Arm and SVE

International Workshop on High-Performance Computing and Programming on Quantum Chemistry and Physics 2020 (HPCPQCP2020)

Dr. Olly Perks (olly.perks@arm.com)

15th January 2020

Agenda

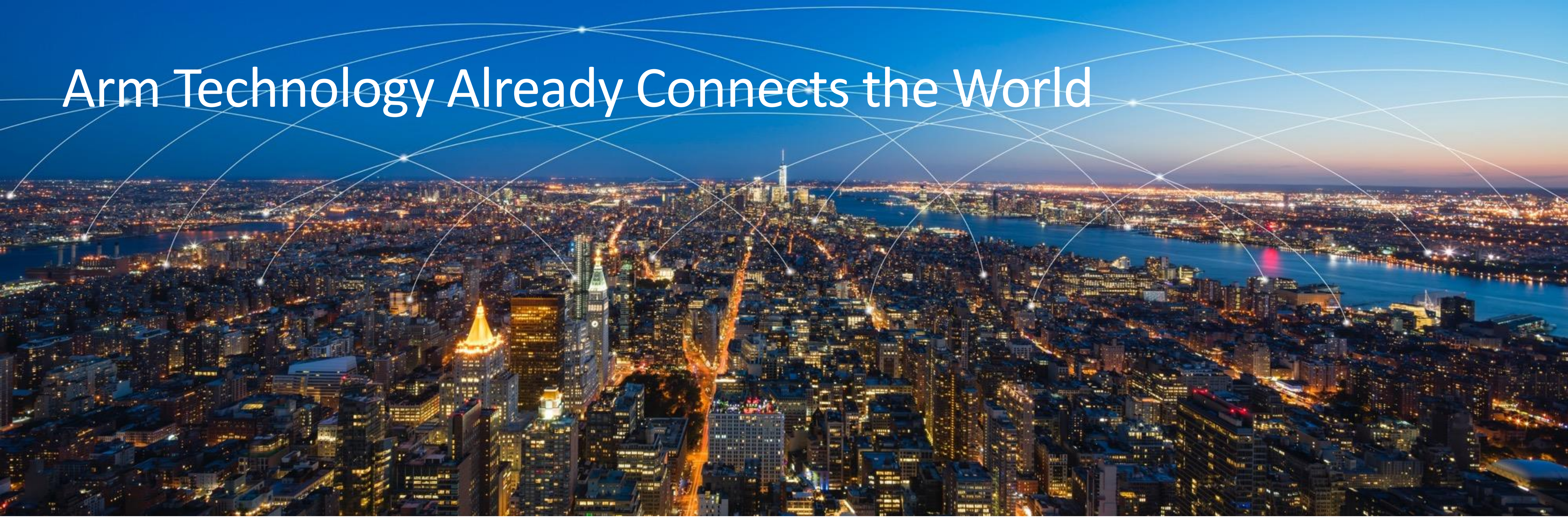
- Part 1: Getting to SVE (45 Mins)
  - A quick introduction of Arm in HPC / Scientific Computing
  - Existing Arm vector units / instructions: NEON
  - The SVE specification
- Part 2: Going Forward with SVE (45 mins)
  - Using SVE for real applications
  - Next generation vector instructions: Beyond SVE

© 2019 Arm Limited

Part 1: Getting to SVE

Recap:
Arm and Arm in HPC

Arm Technology Already Connects the World



Arm is ubiquitous

21 billion chips sold by partners in 2017 alone

Mobile/Embedded/IoT/ Automotive/Server/GPUs

Partnership is key

We design IP, not manufacture chips

Partners build products for their target markets

Choice is good

One size is not always the best fit for all

HPC is a great fit for co-design and collaboration

© 2019 Arm Limited

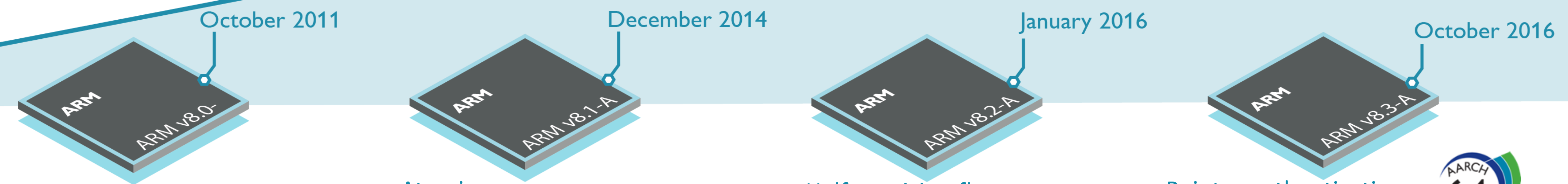
Armv8-A Architecture Evolution

RISC architecture

- Only have 32 bits available for encoding all instructions
- Supports the development of efficient implementations

64-bit capable since 2012

- Known as AArch64 (or AArch32 when run in a 32-bit mode)
- 128-bit vector unit (aka NEON Advanced SIMD)



October 2011

- AArch64 execution state
- A64 instruction set

December 2014

- Atomic memory ops
- Type2 hypervisor support

January 2016

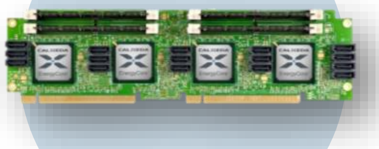
- Half-precision float
- RAS support
- Statistical profiling

October 2016

- Pointer authentication
- Nested virtualization
- Complex float

© 2019 Arm Limited

History of Arm in HPC: A Busy Decade

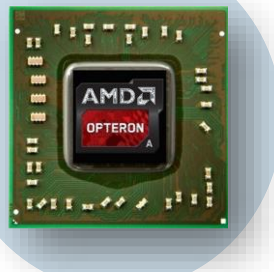


2011 Calxada

- 32-bit ARrmv7-A – Cortex A9

2011-2015 Mont-Blanc 1

- 32-bit Armv7-A
- Cortex A15
- First Arm HPC system

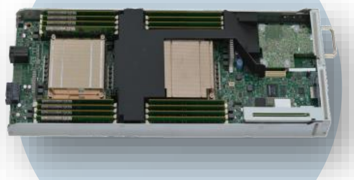


2014 AMD Opteron A1100

- 64-bit Armv8-A
- Cortex A57
- 4-8 Cores

2015 Cavium ThunderX

- 64-bit Armv8-A
- 48 Cores

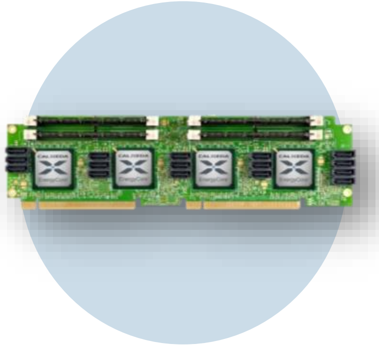


2017 (Cavium) Marvell ThunderX 2

- 64-bit Armv8-A
- 32 Cores

© 2019 Arm Limited

History of Arm in HPC: A Busy Decade

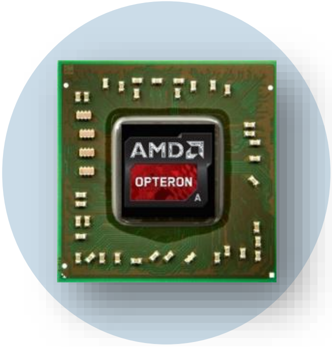


2011 Calxada

- 32-bit ARrmv7-A – Cortex A9

2011-2015
Mont-Blanc 1

- 32-bit Armv7-A
- Cortex A15
- First Arm HPC system

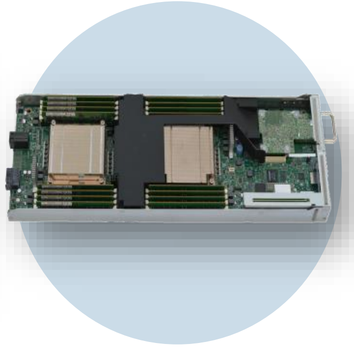


2014
AMD Opteron A1100

- 64-bit Armv8-A
- Cortex A57
- 4-8 Cores

2015
Cavium ThunderX

- 64-bit Armv8-A
- 48 Cores



2017
(Cavium) Marvell ThunderX 2

- 64-bit Armv8-A
- 32 Cores

2019
Fujitsu A64FX

- First Arm chip with SVE vectorisation
- 48 Cores

© 2019 Arm Limited

Why Arm?

Especially for Infrastructure / HPC / Scientific Computing / ML?

Hardware

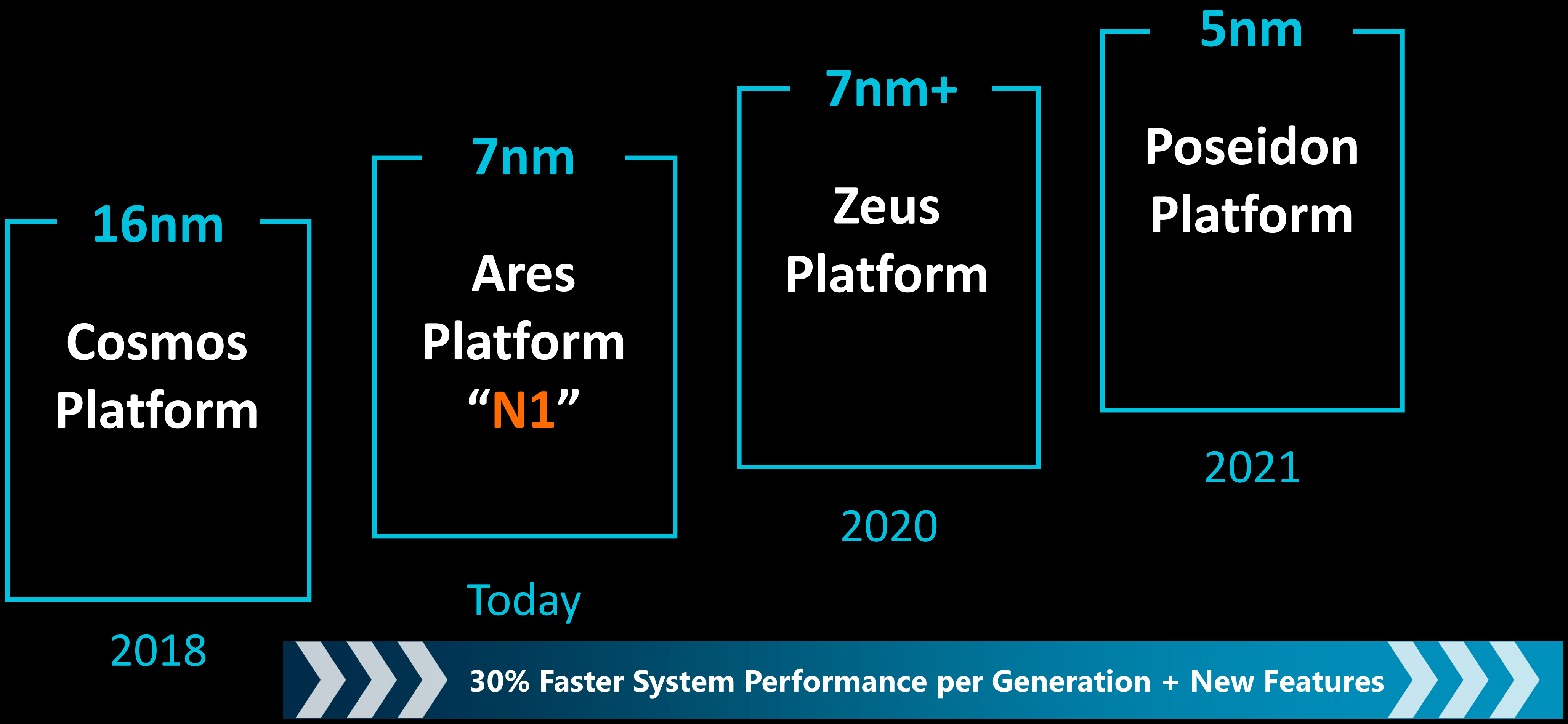
- Flexibility: Allow vendors to differentiate
  - Speed and cost of development
- Provide different licensing
  - Core - Reference design (A53/A72/N1)
  - Architecture - Design your own (TX2, A64FX)
- Other hardware components
  - NoCs, GPUs, memory controllers
  - “Building blocks” design
- Architecture validation for correctness

Software

- All based on the same instruction set
  - Commonality between hardware
  - Reuse of software
- Comprehensive software ecosystem
  - Operating systems, compilers, libraries, tools
  - Not just vendor - third party too
- Large community
  - Everything from Android to HPC

© 2019 Arm Limited

Each generation brings faster performance and new infrastructure specific features



16nm

Cosmos Platform

2018

7nm

Ares Platform "N1"

