DEUCALION

HPC Software Portability: x86 to ARM

B. Malaca

INCD/Deucalion

malaca@di.uminho.pt

Co-funded by the European Union

This project has received funding from the High Performance Computing Joint Undertaking under grant agreement No 101139786

Outline

Why try to port your code from x86 to ARM

Portability challenges you might face

Best practices (and things you can implement rapidly)

Conclusions

New opportunities ARM

DEUCALION

Chip Designers

Different motivations for choosing ARM:

- cost
- licensing
- independence
- geopolitics

HPC Engineers

Useful to embrace ARM: diversity of HPC systems, allowing for partitions tuned specifically for some applications:

- AI
- Finance
- CFD
- other HPC applications

In Deucalion we have 3 partitions, **ARM, x86** and a (smaller) **GPU-accelerated**

Users

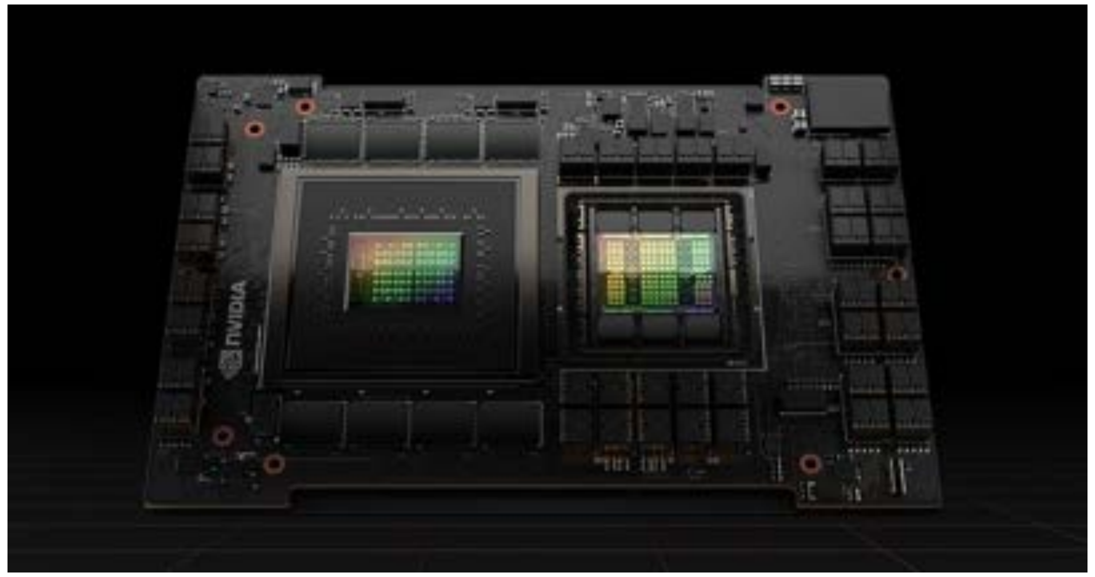
Users need to adapt their codes to take advantage of the available computing power (including on Deucalion)

First European Exascale computer will be ARM-based

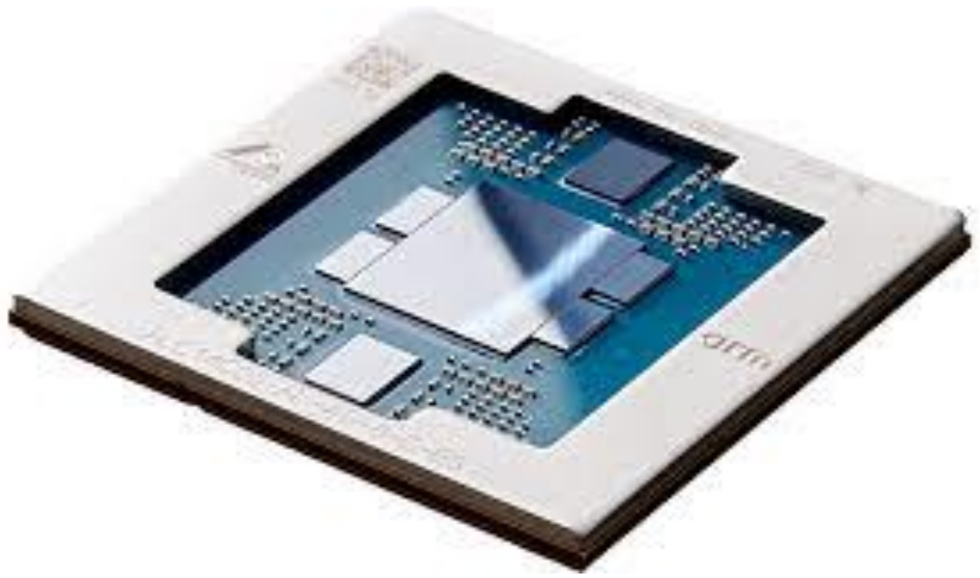
ARM is not a uniform architecture

DEUCALION

A64FX (Fujitsu)



Grace-Hopper (NVidia)



Graviton (AWS)

ARM chips share the same architecture and Instruction Set, meaning that efforts to optimize for a chip are portable to others

Take full advantage of Deucalion!

