support
  - Partial OpenMP 5.0
  - Full OpenMP 5.0/5.1 planned over next year

Scientific and Math Libraries

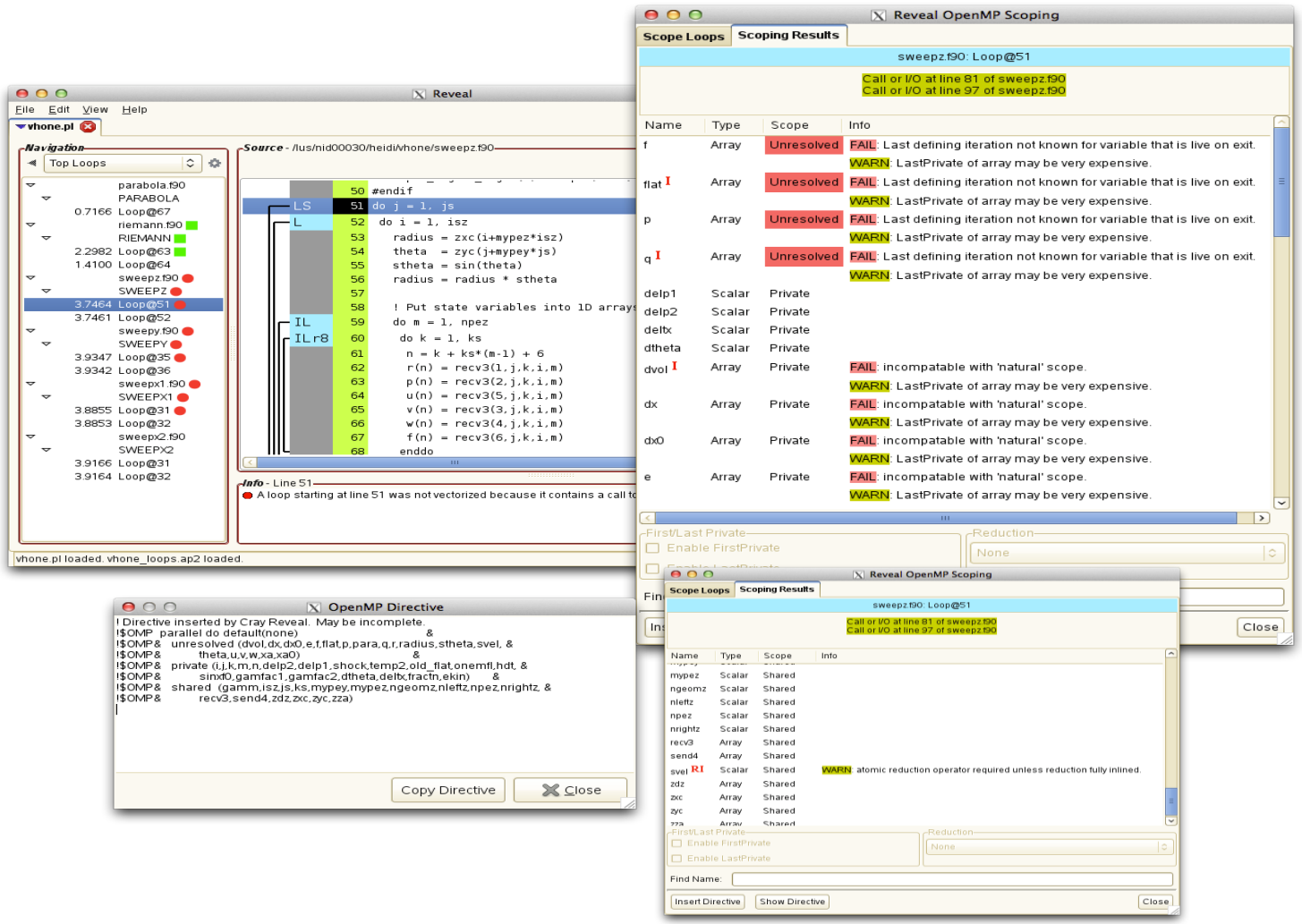
Node performance	Highly tuned BLAS etc. at the low-level
Network performance	• Optimize for network performance • Overlap between communication and computation • Use the best available low-level mechanism • Use adaptive parallel algorithms
Highly adaptive software	Using auto-tuning and adaptation selects optimal algorithms at runtime
Productivity features	Simpler interfaces into complex software

Performance Analysis Tools



- Reduce the time investment associated with porting and tuning applications on Cray systems
- Analyze whole-program behavior across many nodes to identify critical performance bottlenecks within a program
- Use simple and/or advanced interfaces for a wealth of capability that targets analyzing the largest HPC jobs