Today

7nm+

Zeus Platform

2020

5nm

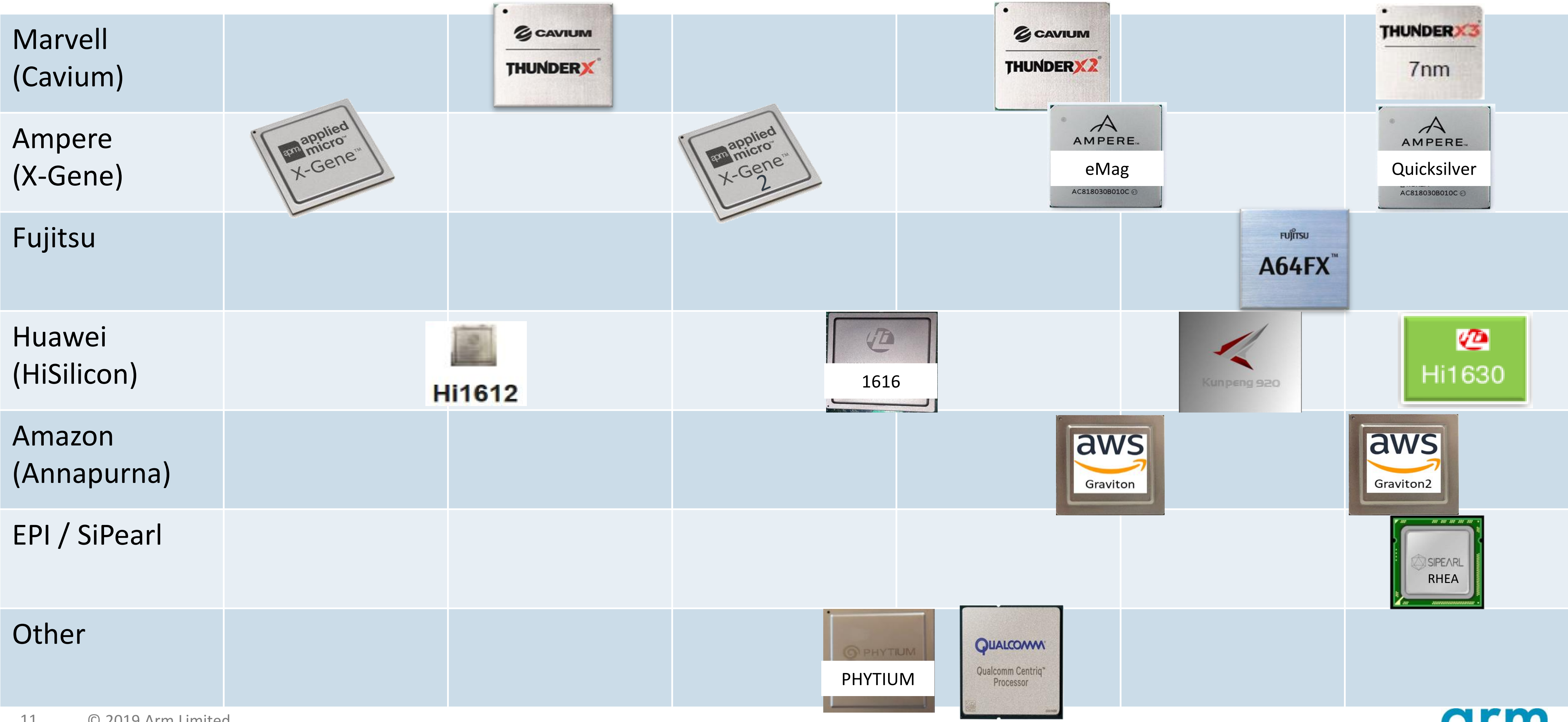
Poseidon Platform

2021

30% Faster System Performance per Generation + New Features

© 2018 Arm Limited

Variation in the Processor Market



Marvell (Cavium)		CAVIUM THUNDERX		CAVIUM THUNDERX2		THUNDERX3 7nm
Ampere (X-Gene)	applied micro X-Gene		applied micro X-Gene 2	AMPERE eMag AC818030B010C		AMPERE Quicksilver AC818030B010C
Fujitsu					FUJITSU A64FX	
Huawei (HiSilicon)		Hi1612	1616		Kunpeng 920	Hi1630
Amazon (Annapurna)				aws Graviton		aws Graviton2
EPI / SiPearl						SIPEARL RHEA
Other			PHYTIUM PHYTIUM	QUALCOMM Qualcomm Centriq Processor		

© 2019 Arm Limited

Arm Processor Top Trumps

Marvell: ThunderX2

Core Count:	32
Clock Speed:	2.5 GHz
Memory Bandwidth:	~130 GB/s
Energy Consumption:	~200 w
Vector Units:	128-bit NEON

Huawei: Kunpeng 920

Core Count:	64
Clock Speed:	2.6 GHz
Memory Bandwidth:	~130 GB/s
Energy Consumption:	~180 w
Vector Units:	128-bit NEON

Fujitsu: A64FX

Core Count:	48
Clock Speed:	2 GHz
Memory Bandwidth:	~1 TB/s
Energy Consumption:	~150 w
Vector Units:	512-bit SVE

© 2019 Arm Limited

HPC Deployments

- More Arm based CPUs are being adopted
  - Lots of large scale deployments
- Different OEMs
  - Cray, HPE, Atos-Bull, Fujitsu, Huawei, E4
- EU Deployments
  - Isambard: Cray 10k TX2 cores
  - GENCIE: Atos 18k TX2 cores
  - Catalyst 3 systems: HPE 4k TX2 core
  - Fugaku Prototype: Fujitsu 36.8K A64FX cores
  - Future Isambard 2: Cray A64FX
  - Future Deucalion: Cray A64FX

THE WORLD'S FIRST PETASCALE ARM SUPERCOMPUTER
Sandia National Laboratories
ASTRA
per aspera ad astra
VANGUARD
U.S. DEPARTMENT OF ENERGY
NNSA
Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

>5k ThunderX2 CPUs

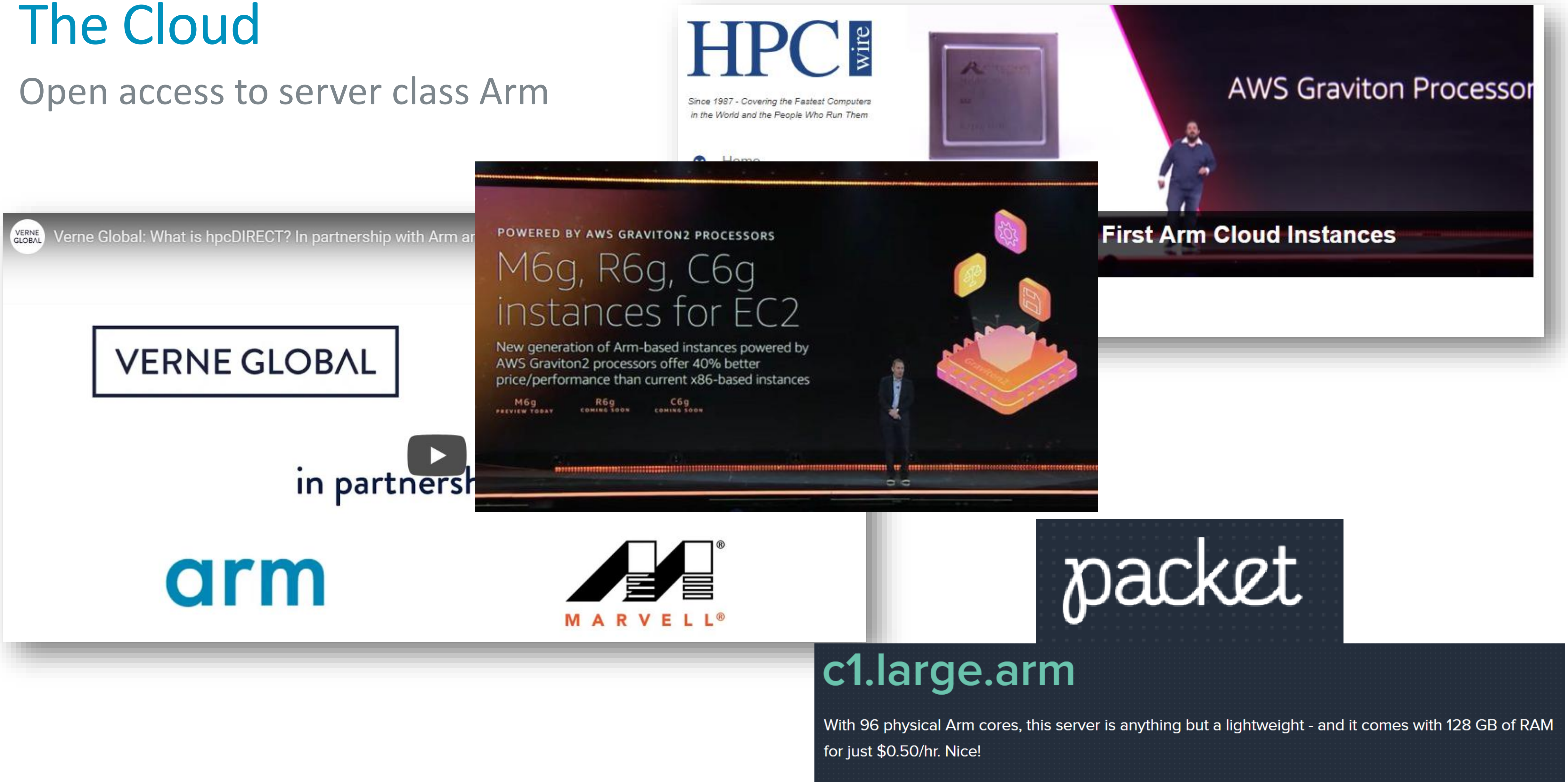
2k Kunpeng 920 CPUs + 8k AI accelerators

150k+ Fujitsu A64FX CPUs

© 2019 Arm Limited

The Cloud

Open access to server class Arm



HPC wire

Since 1987 - Covering the Fastest Computers in the World and the People Who Run Them

AWS Graviton Processor

First Arm Cloud Instances

Verne Global: What is hpcDIRECT? In partnership with Arm ar

VERNE GLOBAL

in partnersh

arm

MARVELL

POWERED BY AWS GRAVITON2 PROCESSORS

M6g, R6g, C6g instances for EC2

New generation of Arm-based instances powered by AWS Graviton2 processors offer 40% better price/performance than current x86-based instances

M6g PREVIEW TODAY

R6g COMING SOON

C6g COMING SOON

packet

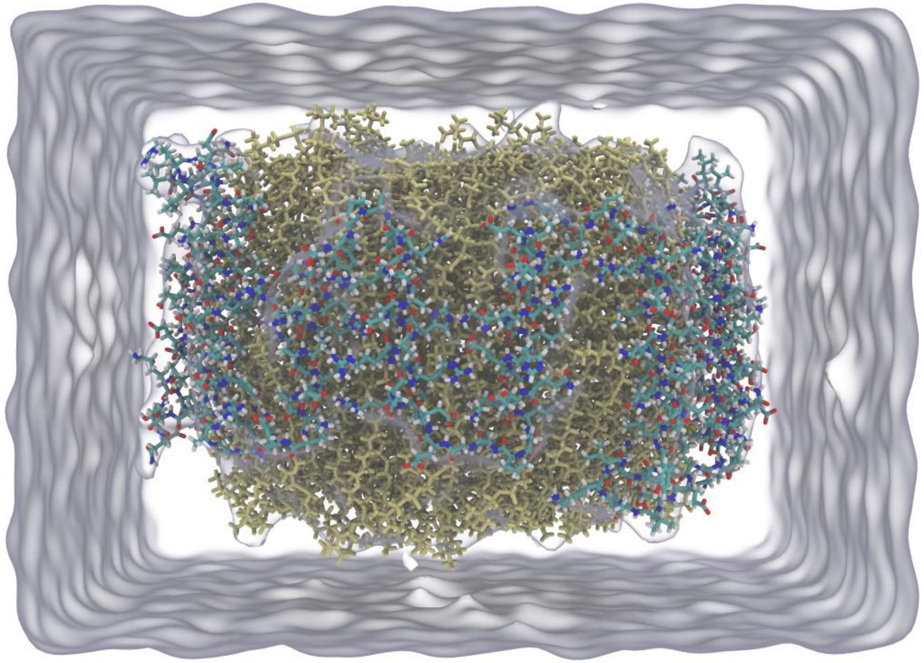
c1.large.arm

With 96 physical Arm cores, this server is anything but a lightweight - and it comes with 128 GB of RAM for just $0.50/hr. Nice!