DEUCALION

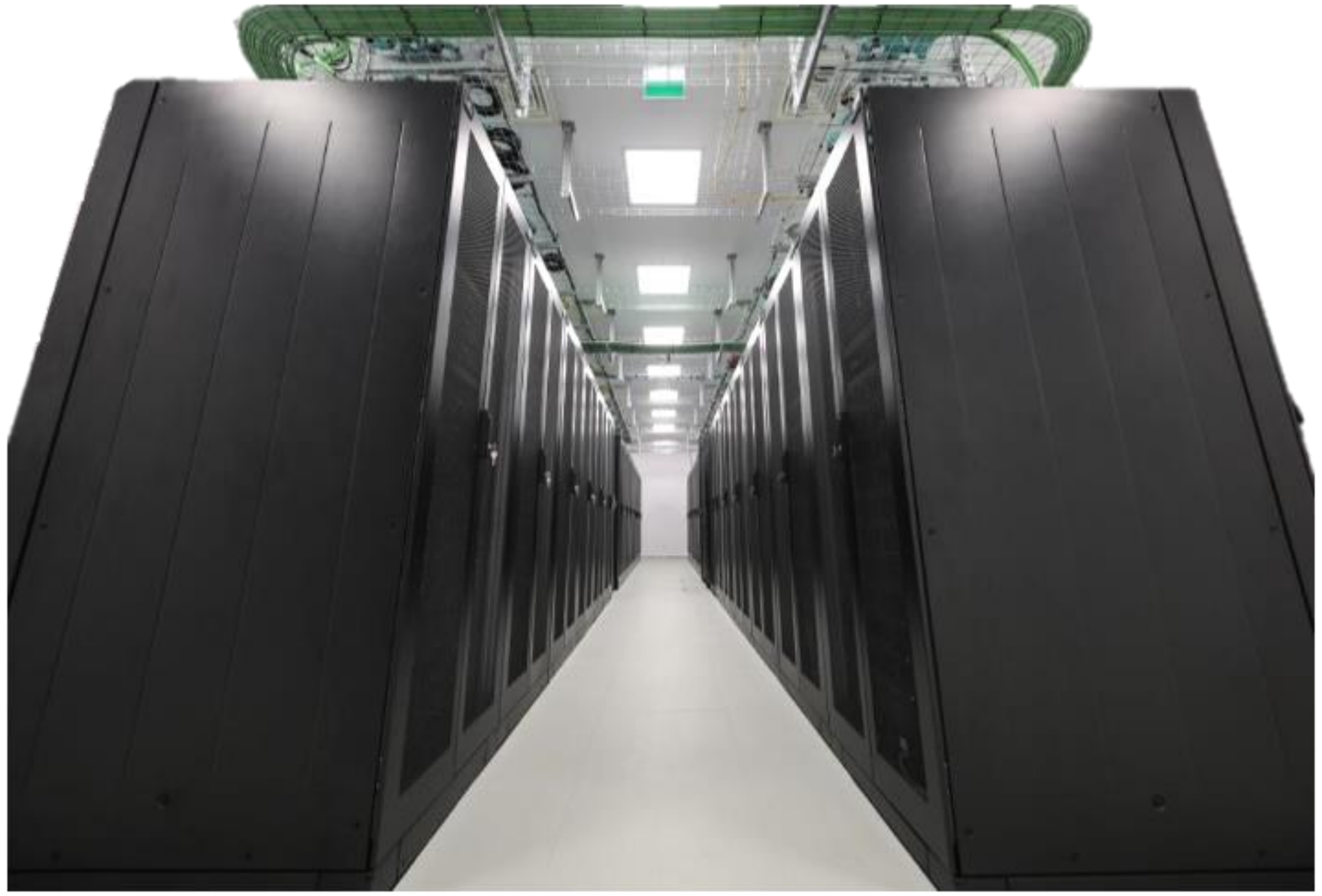
ARM partition

1632 nodes

A64FX chip

32 GB High bandwidth HBM2 RAM (50% faster)

Access to optimized software



X86 partitions

500 nodes + 33 GPU nodes

2 x AMD EPYC 7742 per node (128 cores)

256 GB RAM

Access to optimized software

Porting your code to ARM lets you have access to the largest partition in Deucalion

Portability

DEUCALION

Compilation

Can you compile your code?

Do you need new toolchains for the ARM architecture?

Can you install every dependency (HDF5, other libraries, etc.)

arm

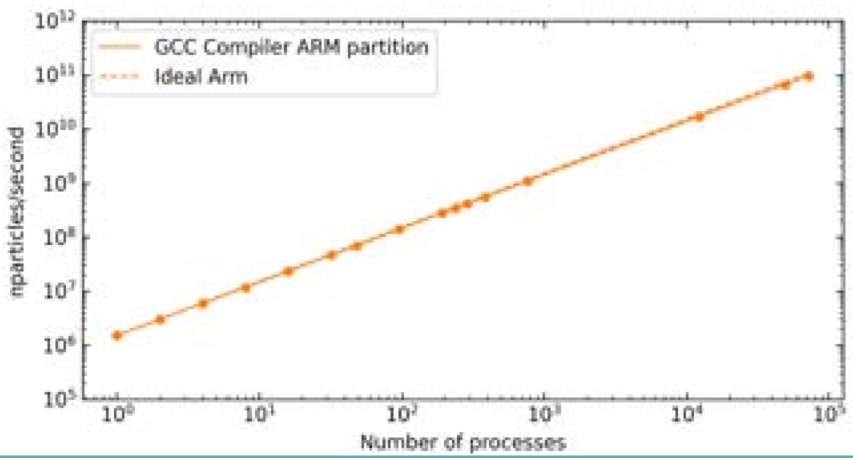
GCC

Running

Can you run your code?

Can you run it as fast as in other architectures (big focus on vectorization)?

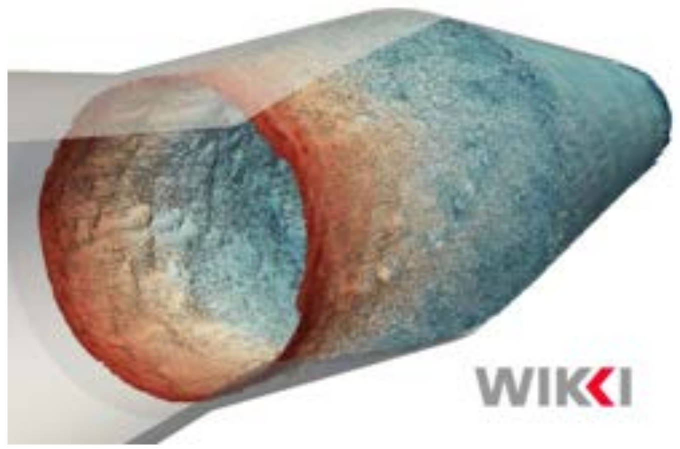
Does it scale efficiently?



Get the same results

Do you get precise bitwise reproducibility?

But...are the results comparable between different (or sometimes even the same) architectures?



Compilation is an easy step

DEUCALION

A lot of applications have ARM-ready versions (OpenFOAM, HDF5, ScaLAPACK, Eigen, FFTW, GROMACS, etc.)

More than 500 modules now available in the ARM partition

Be reassured! You will be able to run your application in our ARM partition ☺

But specific applications make it harder than others

Specific issues that might arise

DEUCALION

Select list of possible issues

- Inline assembly with no corresponding aarch64 inline assembly

Example

```
/*main.c*/

int src = 1;
int dst;

asm ("mov %1, %0\n\t"
    "add $1, %0"
    : "=r" (dst)
    : "r" (src));

printf("%d\n", dst);
```

Specific issues that might arise

DEUCALION

Select list of possible issues

- Inline assembly with no corresponding aarch64 inline assembly
- Assembly source files with no corresponding aarch64 files
- Missing aarch64 architecture detection in autoconf config.guess scripts, etc.
- Linking against libraries that are not available on the aarch64 architecture
- Use of architecture specific intrinsics (more on that later)

Example

```
/*main.c*/

/*This does not exist for ARM
chips!*/

#include <immintrin.h>
```

Specific issues that might arise

DEUCALION

Select list of possible issues

- Inline assembly with no corresponding aarch64 inline assembly
- Assembly source files with no corresponding aarch64 files
- Missing aarch64 architecture detection in autoconf config.guess scripts, etc.
- Linking against libraries that are not available on the aarch64 architecture
- Use of architecture-specific intrinsics (more on that later)
- Preprocessor errors that trigger when compiling on aarch64
- Compiler specific code guarded by compiler specific pre-defined macros