Code Parallelization Assistant



- Reduce effort associated with adding OpenMP to MPI programs
- Works in conjunction with our compiler and performance tools
- Identify work-intensive loops to parallelize, perform dependence analysis, scope variables and generate OpenMP directives

HPE Cray PE Summary for ARM

- Cray PE has been able to extract good performance on Fujitsu A64FX by applying architecture-specific optimizations
  - Able to use gather/scatter and predication from SVE
  - Can achieve 95% peak on dgemm with our scientific libraries

- Two choices available for CCE
  - CCE compiler that produces Neon instructions has shown to perform well on A64FX and has more robust C++ support
  - CCE SVE compiler performs well for Fortran codes
  - Choose CCE flavor based on dominant/most important code (C++ strength vs Fortran SVE strength)

- Libsci targets Marvell TX2 and Fujitsu A64FX ARM processors
  - Can be used with either cce-sve or cce compilers
  - Dgemm uses of A64FX's sector cache starting with cray-libsci version 20.10.1

- Performance tools and debugger help identify issues when porting or profiling codes that target ARM


arm
Hands On:
Compilers

01_Compiler

See README.md for details

- Focus not on understanding the problem, but how to use various SVE toolchains
- Naïve to optimized performance
- Matrix-matrix multiplication in C/C++
  - Initialize random data
  - Perform multiply
  - Report wall clock time
- See **README.md** in each directory for additional details
- Use "make all COMPILER=*COMPILER_NAME*" to compare compiler performance
  - Type make COMPILER=help to see all supported compilers

01_Compiler/01_Naive

See README.md for details

GCC 9.3

```
-----------------------------------------------
./mm_gnu_def.exe 256 256 256
Set up of matrices took: 0.011 seconds
Performing multiply
Naive multiply took: 0.441 seconds
-----------------------------------------------
./mm_gnu_opt.exe 256 256 256
Set up of matrices took: 0.010 seconds
Performing multiply
Naive multiply took: 0.117 seconds
-----------------------------------------------
./mm_gnu_opt_novec.exe 256 256 256
Set up of matrices took: 0.010 seconds
Performing multiply
Naive multiply took: 0.118 seconds
-----------------------------------------------
```

ACfL 20.3

```
-----------------------------------------------
./mm_arm_def.exe 256 256 256
Set up of matrices took: 0.012 seconds
Performing multiply
Naive multiply took: 0.328 seconds
-----------------------------------------------
./mm_arm_opt.exe 256 256 256
Set up of matrices took: 0.010 seconds
Performing multiply
Naive multiply took: 0.123 seconds
-----------------------------------------------
./mm_arm_opt_novec.exe 256 256 256
Set up of matrices took: 0.010 seconds
Performing multiply
Naive multiply took: 0.123 seconds
-----------------------------------------------
```


arm

01_Compiler/02_Block_Trans

See README.md for details

ACfL 20.3

./mm_blk_trans_arm_def.exe 1024 1024 1024 128

Set up of matrices took: 0.190 seconds

Performing multiply

Transpose multiply took: 10.456 seconds

./mm_blk_trans_arm_opt.exe 1024 1024 1024 128

Set up of matrices took: 0.154 seconds

Performing multiply

Transpose multiply took: 5.227 seconds

./mm_blk_trans_arm_opt_novec.exe 1024 1024 1024 128

Set up of matrices took: 0.150 seconds

Performing multiply

Transpose multiply took: 5.211 seconds

Hot loop does not vectorize

```
mm_blk_trans.cpp:43:21: remark: loop not vectorized [-Rpass-missed=sve-loop-vectorize]
                    for(k =kk; k < min(m, kk + blockSize); ++k){
                    ^
mm_blk_trans.cpp:42:17: remark: loop not vectorized [-Rpass-missed=sve-loop-vectorize]
                for(j= jj; j < min(l, jj + blockSize); j+= 2){
                ^
mm_blk_trans.cpp:41:13: remark: loop not vectorized [-Rpass-missed=sve-loop-vectorize]
            for(i= 0; i < n; i+= 2){
            ^
mm_blk_trans.cpp:43:21: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
                    for(k =kk; k < min(m, kk + blockSize); ++k){
                    ^
mm_blk_trans.cpp:42:17: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
                for(j= jj; j < min(l, jj + blockSize); j+= 2){
                ^
mm_blk_trans.cpp:41:13: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
            for(i= 0; i < n; i+= 2){
            ^
```


arm

01_Compiler/03_Vectorize

See README.md for details

ACfL 20.3

```
-----------------------------------------------
./mm_vec_arm_def.exe 1024 1024 1024 1024
Set up of matrices took: 0.190 seconds
Performing multiply
Multiply took: 20.899 seconds
-----------------------------------------------
./mm_vec_arm_opt.exe 1024 1024 1024 1024
Set up of matrices took: 0.156 seconds
Performing multiply
Multiply took: 0.742 seconds
-----------------------------------------------
./mm_vec_arm_opt_novec.exe 1024 1024 1024 1024
Set up of matrices took: 0.165 seconds
Performing multiply
Multiply took: 5.508 seconds
-----------------------------------------------
```