© 2019 Arm Limited

NAMD 3 Alpha, Tesla V100 Performance, ApoA1 92k Atoms

Hardware platform		ns/day,	Speedup
Intel Xeon 8168	+ 1x Tesla V100	102.1,	1.0x
IBM Power9	+ 1x Tesla V100	99.7,	0.97x
Cavium ThunderX2	+ 1x Tesla V100	98.2,	0.96x



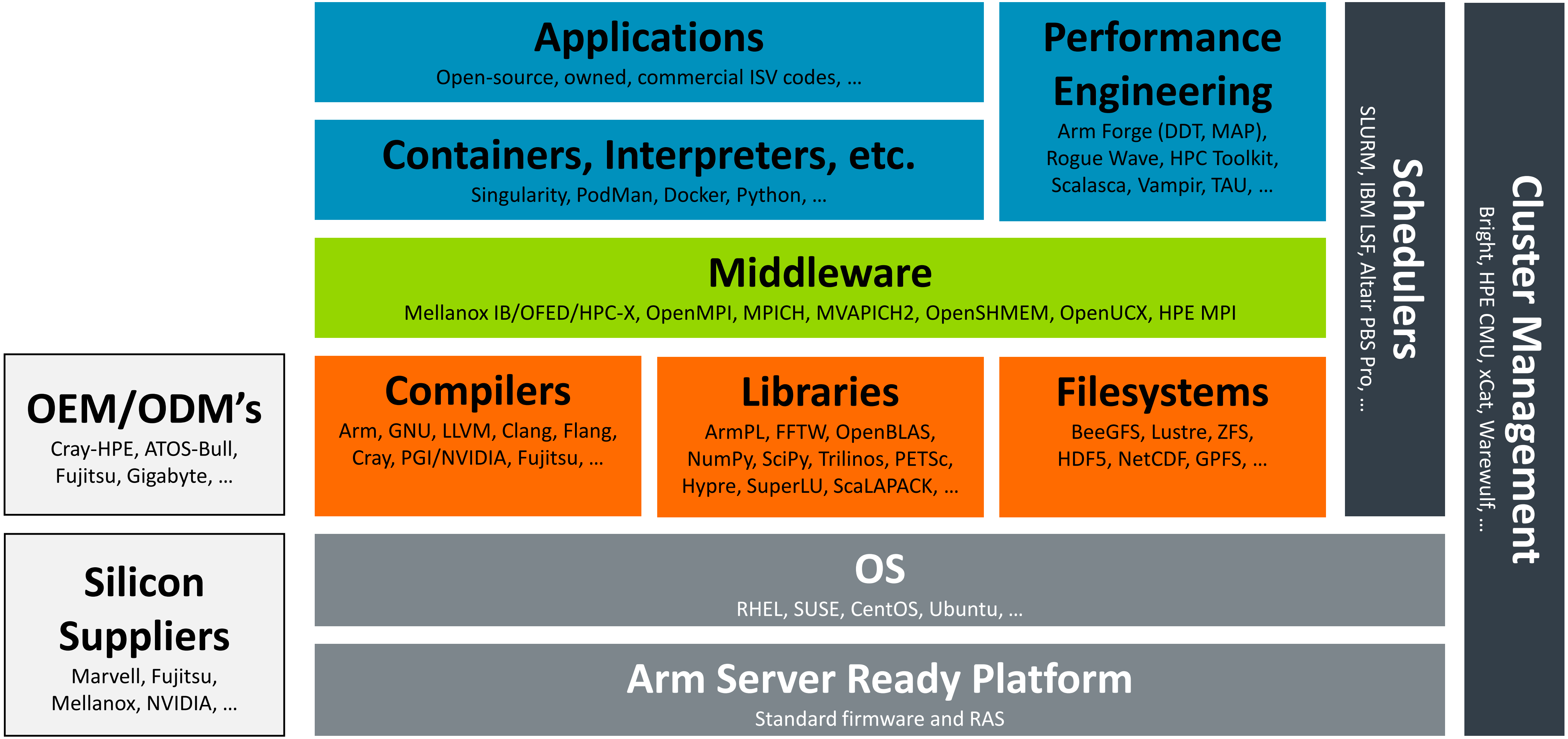
NAMD 3 Alpha Note:

This measured the new NAMD 3 Alpha in single-node mode, yielding greatest overall GPU acceleration for small to moderate sized atomic structure simulations.

ARM+NVIDIA
Preliminary Results

© 2019 Arm Limited

Not Just Hardware



Applications
Open-source, owned, commercial ISV codes, …

Containers, Interpreters, etc.
Singularity, PodMan, Docker, Python, …

Performance Engineering
Arm Forge (DDT, MAP), Rogue Wave, HPC Toolkit, Scalasca, Vampir, TAU, …

Schedulers
SLURM, IBM LSF, Altair PBS Pro, …

Cluster Management
Bright, HPE CMU, xCat, Warewulf, …

Middleware
Mellanox IB/OFED/HPC-X, OpenMPI, MPICH, MVAPICH2, OpenSHMEM, OpenUCX, HPE MPI

OEM/ODM's
Cray-HPE, ATOS-Bull, Fujitsu, Gigabyte, …

Compilers
Arm, GNU, LLVM, Clang, Flang, Cray, PGI/NVIDIA, Fujitsu, …

Libraries
ArmPL, FFTW, OpenBLAS, NumPy, SciPy, Trilinos, PETSc, Hypre, SuperLU, ScaLAPACK, …

Filesystems
BeeGFS, Lustre, ZFS, HDF5, NetCDF, GPFS, …

Silicon Suppliers
Marvell, Fujitsu, Mellanox, NVIDIA, …

OS
RHEL, SUSE, CentOS, Ubuntu, …

Arm Server Ready Platform
Standard firmware and RAS

© 2019 Arm Limited

A Rich and Growing Application Ecosystem

🟧 GROMACS	🟧 LAMMPS	🟨 CESM2	🟩 MrBayes	🟩 Bowtie
🟧 NAMD	🟧 AMBER	▨ Paraview	🟧 SIESTA	🟨 UM
🟨 WRF	🟧 Quantum ESPRESSO	🟧 VASP	🟧 MILC	🟧 GEANT4
🟦 OpenFOAM	🟧 GAMESS	▨ VisIt	🟧 DL-Poly	🟨 NEMO
🟩 BLAST	🟧 NWCHEM	🟧 Abinit	🟩 BWA	🟧 QMCPACK

🟧 *Chem/Phys*	🟨 *Weather*	🟦 *CFD*	▨ *Visualization*	🟩 *Genomics*

LS-DYNA

ANSYS® FLUENT®

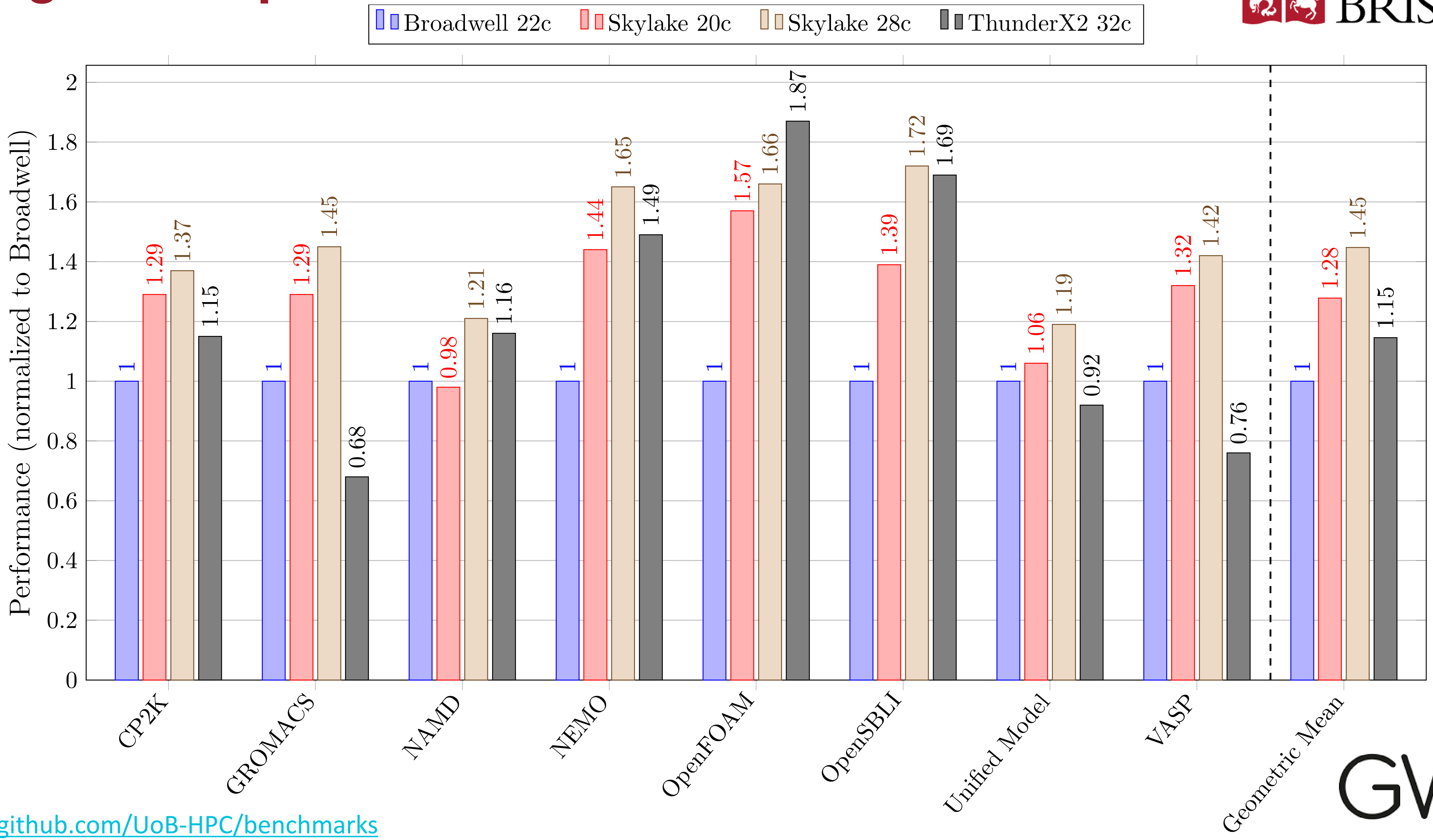
Altair

cādence

Community driven Applications recipes:

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

Single node performance results



https://github.com/UoB-HPC/benchmarks

Arm Provided HPC Software

Just from Arm, other vendors provide their own

Compilers and Libraries

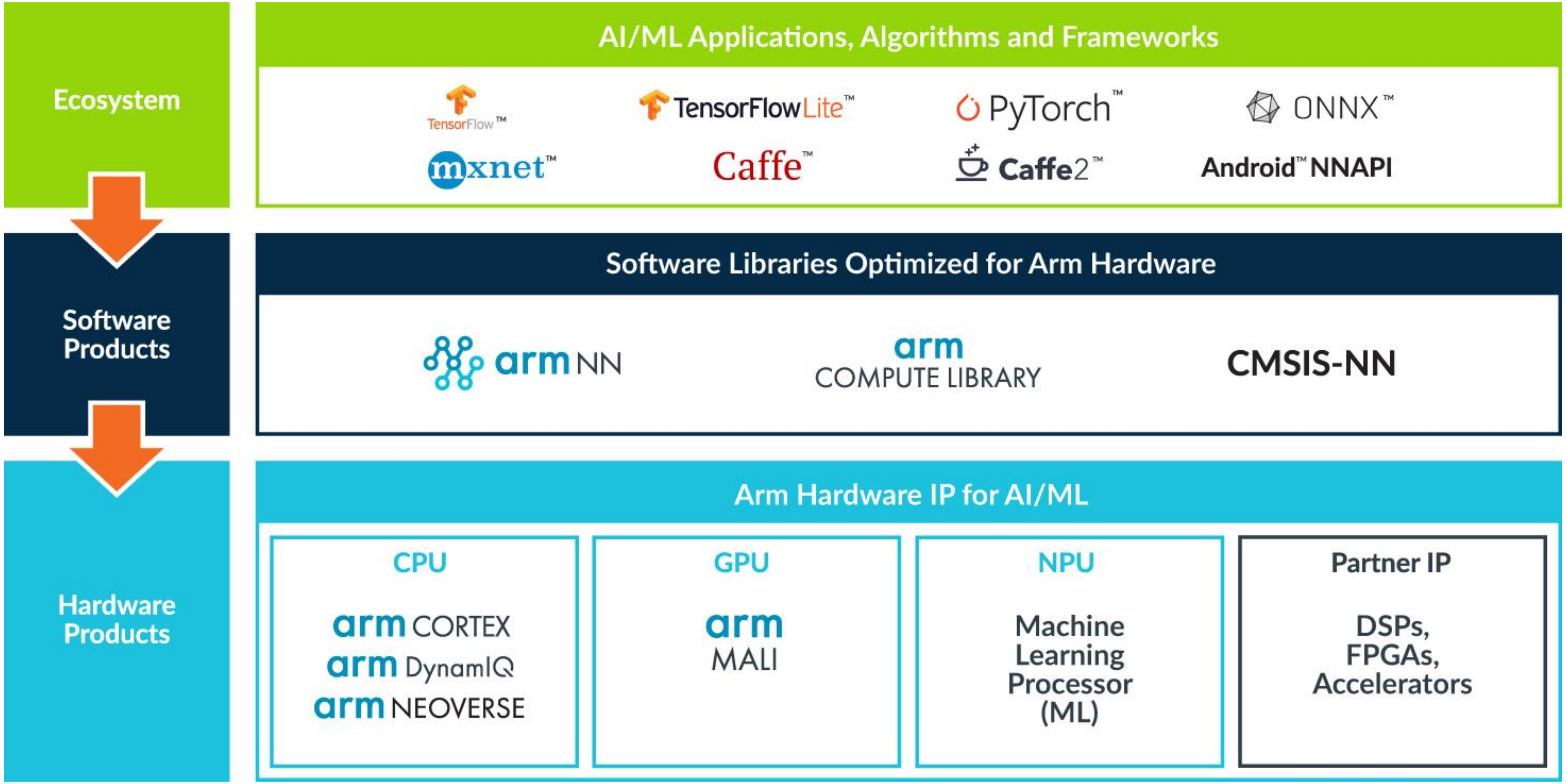
- Arm Compiler for Linux
  - LLVM based compiler optimized for Arm
  - C/C++ & Fortran
- GCC
  - Optimized for Arm
- Maths libraries
  - LAPACK, BLAS, FFT, SpMV, SpMM
  - Transcendentals
  - Micro-architecture optimized

Parallel Tools

- DDT: Debugger
  - C/C++, Fortran and Python
  - MPI, OpenMP, CUDA, OpenACC
- MAP: Performance Profiler
  - Scalable sampling-based profiler
- ArmIE: Instruction Emulator
  - Emulate unsupported instructions
  - SVE

© 2019 Arm Limited

Machine Learning and Artificial Intelligence



Ecosystem

AI/ML Applications, Algorithms and Frameworks

TensorFlow™ | TensorFlowLite™ | PyTorch™ | ONNX™

mxnet™ | Caffe™ | Caffe2™ | Android™ NNAPI

Software Products

Software Libraries Optimized for Arm Hardware

arm NN | arm COMPUTE LIBRARY | CMSIS-NN

Hardware Products

Arm Hardware IP for AI/ML

CPU: arm CORTEX, arm DynamIQ, arm NEOVERSE

GPU: arm MALI

NPU: Machine Learning Processor (ML)