Example

```
/*main.c*/
/*This does not support a Fujitsu
compiler!*/

#if defined(__GNUC__)
/* gcc /*
  #define VAR A
#if defined (__INTEL_LLVM_COMPILER)
  /* Intel icc */
  #define VAR B
#else
  #error Not supported!
#endif
```

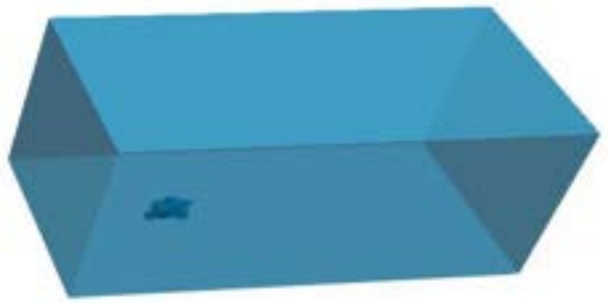
GCC or Fujitsu? It depends

DEUCALION

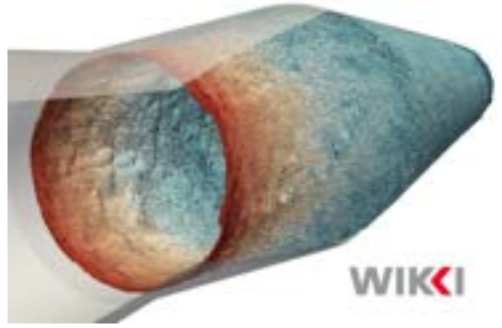
A lot of effort has been put into open-source compilers (e.g., GNU's gcc) so they can match well with proprietary compilers (Fujitsu's fcc)

OpenFOAM

Small motorbike



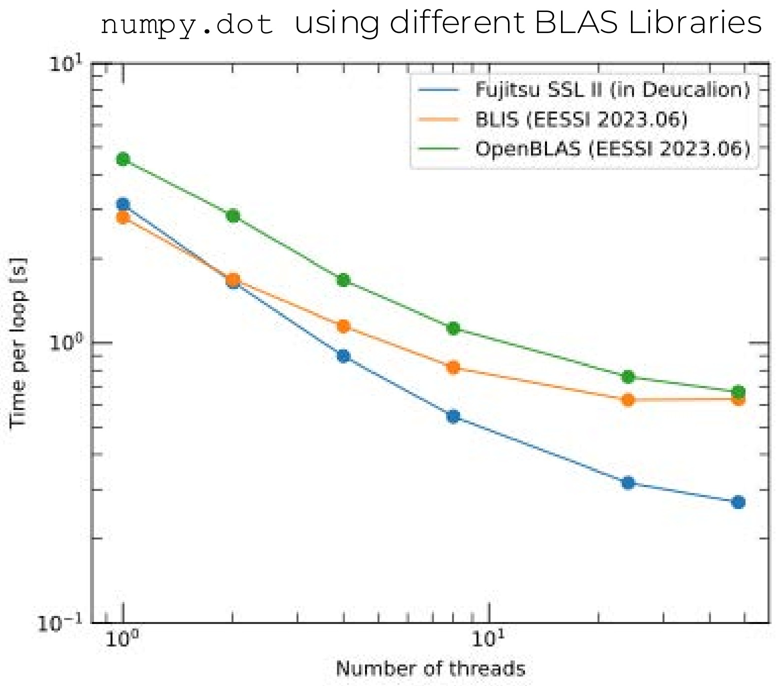
Conical Diffuser



GCC grasps **97%** of the performance of Fujitsu's FCC

Gabriel Marcos Magalhães, OpenFOAM Iberia (2024)

BLAS Libraries



Fujitsu Optimized BLAS still outperforms **(150% speedup)** other libraries

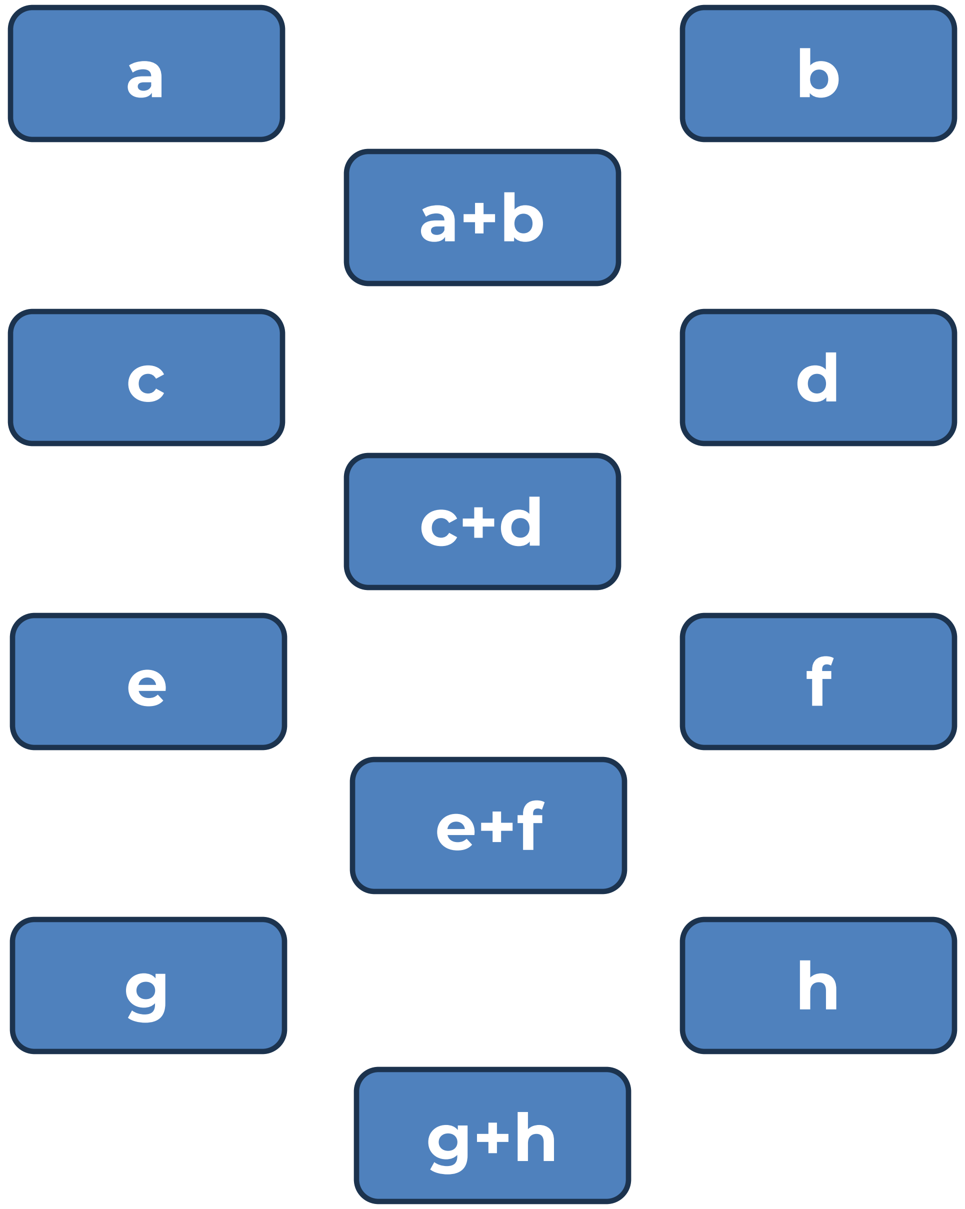
On a64fx, single-threaded BLIS outperforms OpenBLAS and is even faster than Fujitsu's!