Hot loop vectorizes

mm_vec.cpp:37:21: remark: vectorized loop (vectorization width: 2, interleaved count: 1, scalable: true) [-Rpass=sve-loop-vectorize]

```
            for(k=kk; k < kk+blockSize; ++k){
            ^
```

mm_vec.cpp:90:5: remark: vectorized loop (vectorization width: 2, interleaved count: 1, scalable: true) [-Rpass=sve-loop-vectorize]

```
  for(i= 0; i < n; ++i){
  ^
```

mm_vec.cpp:95:5: remark: vectorized loop (vectorization width: 2, interleaved count: 1, scalable: true) [-Rpass=sve-loop-vectorize]

```
  for(i= 0; i < l; ++i){
  ^
```


arm

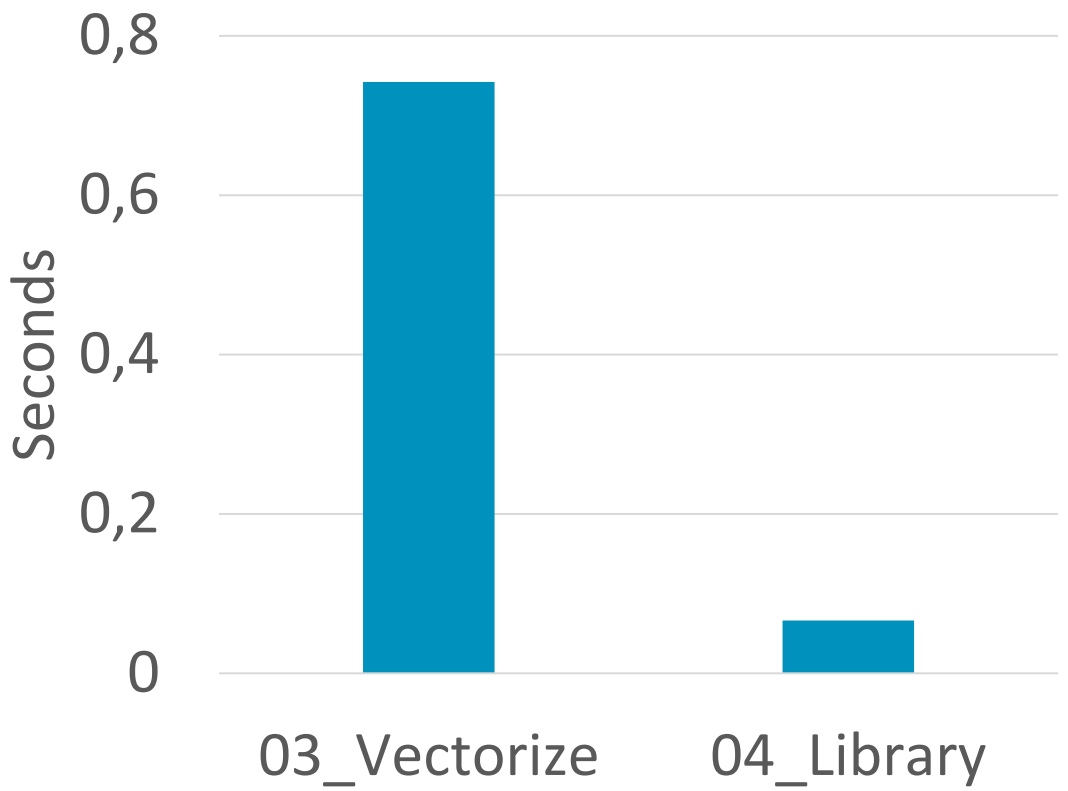
01_Compiler/04_Library

See README.md for details

Arm Performance Library (ArmPL) 20.3

```
-----------------------------------------------
./mm_lib_arm.exe 1024 1024 1024
Set up of matrices took: 0.154 seconds
Using DGEMM routine from ArmPL library
ArmPL library took: 0.066 seconds
-----------------------------------------------
```