Partner IP: DSPs, FPGAs, Accelerators

© 2019 Arm Limited

ML Frameworks on AArch64

On-CPU server-scale ML workloads

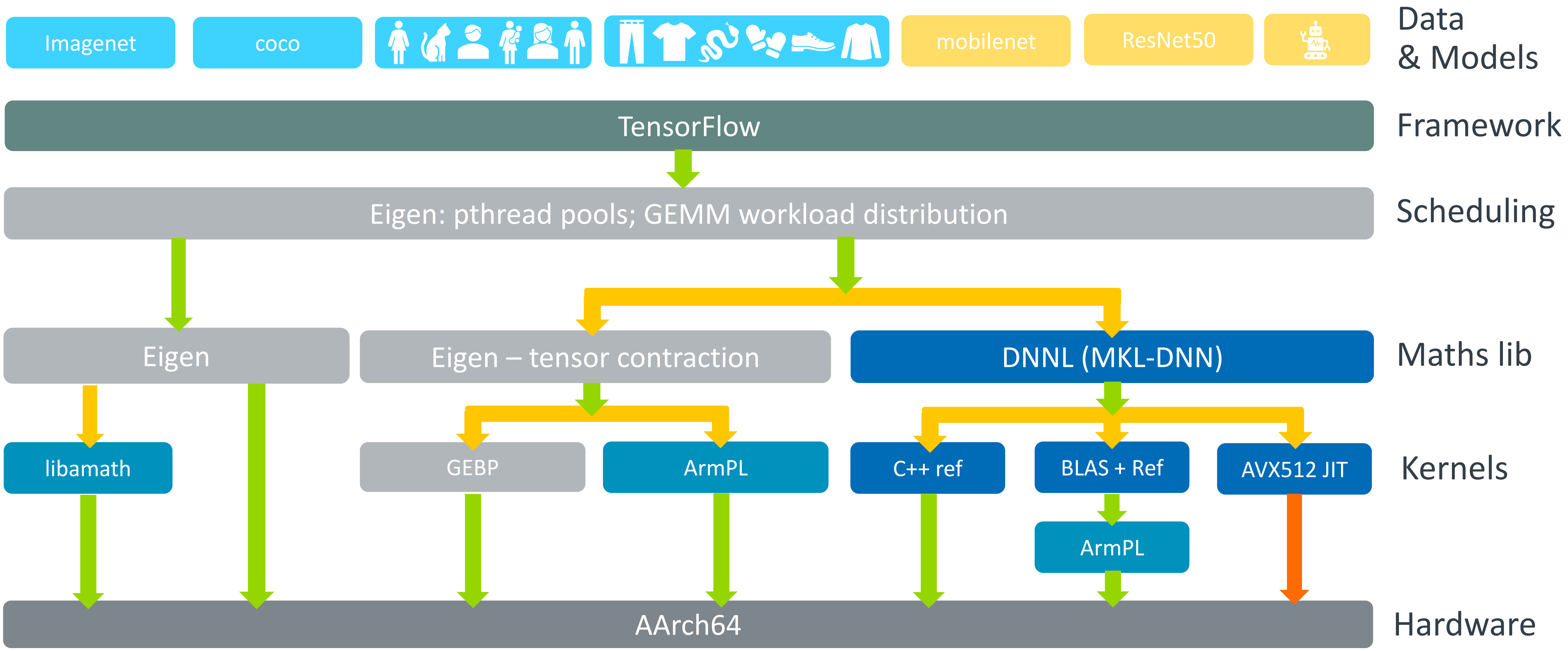
- Leading frameworks and dependencies built on AArch64
  - TensorFlow: https://gitlab.com/arm-hpc/packages/-/wikis/packages/tensorflow
  - PyTorch: https://gitlab.com/arm-hpc/packages/-/wikis/packages/pytorch
  - MXNET: https://gitlab.com/arm-hpc/packages/-/wikis/packages/mxnet

- OS Docker tools for TensorFlow part of *github.com/ARM-software/Tool-Solutions*
  - https://github.com/ARM-software/Tool-Solutions/tree/master/docker/tensorflow-aarch64
    - Compiler: GCC 9.2
    - Maths libraries: Arm Optimized Routines and OpenBLAS 0.3.6
    - Python3 environment built from CPython 3.7 and containing:
      - NumPy 1.17.1
      - TensorFlow 1.15
      - TensorFlow Benchmarks

- Focus has been on TensorFlow, and the maths libraries
  - Inference applications
  - Many-core systems
  - Significant GEMM, and vector maths, work

© 2019 Arm Limited

TensorFlow and maths libraries on AArch64



Imagenet

coco

mobilenet

ResNet50

Data & Models

TensorFlow

Framework

Eigen: pthread pools; GEMM workload distribution

Scheduling

Eigen

Eigen – tensor contraction

DNNL (MKL-DNN)

Maths lib

libamath

GEBP

ArmPL

C++ ref

BLAS + Ref

AVX512 JIT

Kernels

ArmPL

AArch64

Hardware

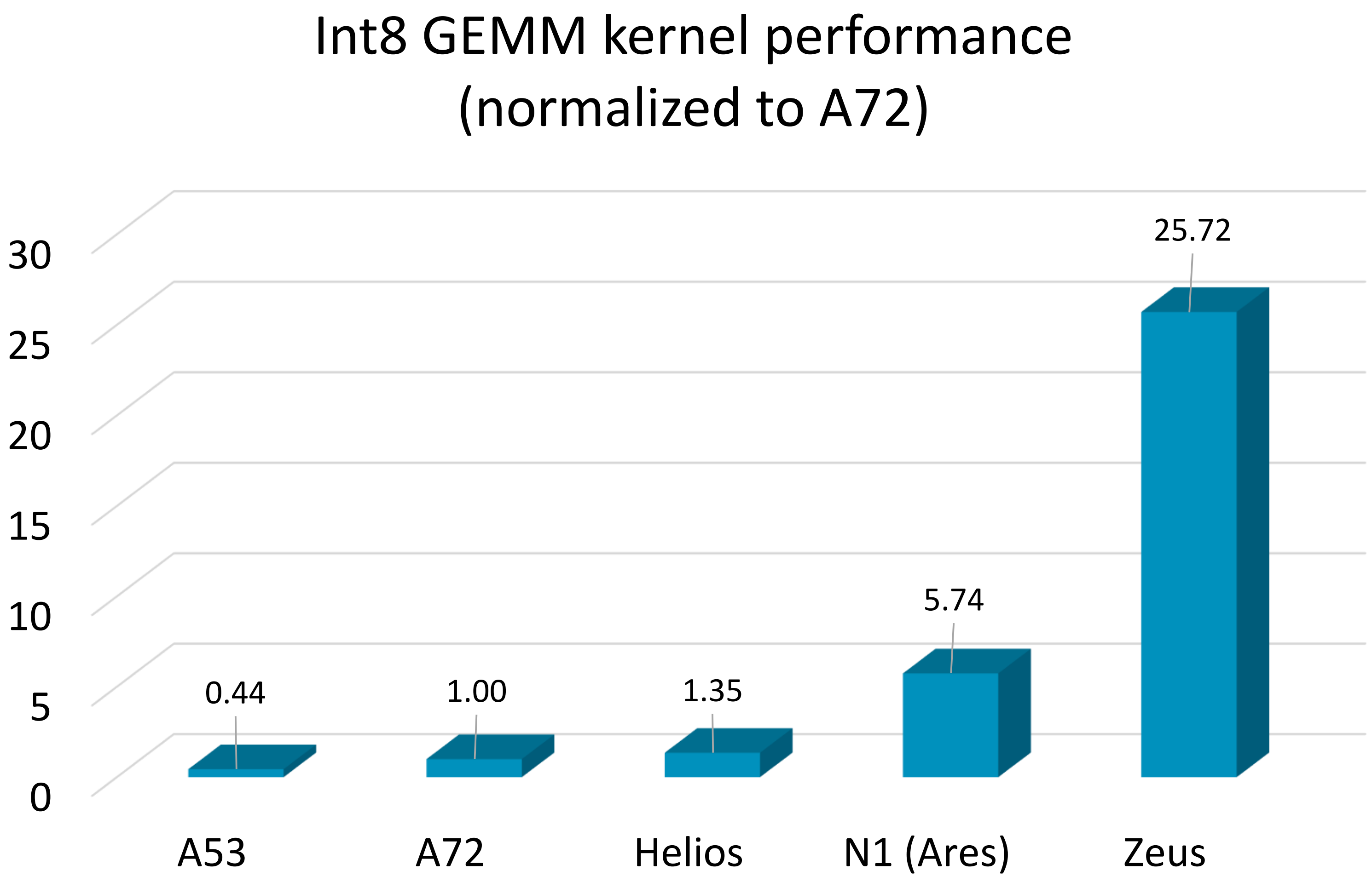
↓ = works on AArch64

↓ = option

↓ = not supported

© 2019 Arm Limited

Increasing ML performance over CPU generations



A72
2x ML performance improvement over Cortex-A53

Helios
>3x ML performance improvement over Cortex-A53 (First Multi-threaded CPU)

N1
>5x ML performance improvement over Cortex-A72 (PPA leadership & ML enhancements)

Zeus
>25x ML performance improvement over Cortext-A72 (Breakthrough ML performance)

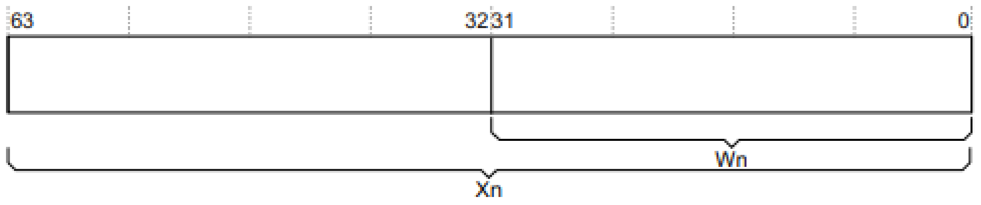
© 2019 Arm Limited

NEON: General Purpose Vector Instructions

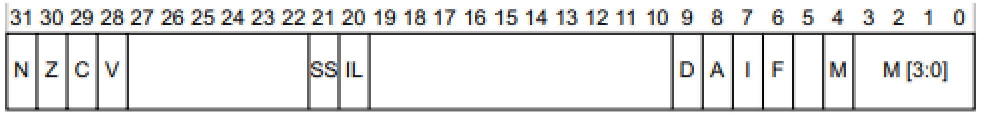
Arm NEON Vector Units

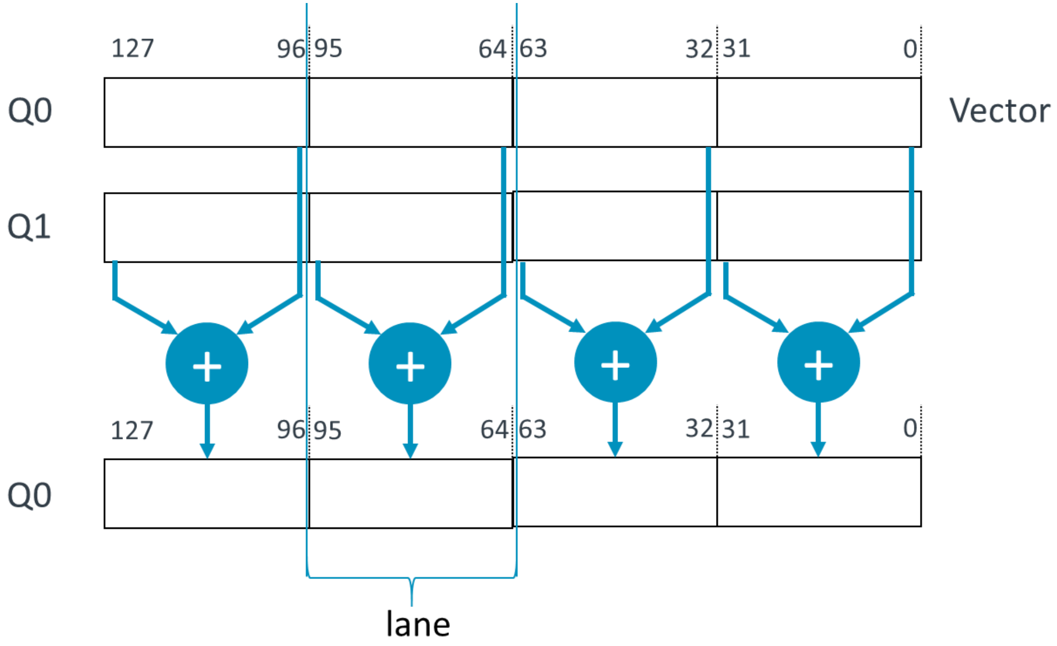
- SIMD Vector Extensions
  - Advanced Single Instruction Multiple Data
  - Fixed width at 128-bit
- As of Armv8-a
  - 31x 64-bit general-purpose registers
    - The 32-bit W register is lower half of 64-bit X register
  - 32x 128-bit floating-point registers
    - D is lower 64 bits of 128-bit Q registers
- Example:
  - fadd   v0.4s, v0.4s, v1.4s
  - Addition of 4 x 32-bit floats
    - 128-bit NEON/32-bit Int = 4 Lanes

64-bit register layout



SPSR: Saved Program State Register





© 2019 Arm Limited

Arm NEON Vector Instructions

- Comprehensive set of vector operations
  - Loads, stores and maths operations
  - Scalar and floating point

$$\underbrace{\texttt{FADD}}_{\text{Instruction}}\ \underbrace{\texttt{<Vd>.<T>,}}_{\text{Destination}}\ \underbrace{\texttt{<Vn>.<T>,}}_{\text{Operand 1}}\ \underbrace{\texttt{<Vm>.<T>}}_{\text{Operand 2}}$$