Miguel Dias Costa (2024)

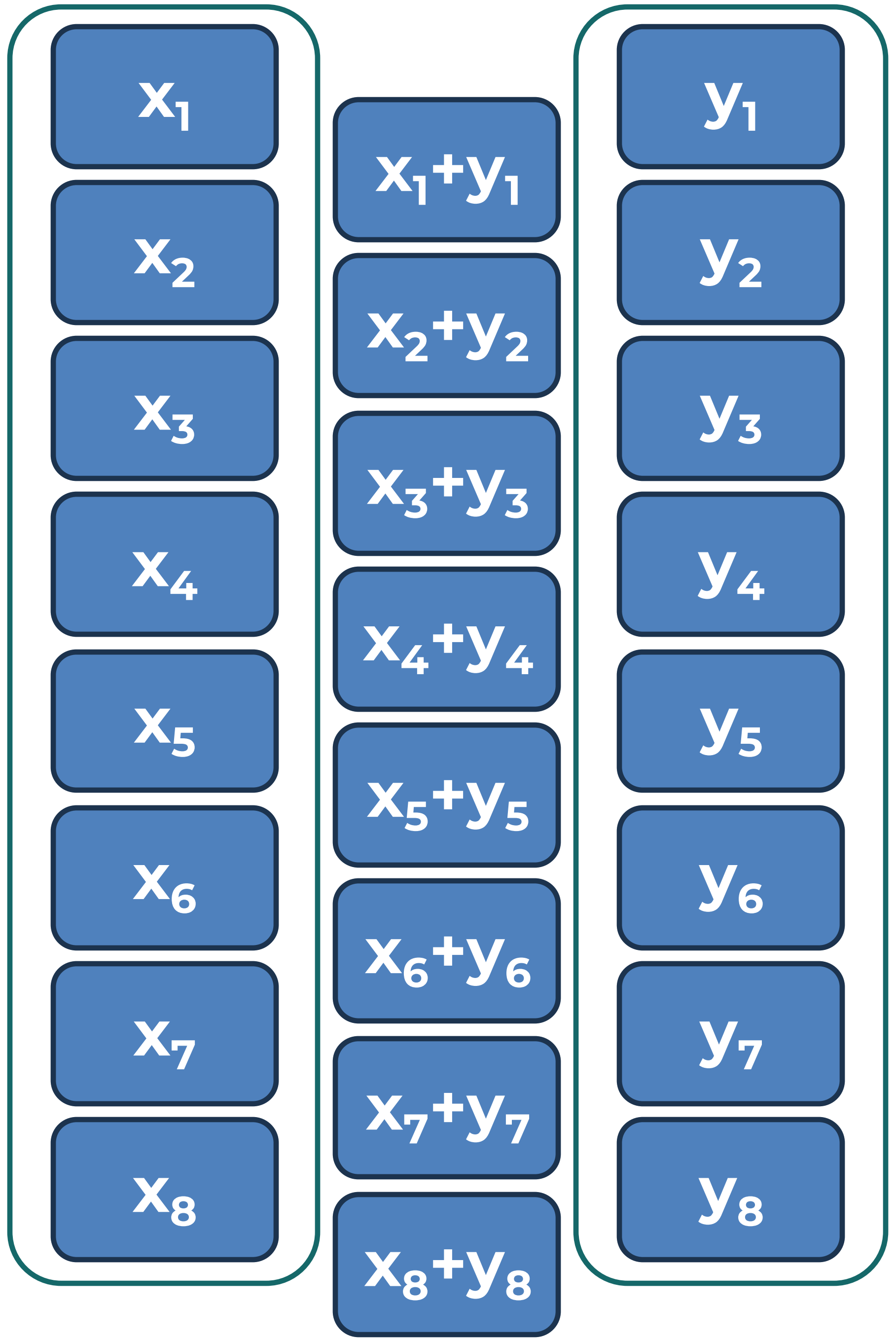
For 99% of modules, there is not a significant performance increase. We maintain and support software stacks that benefit from using Fujitsu's toolchain

Vectorization with diagrams

Scalar Registers
(64 bits)



Vector Registers
(512 bits)



Vectorization with diagrams

DEUCALION

Scalar Registers (64 bits)

a+b

c+d

e+f

g+h

ARM vs Intel

The A64FX chip has a bigger instruction latency than the x86 equivalent but a larger vector register (512 compared with 256 bits)

Vector instructions (SIMD: Single Instruction Multiple Data) are individually slower to compute but get better overall throughput

Some ARM considerations

The A64FX supports both ARM Neon and SVE instructions. Neon only supports **128-bit** vectors. SVE supports different-sized vectors **(up to 2048 bits – A64FX has 512 bits)**

Most of the ARM chips support Neon, but only a few support SVE (including A64FX)

Vector Registers (512 bits)

x_1+y_1

x_2+y_2

x_3+y_3

x_4+y_4

x_5+y_5

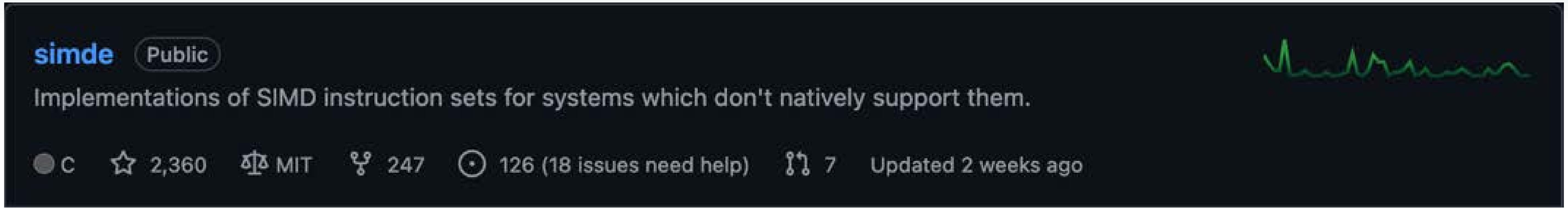
x_6+y_6

x_7+y_7

x_8+y_8

Accessing Neon (128 bits) in ARM

DEUCALION



SIMDe allows to easily implement Neon vectorization from intel intrinsics

Optimized code for SSE (Intel)

```
/*header file*/

/* SSE definitions */
#include <xmmintrin.h>
#include <pmmintrin.h>
```