Compared to handwritten loop



arm

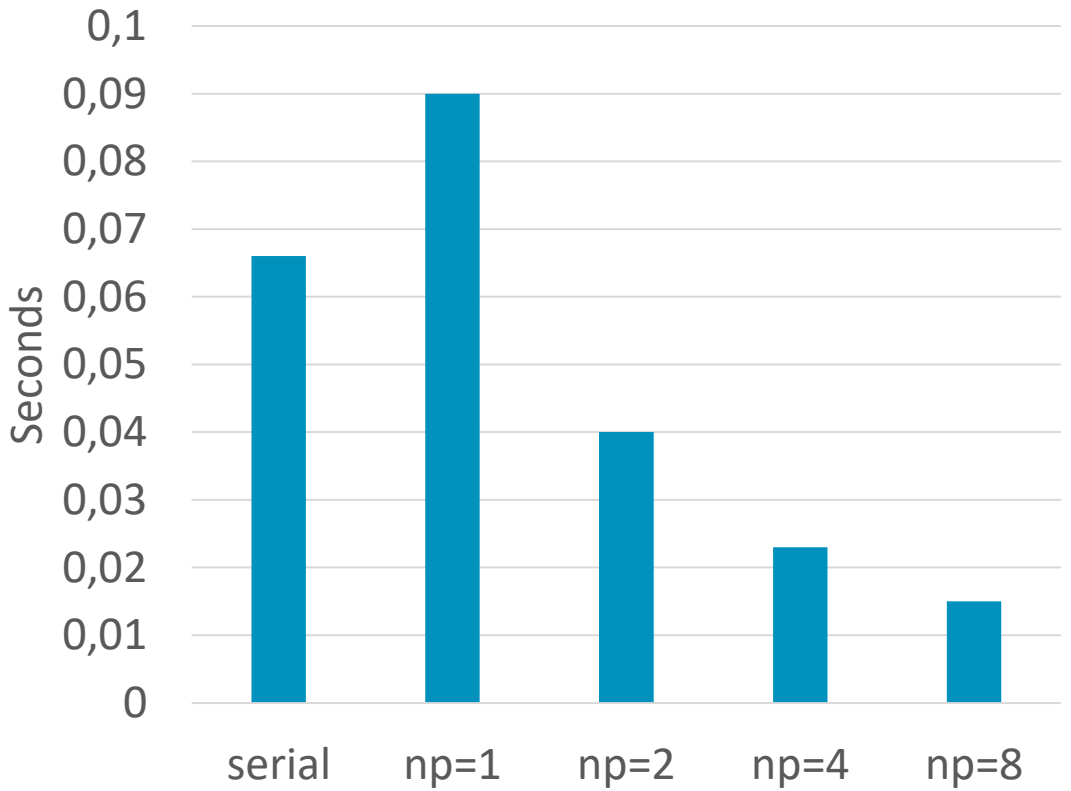
01_Compiler/05_OpenMP

See README.md for details

Arm Performance Library (ArmPL) 20.3

```
-----------------------------------------------
./mm_arm_ser.exe 1024 1024 1024 128
ArmPL library took: 0.066 seconds
OMP_NUM_THREADS=1 ./mm_arm_omp.exe 1024 1024 1024 128
ArmPL library took: 0.090 seconds
OMP_NUM_THREADS=2 ./mm_arm_omp.exe 1024 1024 1024 128
ArmPL library took: 0.040 seconds
OMP_NUM_THREADS=4 ./mm_arm_omp.exe 1024 1024 1024 128
ArmPL library took: 0.023 seconds
OMP_NUM_THREADS=8 ./mm_arm_omp.exe 1024 1024 1024 128
ArmPL library took: 0.015 seconds
-----------------------------------------------
```

Performance scales with thread count



arm

Resources

- Porting and Optimizing HPC Applications for ARM
  - https://developer.arm.com/documentation/101725/0200
- Arm Compiler Auto-vectorization examples
  - https://developer.arm.com/documentation/100891/0612/coding-considerations/auto-vectorization-examples
- Arm SVE Instruction Reference (detailed descriptions of each SVE instruction)
  - https://developer.arm.com/docs/ddi0596/i/a64-sve-instructions-alphabetic-order
- SVE programming examples
  - https://developer.arm.com/documentation/dai0548/latest
- Arm Fortran Compiler Reference
  - https://developer.arm.com/documentation/101380/2030
- Arm Performance Libraries Reference
  - https://developer.arm.com/documentation/101004/2030

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
תודה