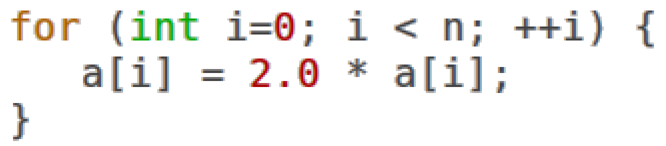
- <Vd> - Destination register, <T> - Type e.g. 4S or 2D

```
FADD V0.4S, V0.4S, V1.4S
```

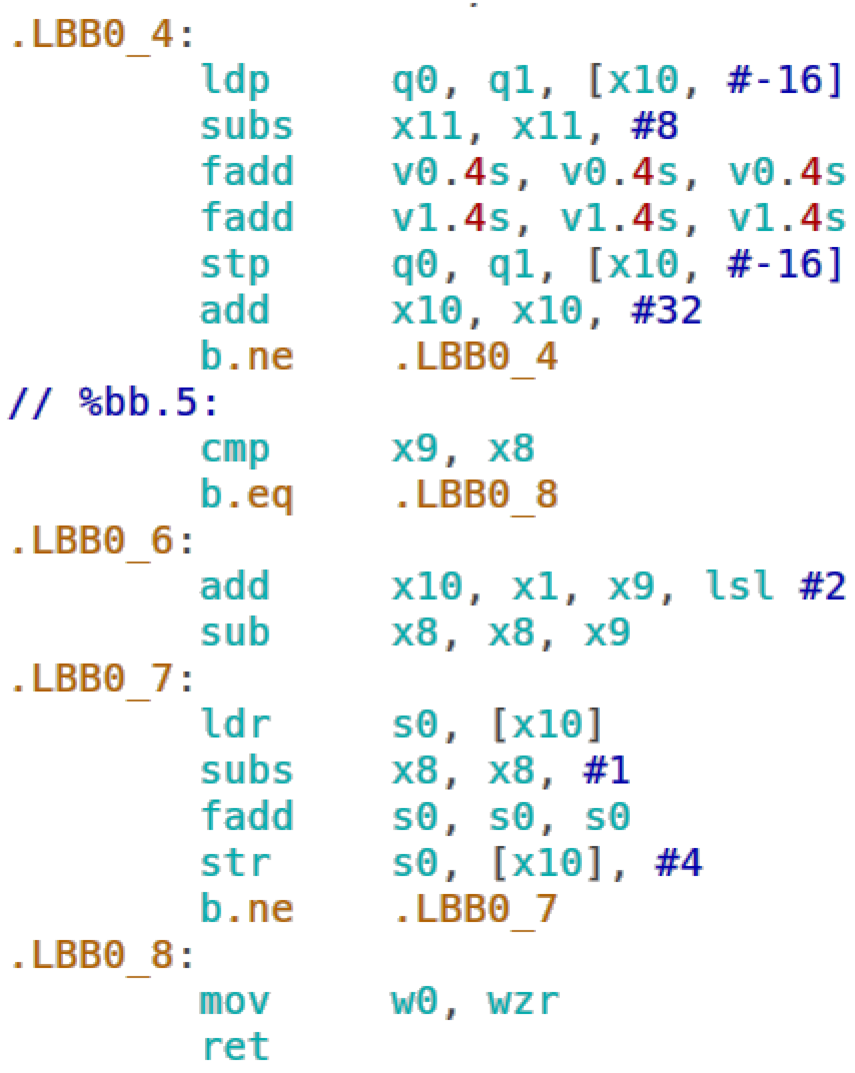
- Add 4 single precision floating point values from V0 to V1 and store in V0
  - V0 += V1

© 2019 Arm Limited

Programming NEON



```
for (int i=0; i < n; ++i) {
   a[i] = 2.0 * a[i];
}
```



```
.LBB0_4:
        ldp     q0, q1, [x10, #-16]
        subs    x11, x11, #8
        fadd    v0.4s, v0.4s, v0.4s
        fadd    v1.4s, v1.4s, v1.4s
        stp     q0, q1, [x10, #-16]
        add     x10, x10, #32
        b.ne    .LBB0_4
// %bb.5:
        cmp     x9, x8
        b.eq    .LBB0_8
.LBB0_6:
        add     x10, x1, x9, lsl #2
        sub     x8, x8, x9
.LBB0_7:
        ldr     s0, [x10]
        subs    x8, x8, #1
        fadd    s0, s0, s0
        str     s0, [x10], #4
        b.ne    .LBB0_7
.LBB0_8:
        mov     w0, wzr
        ret
```

- .LBB0_4: Start with vector loop (and unroll)
  - Load 8 x 32-bit values (into 2 x 128-bit registers)
  - Subtract 8 from loop counter
  - 2x NEON add instructions (register to itself => 2.0*a[i])
  - Store pair of 128-bit registers back
  - Update array offsets
  - Loop if >=8 iterations left

- .LBB0_7: Remainder (fewer than 8 iterations left)
  - Load a single scalar
  - Add it to itself
  - Store
  - Loop if iterations left

© 2019 Arm Limited

Ease of Use for NEON

Where to start

NEON Intrinsics (ACLE)

- `#include arm_neon.h`
  - Header file of NEON intrinsics

- Map to assembly types and instruction names

```
float32x4_t va = vld1q_f32(&a[i]);
va = vmulq_n_f32(va, 2.0);
vst1q_f32(&a[i], va)
```

  - Load 4x 32-bit floats into `*va*` from a[i]
  - Multiply the floats in `*va*` by 2.0
  - Store contents of `*va*` back into a[i]

Auto Vectorisation & Libraries

- Not everyone wants to hand code assembly

- Compilers will generate vector code
  - Generally at optimization levels > -O2
  - Supported in GCC, LLVM, Cray, Arm compiler
  - Vectorisation reports will inform on success

- Vectorised libraries
  - Such as ArmPL maths library

© 2019 Arm Limited

Limitations of NEON

- NEON is firstly only 128-bit
  - Not much use to HPC / Scientific Computing

- Want bigger vectors
  - To expand to 256-bit of 512-bit we would need separate instructions
  - Arm like to offer flexibility to customers - Different vector lengths
    - However it is a RISC architecture (32-bit instruction encoding)

- Suffers from same drawbacks as other vector implementations (AVX)
  - Difficulty to autovectorise
  - Remainder loops

© 2019 Arm Limited

SVE:
Today’s new
Vectorisation Paradigm

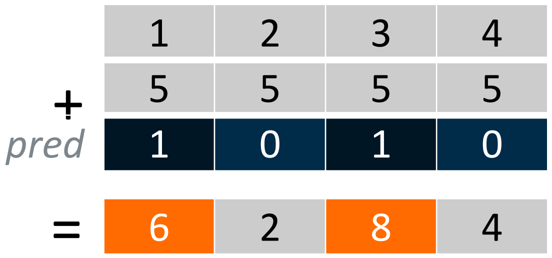
SVE: Scalable Vector Extension

- **SVE is Vector Length Agnostic (VLA)**
  - Vector Length (VL) is a hardware implementation choice from 128 up to 2048 bits.
  - New programming model allows software to scale dynamically to available vector length.
  - No need to define a new ISA, rewrite or recompile for new vector lengths.
- **SVE is not an extension of Advanced SIMD (*aka* Neon)**
  - A separate, optional extension with a new set of instruction encodings.
  - Initial focus is HPC and general-purpose server, not media/image processing.
- **SVE begins to tackle traditional barriers to auto-vectorization**
  - Software-managed speculative vectorization allows uncounted loops to be vectorized.
  - In-vector serialised inner loop permits outer loop vectorization in spite of dependencies.

© 2019 Arm Limited

How can you program when the vector length is unknown?

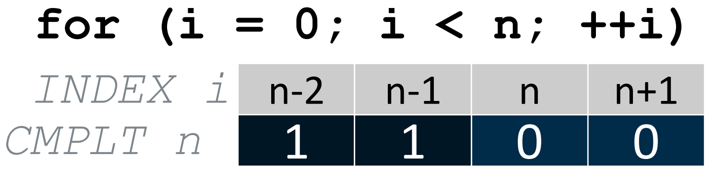
SVE provides features to enable VLA programming from the assembly level and up



	1	2	3	4
+	5	5	5	5
pred	1	0	1	0
=	6	2	8	4

Per-lane predication

Operations work on individual lanes under control of a predicate register.

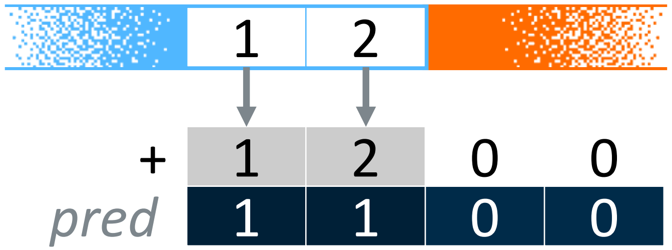


```
for (i = 0; i < n; ++i)
```

INDEX i	n-2	n-1	n	n+1
CMPLT n	1	1	0	0

Predicate-driven loop control and management

Eliminate scalar loop heads and tails by processing partial vectors.



	1	2		
+	1	2	0	0
pred	1	1	0	0

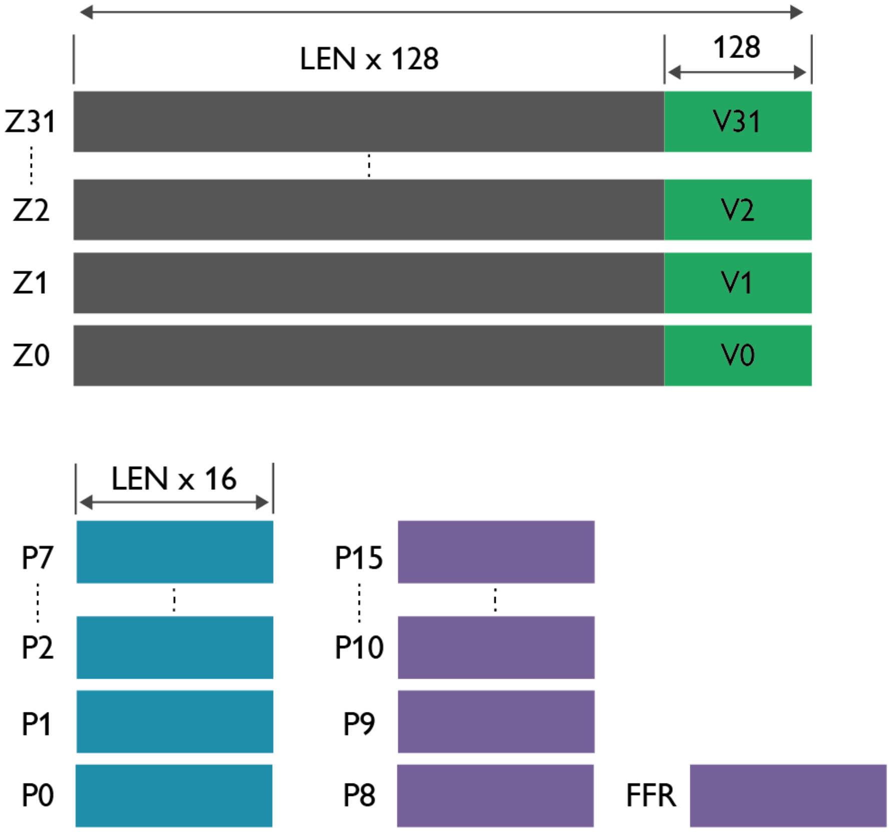
Vector partitioning & software-managed speculation

First Faulting Load instructions allow memory accesses to cross into invalid pages.

© 2019 Arm Limited

SVE Registers

- **Scalable vector registers**
  - `Z0-Z31` extending NEON's 128-bit `V0-V31`.
  - Packed DP, SP & HP floating-point elements.
  - Packed 64, 32, 16 & 8-bit integer elements.
- **Scalable predicate registers**
  - `P0-P15` predicates for loop / arithmetic control.
  - 1/8th size of SVE registers (1 bit / byte).
  - `FFR` first fault register for software speculation.

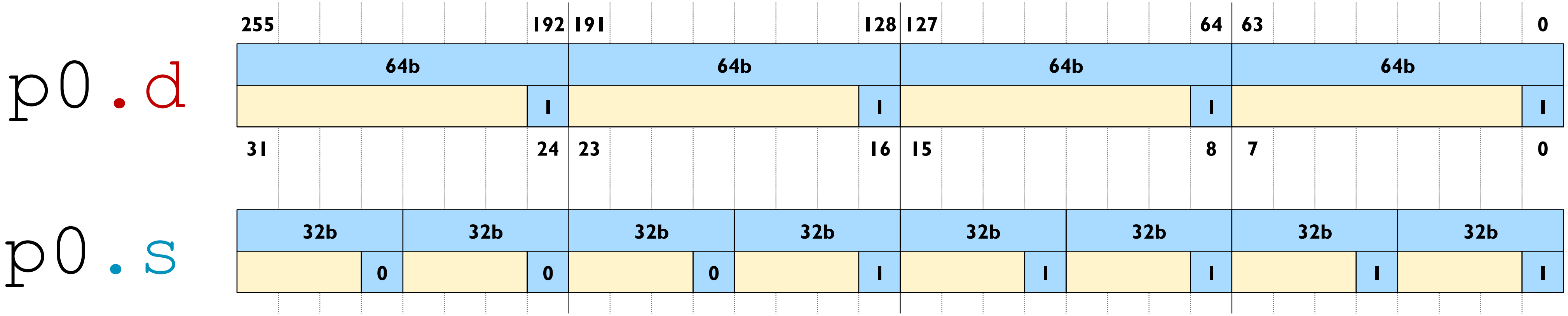


© 2019 Arm Limited

Predicates: Active Lanes vs Inactive Lanes

Predicate registers track lane activity

- 16 predicate registers (P0-P15)
- 1 predicate bit per 8 vector bits (lowest predicate bit per lane is significant)
- On **load**, active elements update the destination
- On **store**, inactive lanes leave destination unchanged (p0/m) or set to 0's (p0/z)



© 2019 Arm Limited

SVE Predicate condition flags

SVE is a *predicate-centric* architecture

- Predicates are central, not an afterthought
- Support complex nested conditions and loops.
- Predicate generation also sets condition flags.
- Reduces vector loop management overhead.