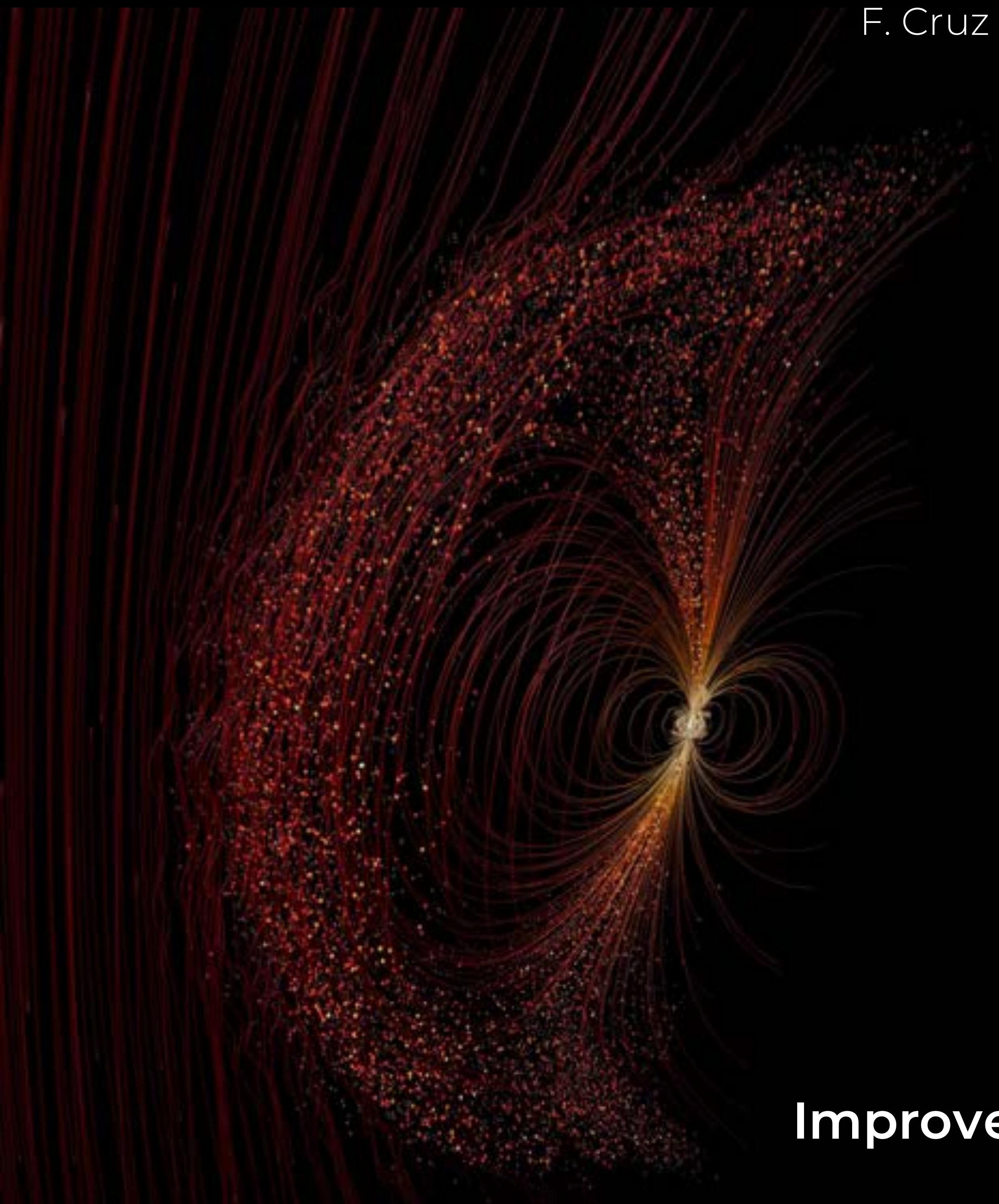
Optimized code for ARM

```
/*header file*/

/*Natively substitute every Intel
instruction*/
#ifndef SIMDE_ENABLE_NATIVE_ALIASES
#define SIMDE_ENABLE_NATIVE_ALIASES
#endif

#include "simde/simde/x86/sse.h"
#include "simde/simde/x86/sse3.h"
```

If you already implemented SIMD code for x86 architectures, you could easily port it to ARM Neon

Example: OSIRIS (more on this later)