Overloading the A64 NZCV condition flags

Flag	SVE	Condition
N	First	Set if first active element is true
Z	None	Set if no active element is true
C	!Last	Set if last active element is false
V		Scalarized loop state, else zero

Reuses the A64 conditional instructions

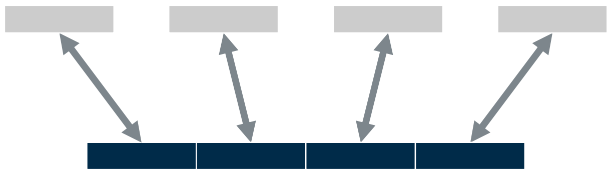
- Conditional branches B.EQ → B.NONE
- Conditional select, set, increment, etc.

Condition Test	A64 Name	SVE Alias	SVE Interpretation
Z=1	EQ	NONE	No active elements are true
Z=0	NE	ANY	Any active element is true
C=1	CS	NLAST	Last active element is not true
C=0	CC	LAST	Last active element is true
N=1	MI	FIRST	First active element is true
N=0	PL	NFRST	First active element is not true
C=1 & Z=0	HI	PMORE	More partitions: some active elements are true but not the last one
C=0 | Z=1	LS	PLAST	Last partition: last active element is true or none are true
N=V	GE	TCONT	Continue scalar loop
N!=V	LT	TSTOP	Stop scalar loop

© 2019 Arm Limited

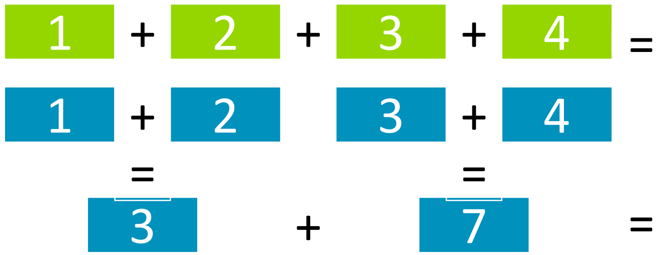
SVE supports vectorization in complex code

Right from the start, SVE was engineered to handle codes that usually won't vectorize



Gather-load and scatter-store

Loads a single register from several non-contiguous memory locations.



Extended floating-point horizontal reductions

In-order and tree-based reductions trade-off performance and repeatability.

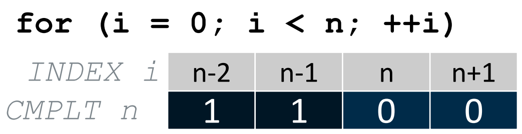
© 2019 Arm Limited

Questions and Break

Part 2: Going Forward with SVE

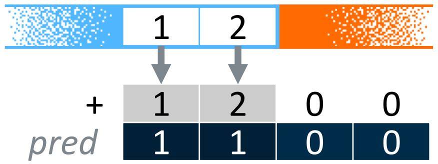
Quick Recap

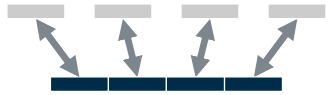
- SVE enables Vector Length Agnostic (VLA) programming
- VLA enables portability, scalability, and optimization
- Predicates control which operations affect which vector lanes
  - Predicates are not bitmasks
  - You can think of them as dynamically resizing the vector registers
- The actual vector length is set by the CPU architect
  - Any multiple of 128 bits up to 2048 bits
  - May be dynamically reduced by the OS or hypervisor
- SVE was designed for HPC and can vectorize complex structures
- Many open source and commercial tools currently support SVE

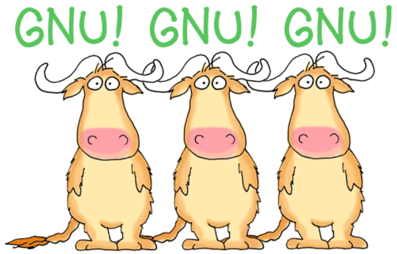


```
for (i = 0; i < n; ++i)
```

INDEX i	n-2	n-1	n	n+1
CMPLT n	1	1	0	0







© 2019 Arm Limited

VLA Programming

Vector Length Agnostic

Vector Length Agnostic Programming

A paradigm shift for developers

Advantages

- Not thinking about vector length
  - Rather just vectorization
- No peel/remainder loops
  - All handled by predication
- Key is loop structures
  - Predicates are powerful

Considerations

- Should not be writing fixed width
  - Applies to data structures and instructions
  - More portable for different hardware
- Can the compiler identify loop structure?
  - Generate predicated instructions
- However: VLA *may* be slower
  - Cost of generating predicates
  - Near empty loops

© 2019 Arm Limited

How do you count by vector width?

No need for multi-versioning: one increment to rule all vector sizes

```
ld1w z1.s, p0/z, [x0,x4,lsl 2]      // p0:z1=x[i]
ld1w z2.s, p0/z, [x1,x4,lsl 2]      // p0:z2=y[i]
fmla z2.s, p0/m, z1.s, z0.s         // p0?z2+=x[i]*a
st1w z2.s, p0, [x1,x4,lsl 2]        // p0?y[i]=z2

incw x4                             // i+=(VL/32)
```

“Increment `x4` by the number of 32-bit lanes (`w`) that fit in a VL.”

© 2019 Arm Limited

Initialization when vector length is unknown

- Vectors cannot be initialized from compile-time constant, so…
  - `INDEX Zd.S,#1,#4` : `Zd = [ 1, 5, 9, 13, 17, 21, 25, 29 ]`
- Predicates cannot be initialized from memory, so…
  - `PTRUE  Pd.S, MUL3` : `Pd = [ T, T, T , T, T, T , F, F ]`
- Vector loop increment and trip count are unknown at compile-time, so…
  - `INCD Xi` : increment scalar Xi by # of 64b dwords in vector
  - `WHILELT Pd.D,Xi,Xe` : next iteration predicate `Pd = [ while i++ < e ]`
- Vectors stores to stack must be dynamically allocated and indexed, so…
  - `ADDVL SP,SP,#-4` : decrement stack pointer by (4*VL)
  - `STR Zi, [SP,#3,MUL VL]` : store vector Z1 to address (SP+3*VL)

© 2019 Arm Limited

Vectorizing A Scalar Loop With ACLE

a[:] = 2.0 * a[:]

Original Code

```
for (int i=0; i < N; ++i) {
  a[i] = 2.0 * a[i];
}
```

128-bit NEON vectorization

```
int i;

// vector loop
for (i=0; (i<N-3) && (N&~3); i+=4) {
  float32x4_t va = vld1q_f32(&a[i]);
  va = vmulq_n_f32(va, 2.0);
  vst1q_f32(&a[i], va)
}
// drain loop
for (; i < N; ++i)
  a[i] = 2.0 * a[i];
```

This is NEON, ***not*** SVE!

© 2019 Arm Limited

```
for (int i=0; i < N; ++i) {
  a[i] = 2.0 * a[i];
}
```

Vectorizing A Scalar Loop With ACLE

a[:] = 2.0 * a[:]

128-bit NEON vectorization

```
int i;

// vector loop
for (i=0; (i<N-3) && (N&~3); i+=4) {
  float32x4_t va = vld1q_f32(&a[i]);
  va = vmulq_n_f32(va, 2.0);
  vst1q_f32(&a[i], va)
}
// drain loop
for (; i < N; ++i)
  a[i] = 2.0 * a[i];
```

SVE vectorization

```
for (int i = 0 ; i < N; i += svcntw() )

{

  svbool_t Pg = svwhilelt_b32(i, N);

  svfloat32_t va = svld1(Pg, &a[i]);

  va = svmul_x(Pg, va, 2.0);

  svst1(Pg, &a[i], va);

}
```

© 2019 Arm Limited

Vectorizing A Scalar Loop With ACLE

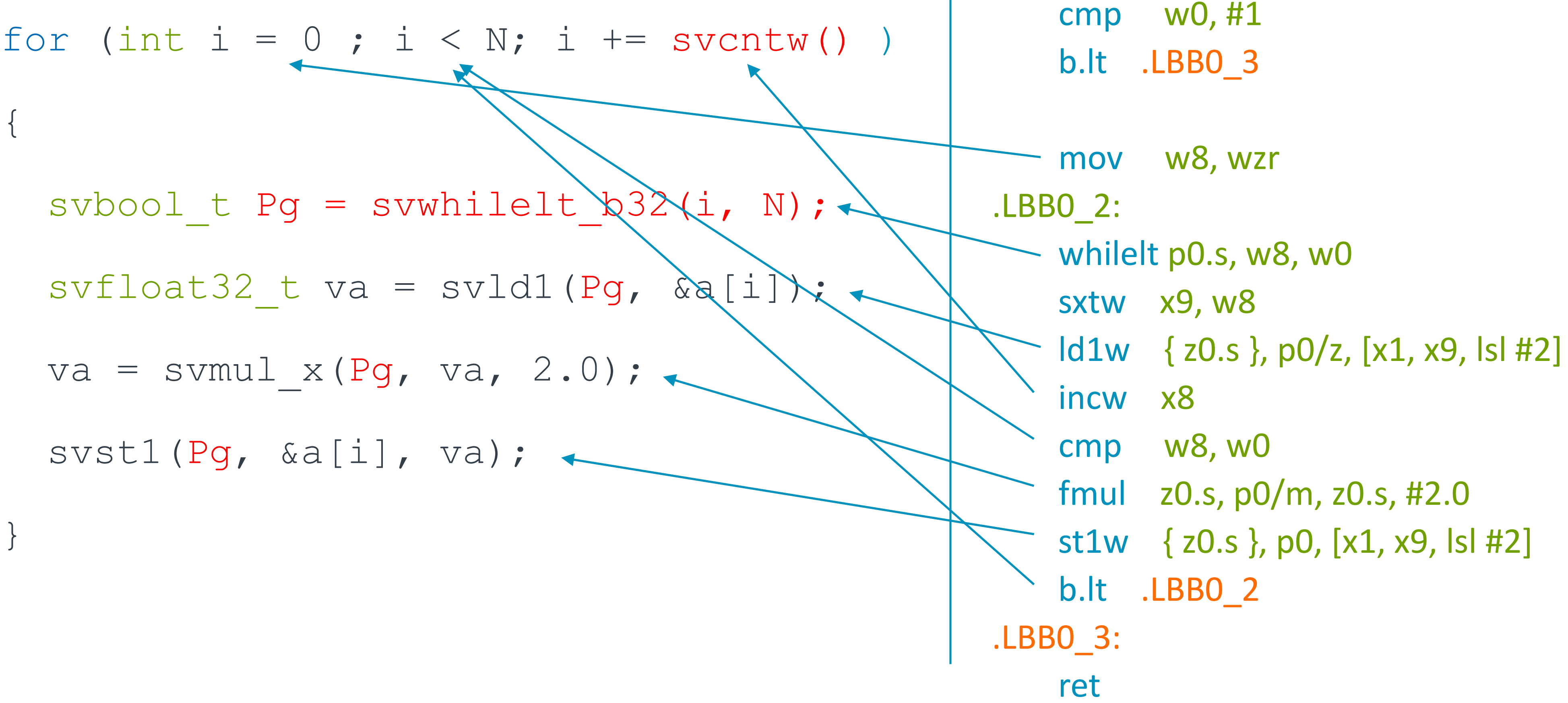
a[:] = 2.0 * a[:]

```
for (int i=0; i < N; ++i) {
  a[i] = 2.0 * a[i];
}
```

SVE vectorization

Assembly

armclang -march=armv8.2-a+sve -O3 -S sve2.c



```
for (int i = 0 ; i < N; i += svcntw() )
{
   svbool_t Pg = svwhilelt_b32(i, N);
   svfloat32_t va = svld1(Pg, &a[i]);
   va = svmul_x(Pg, va, 2.0);
   svst1(Pg, &a[i], va);
}
```

```
        cmp     w0, #1
        b.lt    .LBB0_3

        mov     w8, wzr
.LBB0_2:
        whilelt p0.s, w8, w0
        sxtw    x9, w8
        ld1w    { z0.s }, p0/z, [x1, x9, lsl #2]
        incw    x8
        cmp     w8, w0
        fmul    z0.s, p0/m, z0.s, #2.0
        st1w    { z0.s }, p0, [x1, x9, lsl #2]
        b.lt    .LBB0_2
.LBB0_3:
        ret
```

© 2019 Arm Limited

SVE Gives You More

- SVE is really powerful (mainly due to predicates)
  - Compilers can exploit this power
  - Autovectorisation getting much better

- Power is also being able to vectorise new things
  - Previously hard to vectorise
  - Mapping IF statements to predicates

- Shown in the following real-world example
  - Users code had a ‘mask’ in the inner loop
  - Previously prevented vectorisation

© 2019 Arm Limited

More Complex Example

Code

```
void foo(char* mask,
             double * result, int n)
{
   for (int x = 0; x < n; x++)
        if(mask[x] == 0)
             result[x] += 2.0;


}
```

```
armclang -S -march=armv8.2-a+sve -O3 sve.c
```

Assembly

```
.LBB0_7:
      mov     x9, xzr
      whilelo p0.d, xzr, x8
      mov     z0.d, #0
      fmov    z1.d, #2.00000000

.LBB0_8:
      ld1b    { z2.d }, p0/z, [x0, x9]
      cmpeq   p0.d, p0/z, z2.d, z0.d
      ld1d    { z2.d }, p0/z, [x1, x9, lsl #3]
      fadd    z2.d, z2.d, z1.d
      st1d    { z2.d }, p0, [x1, x9, lsl #3]
      incd    x9
      whilelo p0.d, x9, x8
      b.mi    .LBB0_8

.LBB0_9:
      ret
```

Annotations

Generate initial loop predicate

Generate initial register values

Load mask with loop predicate

Generate predicate mask[i]==0

Load result with mask predicate

Addition with mask predicate

Store with mask predicate

Loop counter increment

Generate new loop predicate

Loop

© 2019 Arm Limited

Toolchain Support

Getting Help

How to Make Use of SVE

Using the tools that are available

- Many ways to use SVE in your application
  - Different levels of effort, reward and portability

- SVE libraries
  - Such as SVE ArmPL for SVE maths operations, just link the correct version
- Auto-vectoristion
  - Compiler spots vectorisation and exploits it, compiler reports to inform
  - `-march=armv8-a+sve`
- Pragmas
  - Help and guide auto-vectorisation (e.g. OMP SIMD, IVDEP)
- Intrinsics
  - ACLE for SVE - friendly interface into SVE instructions
- Assembly
  - Hand code your SVE instructions

© 2019 Arm Limited

Arm Compiler for Linux user-space

Commercial C/C++/Fortran compiler with best-in-class performance

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

© 2019 Arm Limited

Arm Compiler – Auto-vectorization

LLVM9

- Arm pulls all relevant cost models and optimizations into the downstream codebase.
- Auto-vectorization via LLVM **vectorizers:**
  - Use cost models to drive decisions about what code blocks can and/or should be vectorized.
  - Since October 2018, two different vectorizers used from LLVM: Loop Vectorizer and SLP Vectorizer.
- Loop Vectorizer support for SVE and NEON:

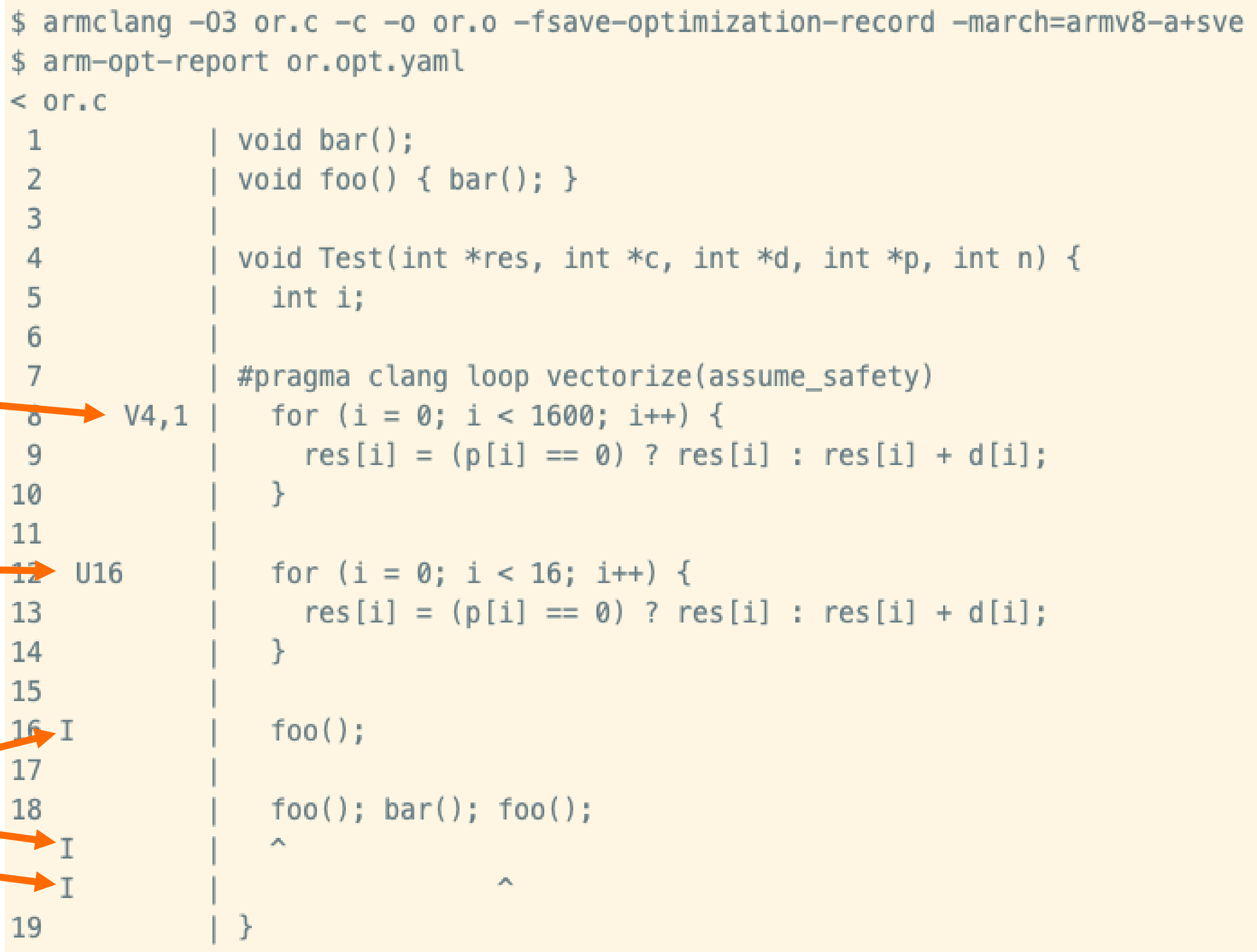
- Loops with unknown trip count
- Runtime checks of pointers
- Reductions
- Inductions
- "If" conversion

- Pointer induction variables
- Reverse iterators
- Scatter / gather
- Vectorization of mixed types
- Global structures alias analysis

© 2019 Arm Limited

New: ACFL Vectorisation Reports

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



```
$ 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          | }
```

Vectorized

4x 32-bit lanes

1-way interleaving

Fully unrolled

All 3 instances of `foo()` were inlined

Arm Performance Libraries

Optimized BLAS, LAPACK and FFT

Now
With
SVE

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 math.h functions implementations

Best-in-class serial and parallel performance

- Generic Armv8-A optimizations by Arm
- Tuning for specific platforms like Marvell 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

© 2019 Arm Limited

SVE Compiler Support

Compiler	Assembly / Disassembly	Inline Assembly	ACLE	Auto-vectorization	Math Libraries
Arm Compiler for HPC	SVE + SVE2	SVE + SVE2	SVE + SVE2	SVE+ SVE2	SVE
LLVM/Clang	SVE + SVE2	SVE + SVE2	SVE + SVE2 in LLVM 10	SVE + SVE2 in LLVM 11	
GNU	SVE + SVE2	SVE + SVE2	SVE + SVE2 in GNU 10	SVE now SVE2 in GNU10	

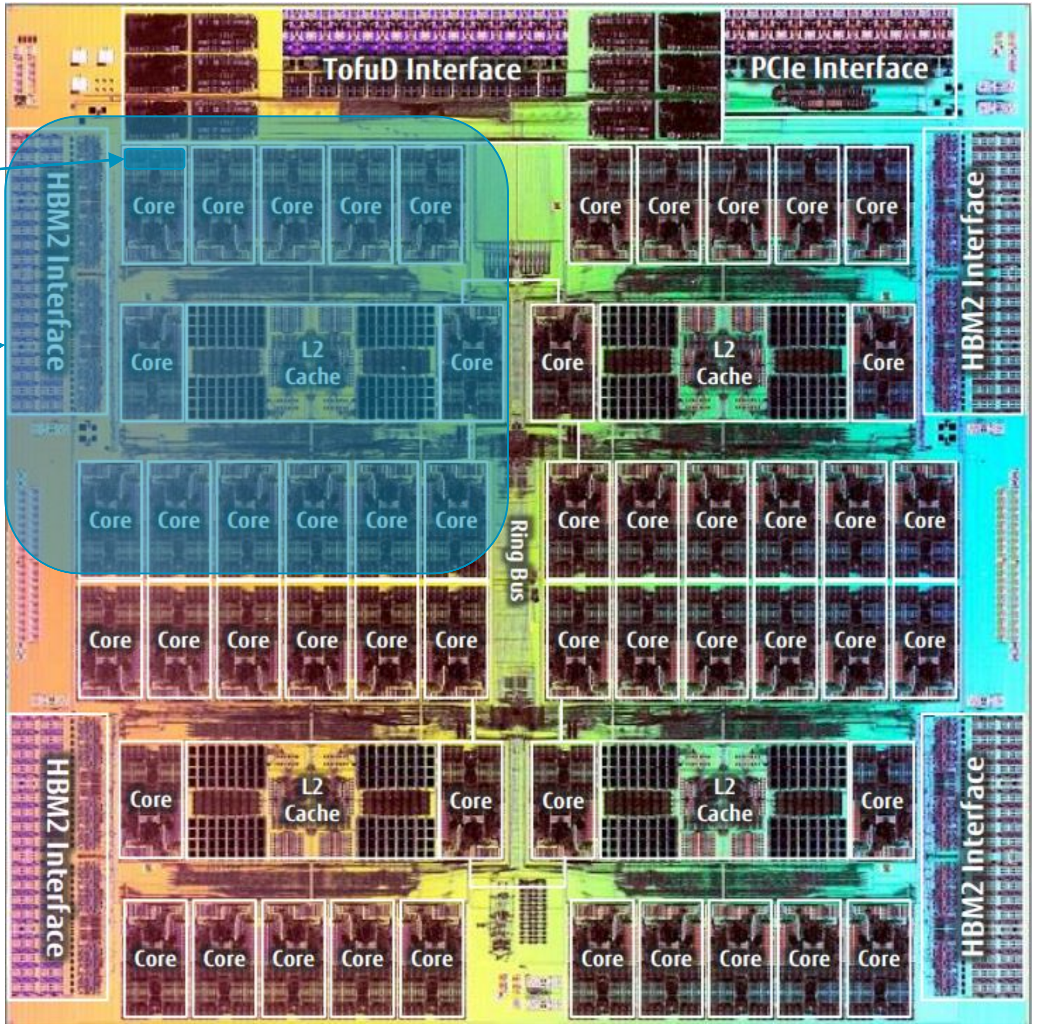
© 2019 Arm Limited

Realising SVE

The First SVE CPU - Available ~Now

Fujitsu A64FX

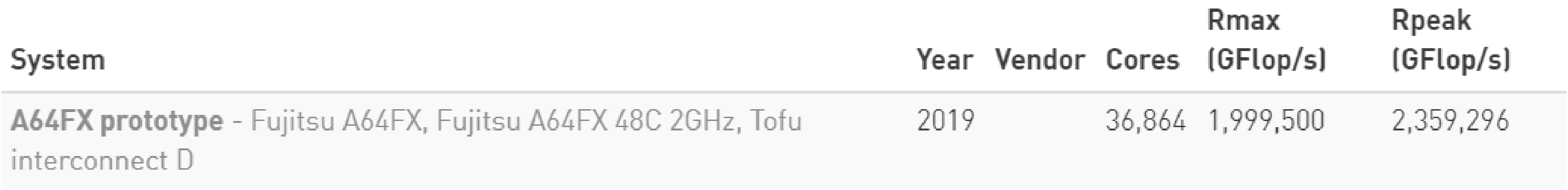
- Custom Arm design by Fujitsu
- 512-bit SVE
- 4 Core Memory Groups
  - 12 cores + 1 OS core
  - 8 GB High Bandwidth Memory (HBM)
- 1 socket / node, 2 nodes / blade
  - Direct water cooling