DEUCALION



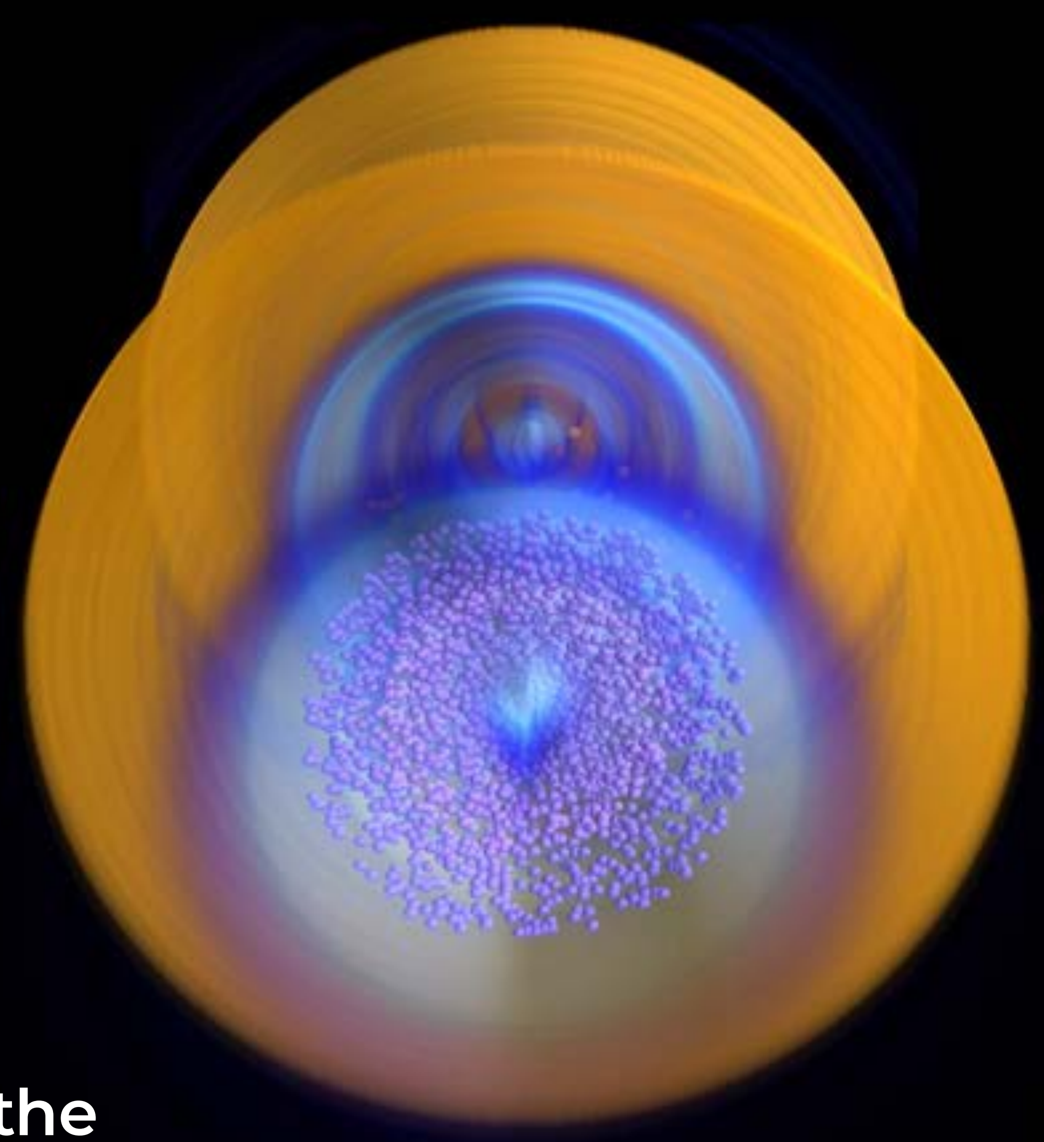
F. Cruz

OSIRIS

OSIRIS is a well-known open-source code for plasma physics, with a large user base

Lack of native vectorization for ARM overcome using SIMDe

Multiple OSIRIS-based projects running on ARM at Deucalion



B. Malaca

Improvement of 8-20% by using SIMDe directly on the source code!

Improvements after SVE (512 bits)

DEUCALION

GROMACS supports compilation with both Neon and SVE vectorization

With more general Neon vectorization

User example

```
                    Core t (s)   Wall t (s)        (%)
       Time:          8184.525     170.513      4799.9
                      (ns/day)    (hour/ns)
Performance:            50.681       0.474
```

water-cut1.0_GMX50_bare benchmark

```
                    Core t (s)   Wall t (s)        (%)
       Time:          4940.566     102.930      4799.9
                      (ns/day)    (hour/ns)
Performance:             0.841      28.535
```

With specific SVE vectorization

User example

```
                    Core t (s)   Wall t (s)        (%)
       Time:          6381.780     132.957      4799.9
                      (ns/day)    (hour/ns)
Performance:            64.996       0.369
```

water-cut1.0_GMX50_bare benchmark

```
                    Core t (s)   Wall t (s)        (%)
       Time:          2903.855      60.499      4799.8
                      (ns/day)    (hour/ns)
Performance:             1.431      16.772
```

Even after getting Neon to work, you can expect tens of percent speedup after supporting SVE in our ARM partition

Other tips for efficient A64FX use

DEUCALION

Unroll and interleave leads to more instructions per clock cycle

Before

```
for(int i=0; i<N; i++) {
  int c = a[i]; // load
  c = c + 1; // compute
  c = c + 2; // compute
  c = c + 3; // compute
  c = c + 4; // compute
  a[i] = c; // store
}
```

After unrolling

```
for(int i=0; i<N-1; i+=2) {
  int c0 = a[i];
  c0 = c0 + 1;
  c0 = c0 + 2;
  c0 = c0 + 3;
  c0 = c0 + 4;
  a[i] = c0;
  int c1 = a[i+1];
  c1 = c1 + 1;
  c1 = c1 + 2;
  c1 = c1 + 3;
  c1 = c1 + 4;
  a[i+1] = c1;
}
```

Unrolling loops does not automatically lead to better performance

Other tips for efficient A64FX use

DEUCALION

Unroll and interleave leads to more instructions per clock cycle

Before

```
for(int i=0; i<N; i++) {
  int c = a[i]; // load
  c = c + 1; // compute
  c = c + 2; // compute
  c = c + 3; // compute
  c = c + 4; // compute
  a[i] = c; // store
}
```

After unrolling and interleaving

```
for(int i=0; i<N-1; i+=2) {
  int c0 = a[i];
  int c1 = a[i+1];
  c0 = c0 + 1;
  c1 = c1 + 1;
  c0 = c0 + 2;
  c1 = c1 + 2;
  c0 = c0 + 3;
  c1 = c1 + 3;
  c0 = c0 + 4;
  c1 = c1 + 4;
  a[i] = c0;
  a[i+1] = c1;
}
```

Unrolling loops and interleaving instructions tends to improve performance quite a bit

Other tips for efficient A64FX use

DEUCALION

Speedups with little effort

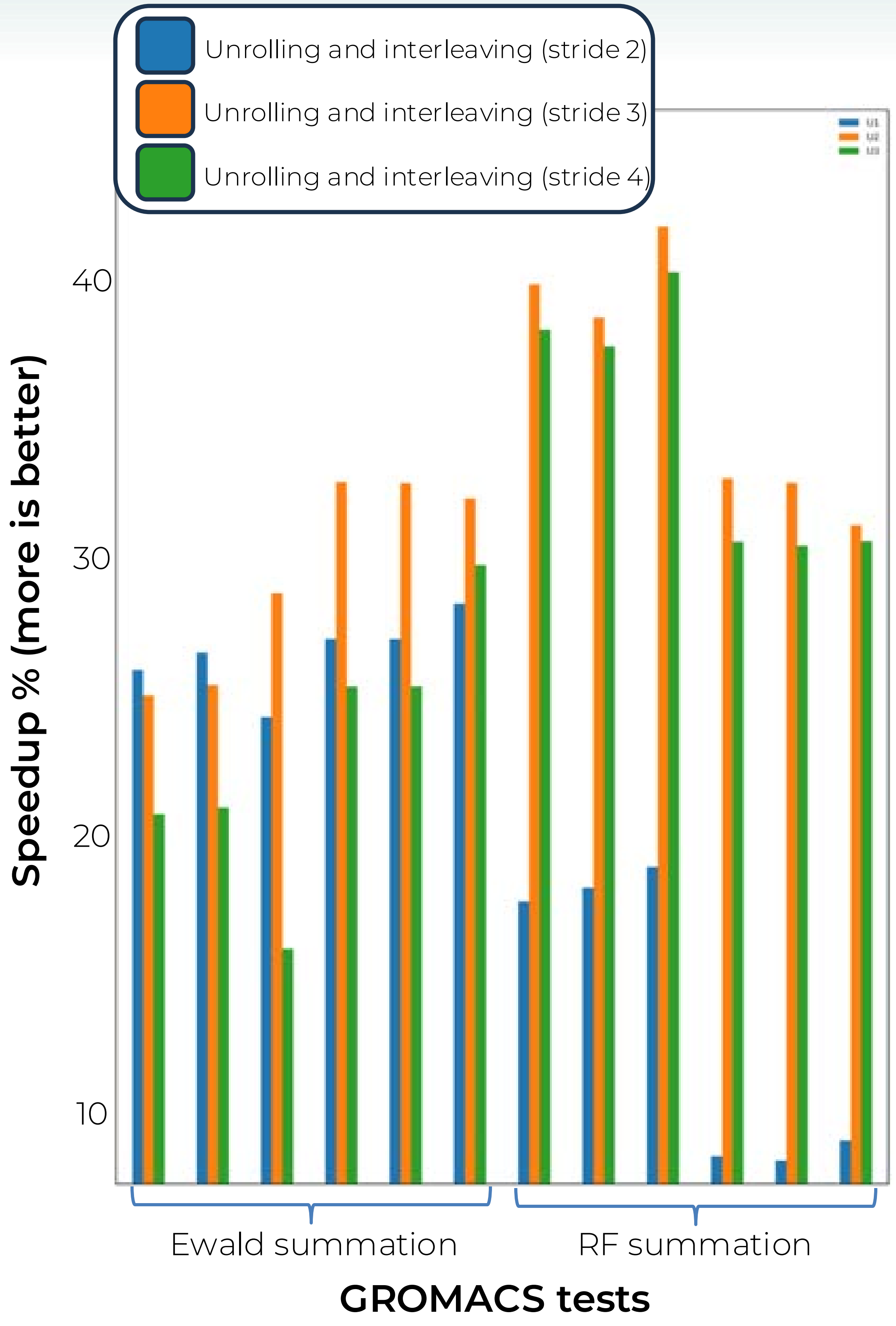
In some use cases the speedup for unrolling (stride=3) and interleaving was about 30% (mostly from out-of-order execution)

Other optimization studies refer that in nested loops you should have a bigger inner loop.

It takes a village

You can get a list of a lot of different optimizations performed by HPC users at:
https://www.hpci-office.jp/en/events/seminars

Last one on 27th November about LAMMPS

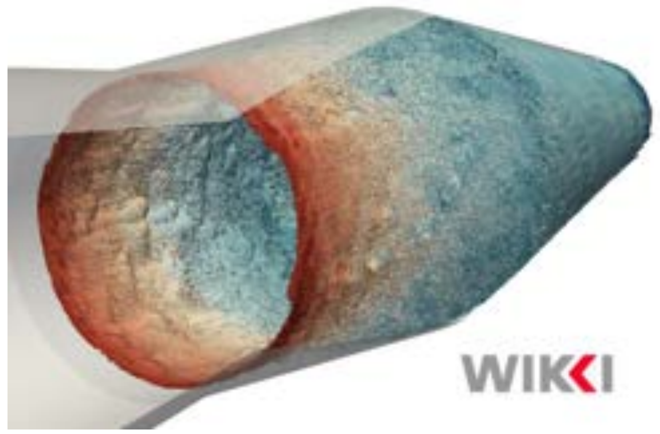


*Gilles Gouaillardet, https://www.hpci-office.jp/documents/meeting_A64FX/220727/GROMACS_A64fx.pdf

Comparison with x86

All comparisons are made with the same number of cores. Even though memory access is faster with the A64FX, **the clock speed of the x86 is 70% faster.**

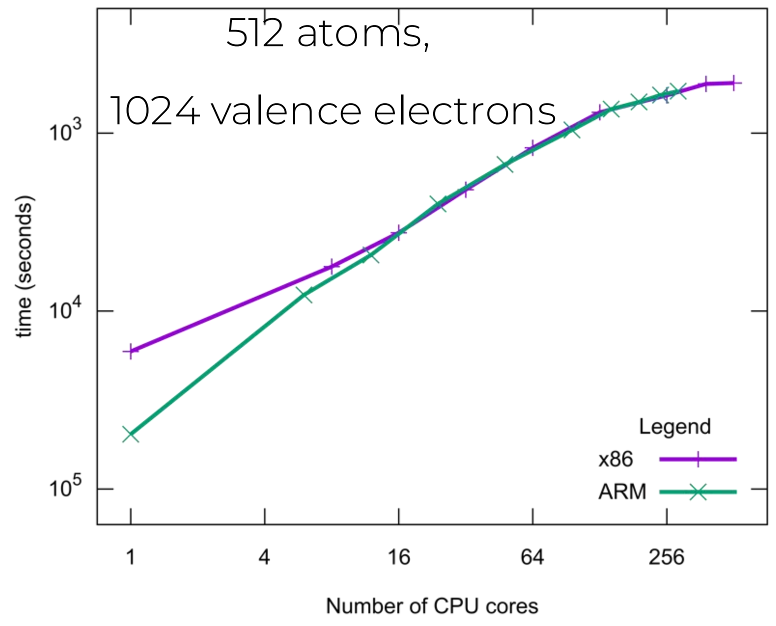
OpenFOAM (CFD)



X86 is 1.4x faster than ARM (both are not vectorized)

Gabriel Marcos Magalhães, UMinho

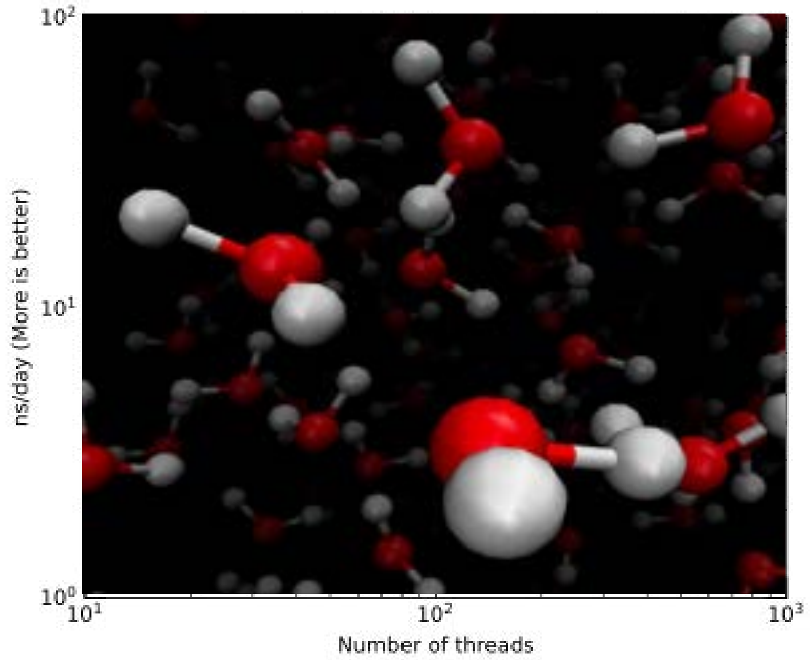
VASP (DFT)



ARM and x86 have the same performance

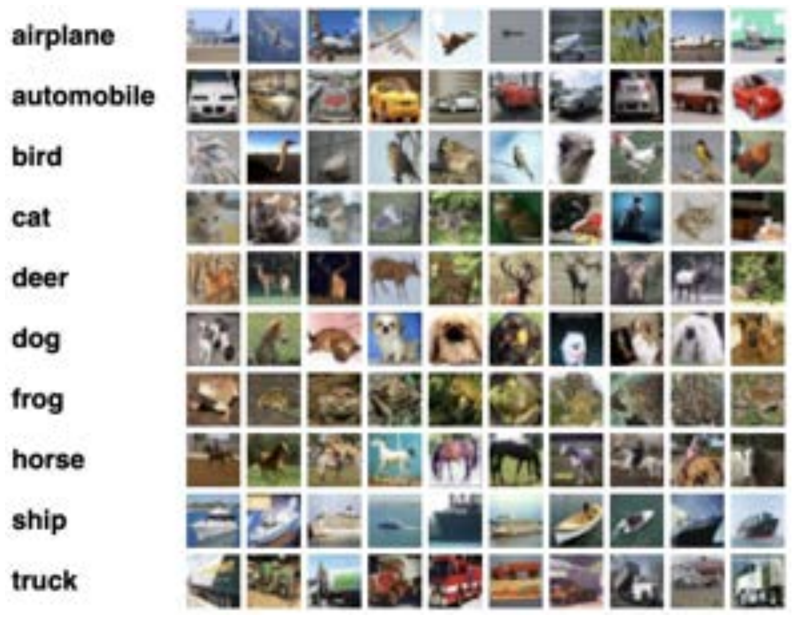
José Coutinho, Universidade Aveiro

GROMACS (MD)



X86 is 2.4x faster than ARM

Pytorch (AI)



ARM 2-10x faster than x86. 1 GPU is equivalent to 15-20 ARM nodes

We routinely see **2-3x slower per-core performance on the ARM nodes**, with better performances in memory-bound, optimized codes (particularly in AI)

Useful communities

DEUCALION

EasyBuild.io
building software with ease

EESSI
EUROPEAN ENVIRONMENT FOR SCIENTIFIC SOFTWARE INSTALLATIONS

EasyBuild

Simpler way of installing scientific software with the proper flags

Automatic creation of modules, able to install several versions of the same software easily

EESSI

Streams scientific code directly to any machine

User can use a code without having any knowledge about the specific hardware architecture

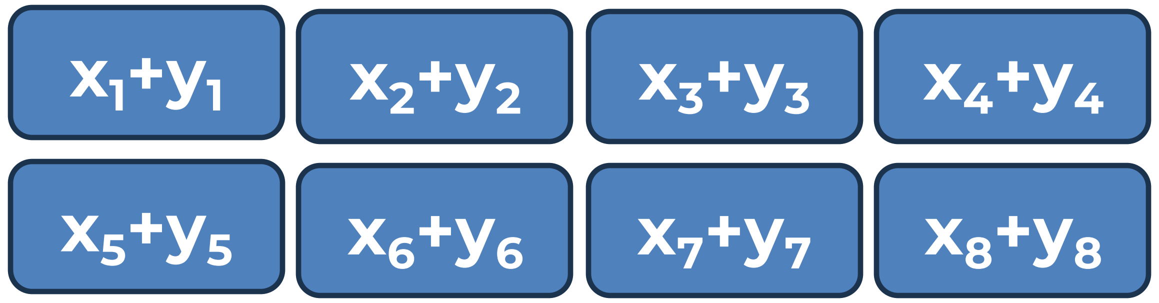
More than tools, these are excellent (and active!) communities that you should take advantage of!

Conclusions

DEUCALION

Use all of Deucalion!

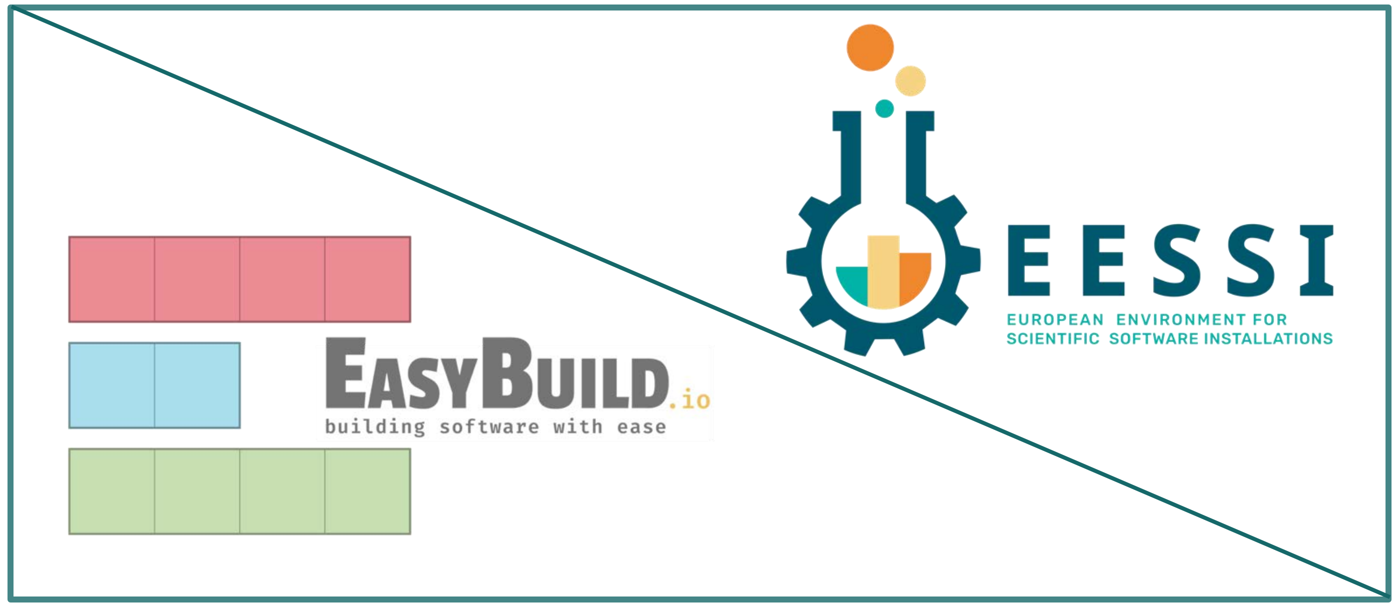
Vectorization is the biggest challenge



Some tricks to get ARM vectorization

```
/*Natively substitute every Intel
instruction*/
#ifndef SIMDE_ENABLE_NATIVE_ALIASES
#define SIMDE_ENABLE_NATIVE_ALIASES
#endif
```

It takes a village!



Contact us!

https://www.macc.fccn.pt

https://x.com/minhoacc

https://www.linkedin.com/company/minhoacc

Co-funded by the European Union

This project has received funding from the High Performance Computing Joint Undertaking under grant agreement No 101139786

DEUCALION

Co-funded by the European Union

This project has received funding from the High Performance Computing Joint Undertaking under grant agreement No 101139786