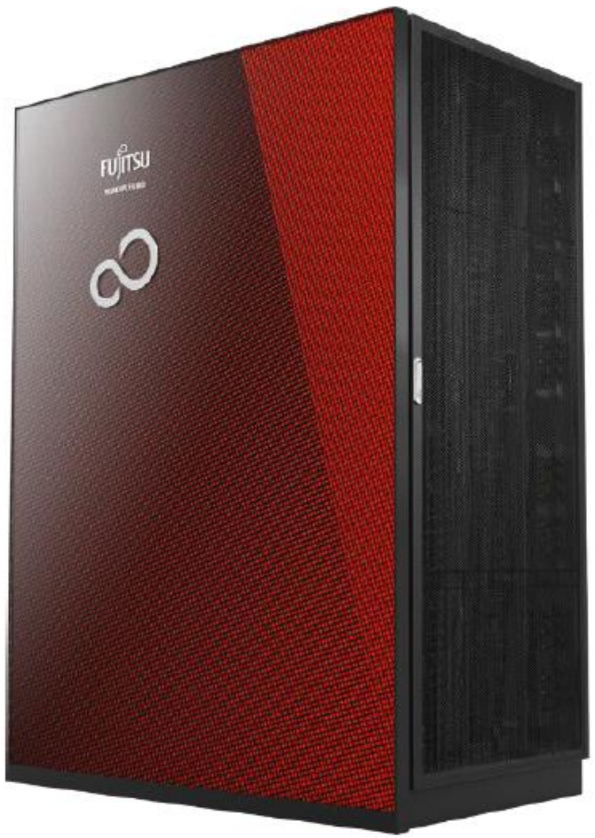
© 2019 Arm Limited

https://www.top500.org/system/179706

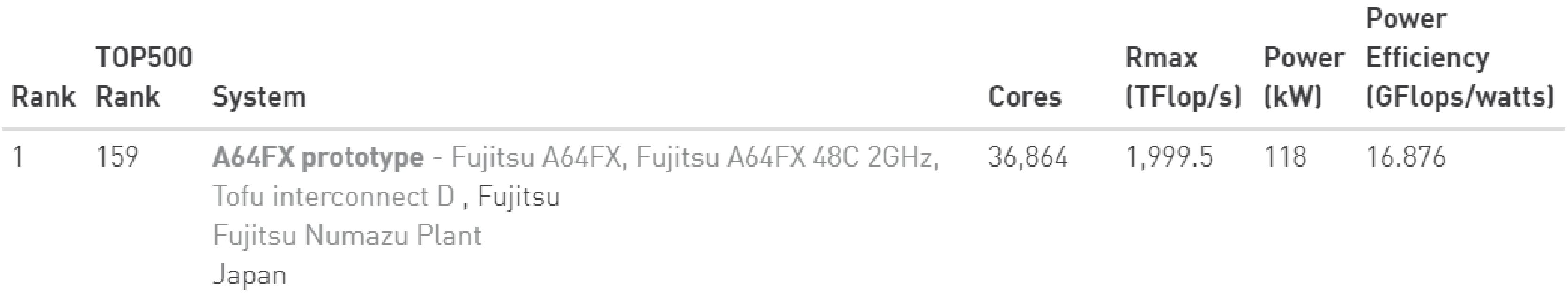
A64FX Now in Top500 - #159



System	Year	Vendor	Cores	Rmax (GFlop/s)	Rpeak (GFlop/s)
A64FX prototype - Fujitsu A64FX, Fujitsu A64FX 48C 2GHz, Tofu interconnect D	2019		36,864	1,999,500	2,359,296



Green500 - #1



Rank	TOP500 Rank	System	Cores	Rmax (TFlop/s)	Power (kW)	Power Efficiency (GFlops/watts)
1	159	**A64FX prototype** - Fujitsu A64FX, Fujitsu A64FX 48C 2GHz, Tofu interconnect D , Fujitsu Fujitsu Numazu Plant Japan	36,864	1,999.5	118	16.876

© 2019 Arm Limited

Post SVE: The Vector Instructions of the Next Generation

FMMLA: High Performance Matrix Multiplication

- Added to Armv8.6
  - NEON and SVE instructions
  - FMMLA instructions for FP (SVE)

```
FMMLA <Zda>.S, <Zn>.S, <Zm>.S
FMMLA <Zda>.D, <Zn>.D, <Zm>.D
```

- 2x2 matrix multiplication
  - Works on multiple of ‘vector granules’
  - 2x2xFP32 = 128-bit granules
  - Assumes vector length is multiple

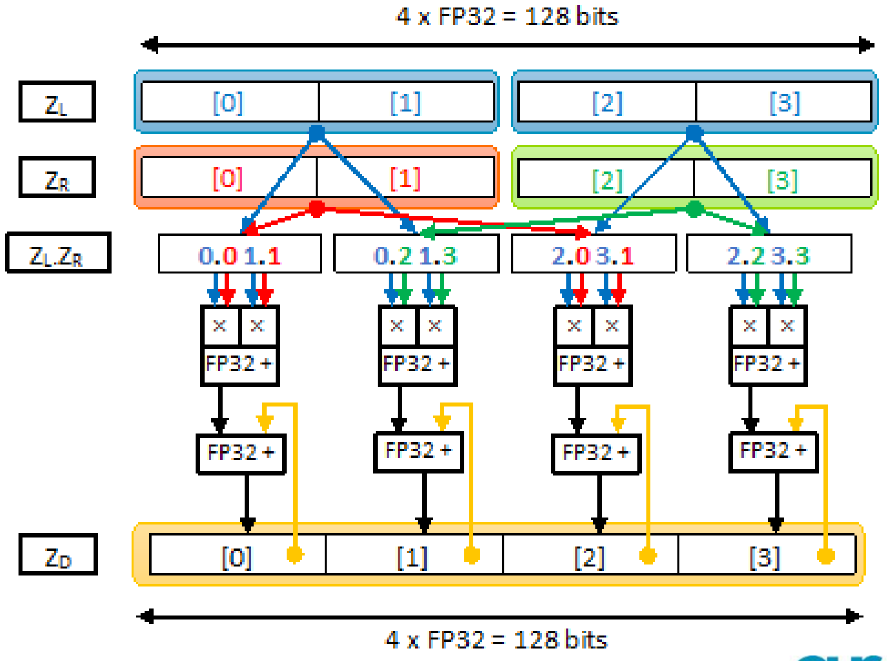
- May require layout transformations
  - Outer loop to minimise cost

- Accelerated libraries

$$\begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} \mathrel{+}= \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} \;X\; \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array}$$

Dest (D)	Left (L)	Right (R)
2x2xFP32	2x2xFP32	2x2xFP32

D[0] += (L[0] * R[0]) + (L[1] * R[1])
D[1] += (L[0] * R[2]) + (L[1] * R[3])
D[2] += (L[2] * R[0]) + (L[3] * R[1])
D[3] += (L[2] * R[2]) + (L[3] * R[3])



© 2019 Arm Limited

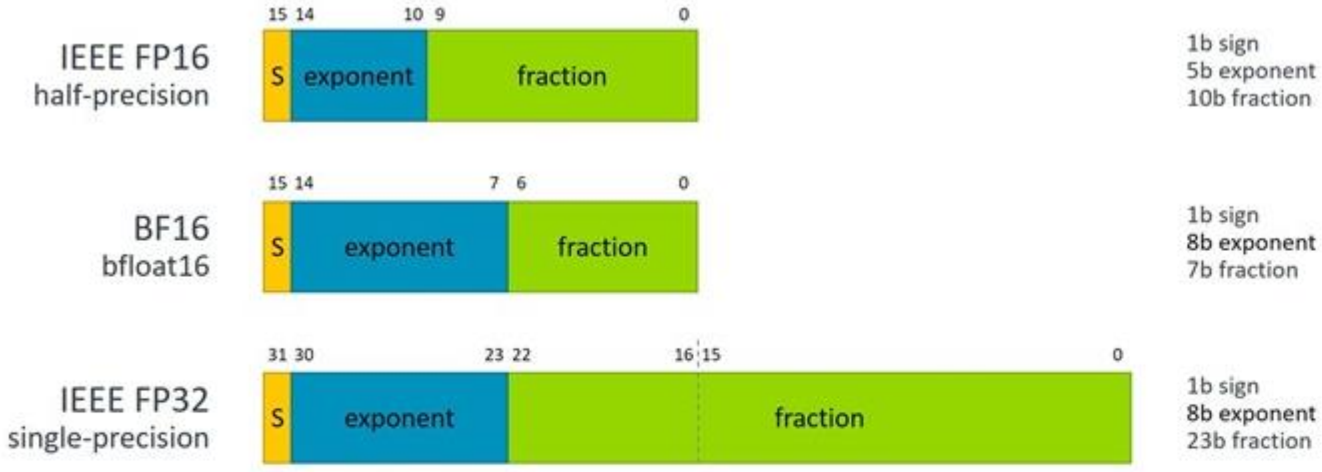
New Data Type Support: BFloat16

- New addition to Armv8-A
  - Adds support for BF16

- Instructions for NEON and SVE
  - Including:
    - **BFDOT:** Dot Product (1x2)x(2x1)
    - **BFMMLA:** Mat Multiply (2x4)x(4x2)

- Significant performance gains
  - ML training and inference workloads

- Supported in Arm libraries
  - Arm NN and Arm Compute Libraries



© 2019 Arm Limited

Scalable Vector Extensions V2 (SVE2)

SVE for non HPC markets

Built on SVE

Improved scalability

Vectorization of more workloads

- Built on the SVE foundation.
  - Scalable vectors with hardware choice from 128 to 2048 bits.
  - Vector-length agnostic programming for “write once, run anywhere”.
  - Tackles some obstacles to compiler auto-vectorisation.
- Scaling single-thread performance to exploit long vectors.
  - SVE2 adds NEON™-style fixed-point DSP/multimedia plus other new features.
  - Performance parity and beyond with classic NEON DSP/media SIMD.
  - Tackles further obstacles to compiler auto-vectorization.
- Enables vectorization of a wider range of applications than SVE.
  - Multiple use cases in Client, Edge, Server and HPC.
    - DSP, Codecs/filters, Computer vision, Photography, Game physics, AR/VR,
    - Networking, Baseband, Database, Cryptography, Genomics, Web serving.
  - Improves competitiveness of Arm-based CPU vs proprietary solutions.
  - Reduces s/w development time and effort.

© 2019 Arm Limited

SVE2 Instructions Add:

What's new

- Thorough support for fixed-point DSP arithmetic
  - (traditional Neon DSP/Media processing, complex numbers arithmetic for LTE)
- Multi-precision arithmetic
  - (bignum, crypto)
- Non-temporal gather/scatter
  - (HPC, sort)
- Enhanced permute and bitwise permute instructions
  - (CV, FIR, FFT, LTE, ML, genomics, cryptanalysis)
- Histogram acceleration support
  - (CV, HPC, sort)
- String processing acceleration support
  - (parsers)
- (optional) Cryptography support instructions for AES, SM4, SHA standards
  - (encryption)

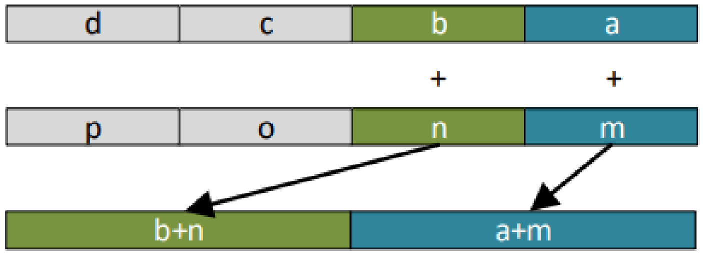
© 2019 Arm Limited

Example: Widening and Narrowing

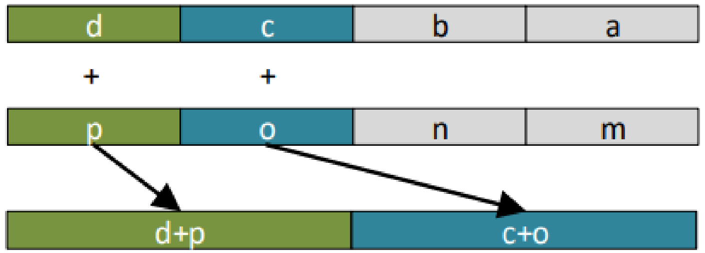
NEON vs SVE2

NEON

UADDL Vd.2D, Vn.2S, Vm.2S



UADDL2 Vd.2D, Vn.4S, Vm.4S

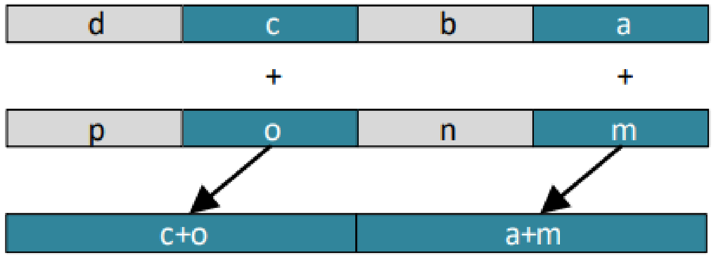


- NEON uses high/low half of vector
- Expensive for large vector lengths
  - >> 128-bit

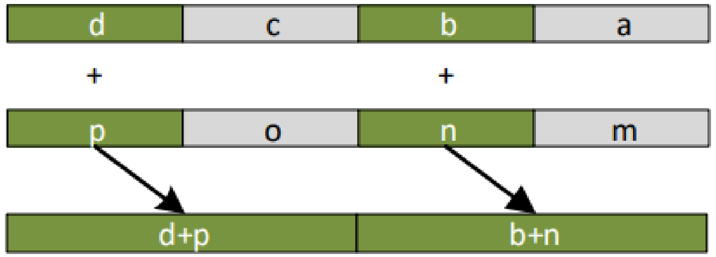
- SVE2 uses odd/even half of vector
- Bottom and top
- Happens ‘in-lane’

SVE2

UADDLB Zd.D, Zn.S, Zm.S



UADDLT Zd.D, Zn.S, Zm.S



© 2019 Arm Limited

Transactional Memory Extension (TME)

Scalable Thread-Level Parallelism (TLP) for multi-threaded applications

Hardware Transactional Memory

Improved scalability

Simpler software design

- Hardware Transactional Memory (HTM) for the Arm architecture.
  - Improved competitiveness with other architectures that support HTM. • Strong isolation between threads.
  - Failure atomicity.
- Scaling multi-thread performance to exploit many-core designs.
  - Database.
  - Network dataplane.
  - Dynamic web serving.
- Simplifies software design for massively multi-threaded code.
  - Supports Transactional Lock Elision (TLE) for existing locking code.
  - Low-level concurrent access to shared data is easier to write and debug.

© 2019 Arm Limited

Conclusion

Vector Instruction Sets on Arm

- Arm is new to HPC but has a compelling justification and heritage

- Arm has a constantly evolving set of vector instructions
  - From NEON to SVE and beyond

- Designed to give flexibility to hardware designers
  - Maximise ability to differentiate
  - Still present a consistent end-user experience

- Designed for real use-cases for improved performance
  - Started with HPC workloads
  - Moving to AI / ML server

© 2019 Arm Limited

The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks