EPICURE
Unlocking European-level HPC Support

An introduction to Arm Scalable Vector Extensions

Ricardo A. Fonseca[1,2]

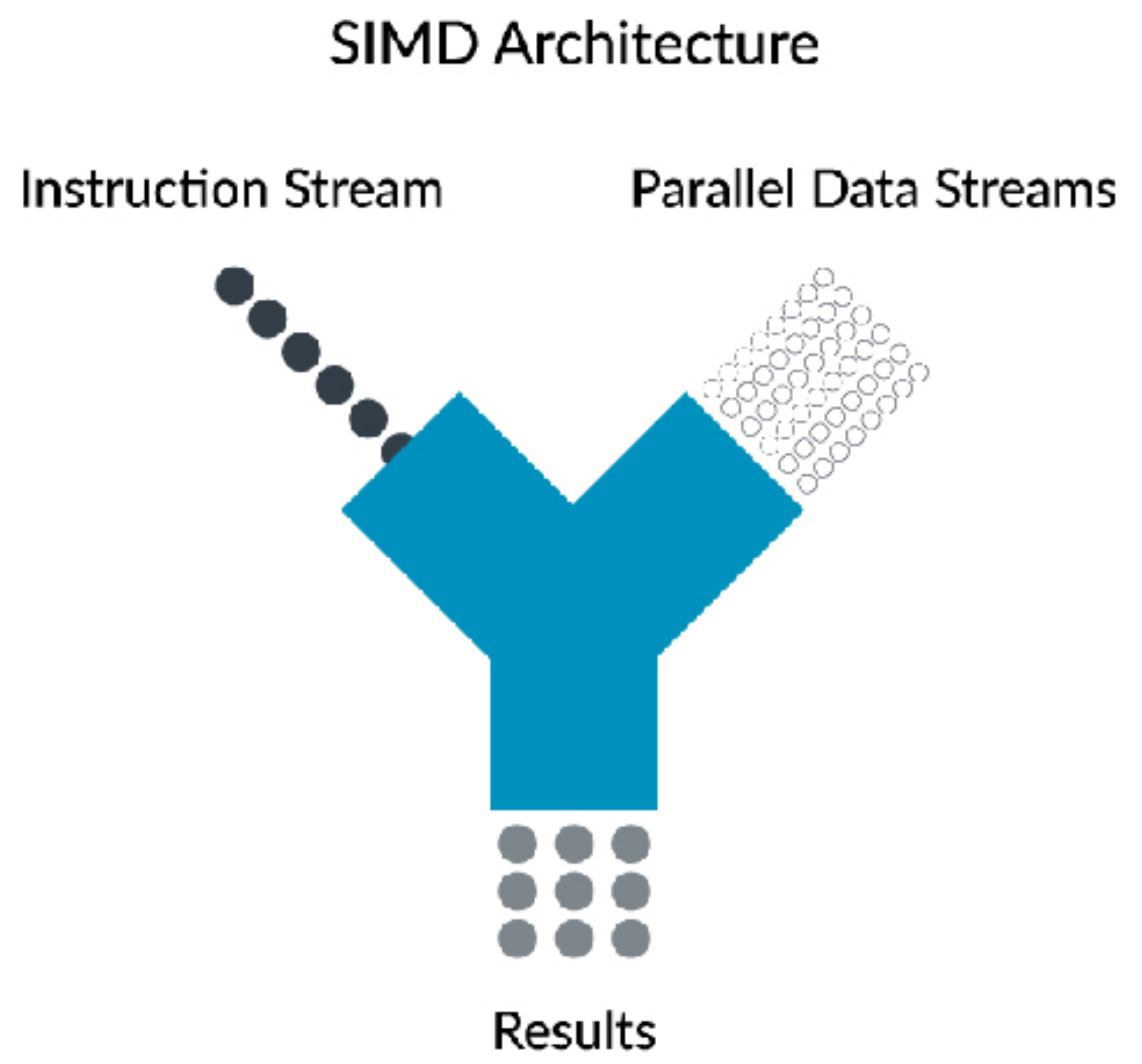
[1] GoLP/IPFN, Instituto Superior Técnico, Lisboa, Portugal

[2] DCTI, ISCTE-Instituto Universitário de Lisboa, Portugal

golp

iscte INSTITUTO UNIVERSITÁRIO DE LISBOA

Improving performance with SIMD code



Single Instruction / Multiple Data (SIMD)

- Modern CPUs include a SIMD vector unit
- Vector registers: (n × float/int values)
- Instructions act on vector registers
- Same operation on n different values simultaneously
- **Up to n× speedup from serial code**

From x86 to Arm

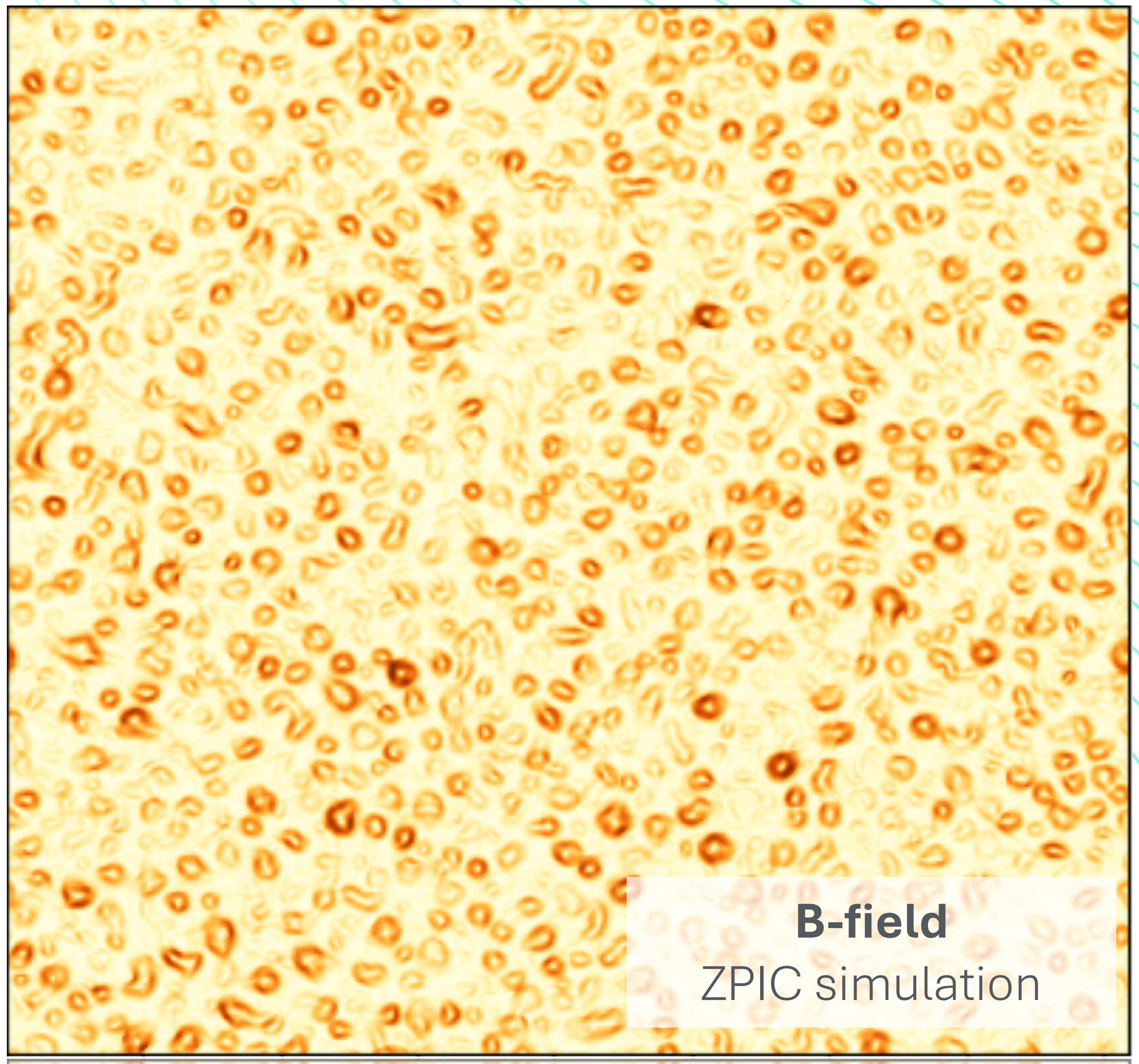
- Current x86 architectures include AVX2 / AVX512
- **ARM v8-A features Scalable Vector Instructions - SVE**

Outline

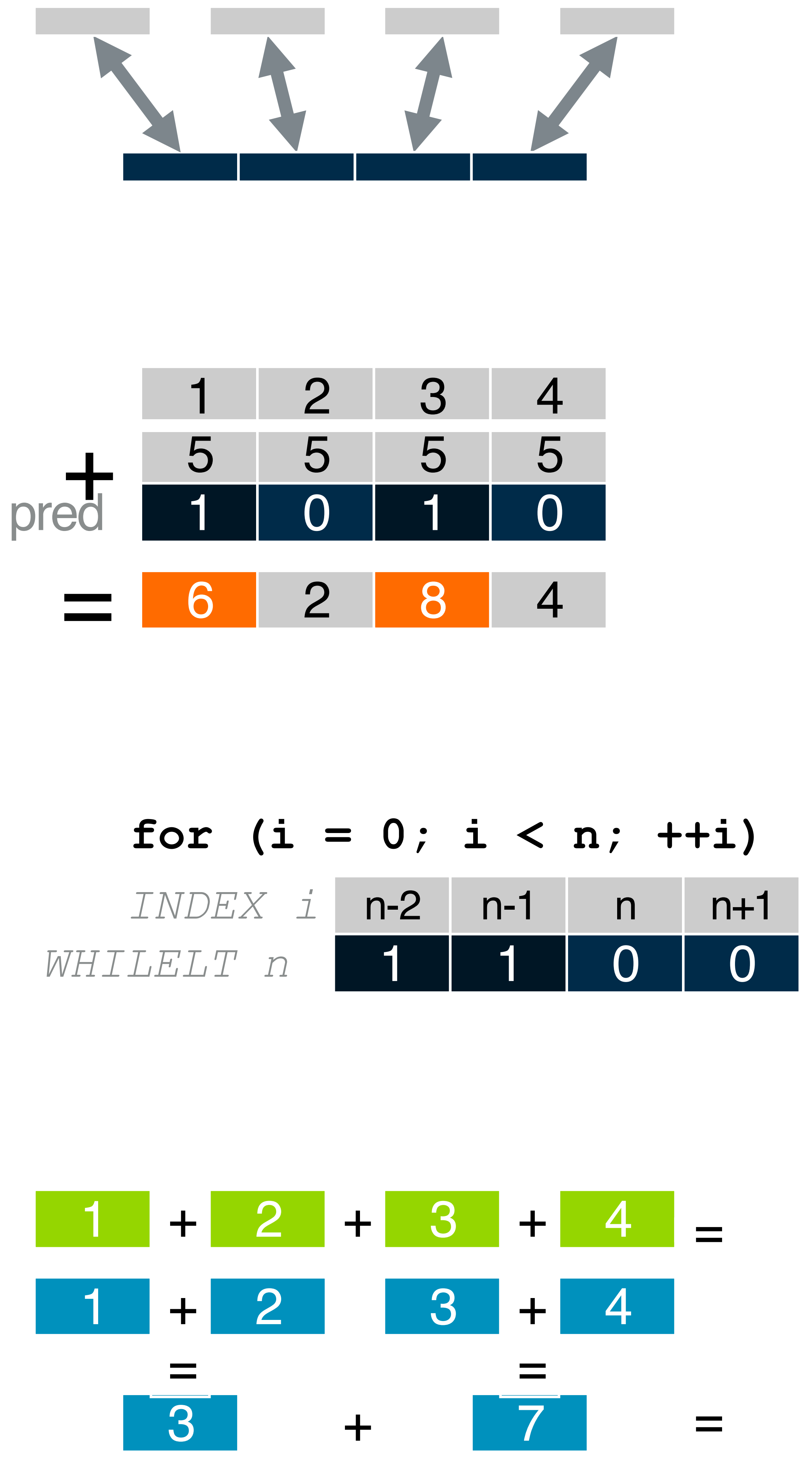
- **Introduction to Arm Scalable Vector Extensions**
  - SVE and Vector Length Agnostic programming
  - Using SVE without intrinsics
  - Using ACLE
  - Data types and operations

- **Migrating from other SIMD architectures**
  - Main issues
  - Case study - the ZPIC code

- **Overview**

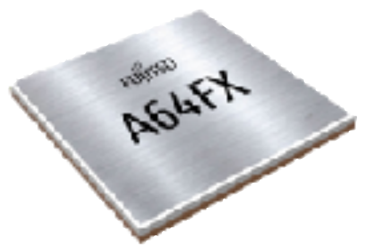


What are the Scalable Vector Extensions (SVE)?



- **Vector extension to the ARM v8-A Architecture**
  - Standard SIMD operations, plus...
  - Gather-load and scatter store operations
  - Per-lane prediction
  - Predicate-driven loop control and management
  - Extended floating-point horizontal reductions

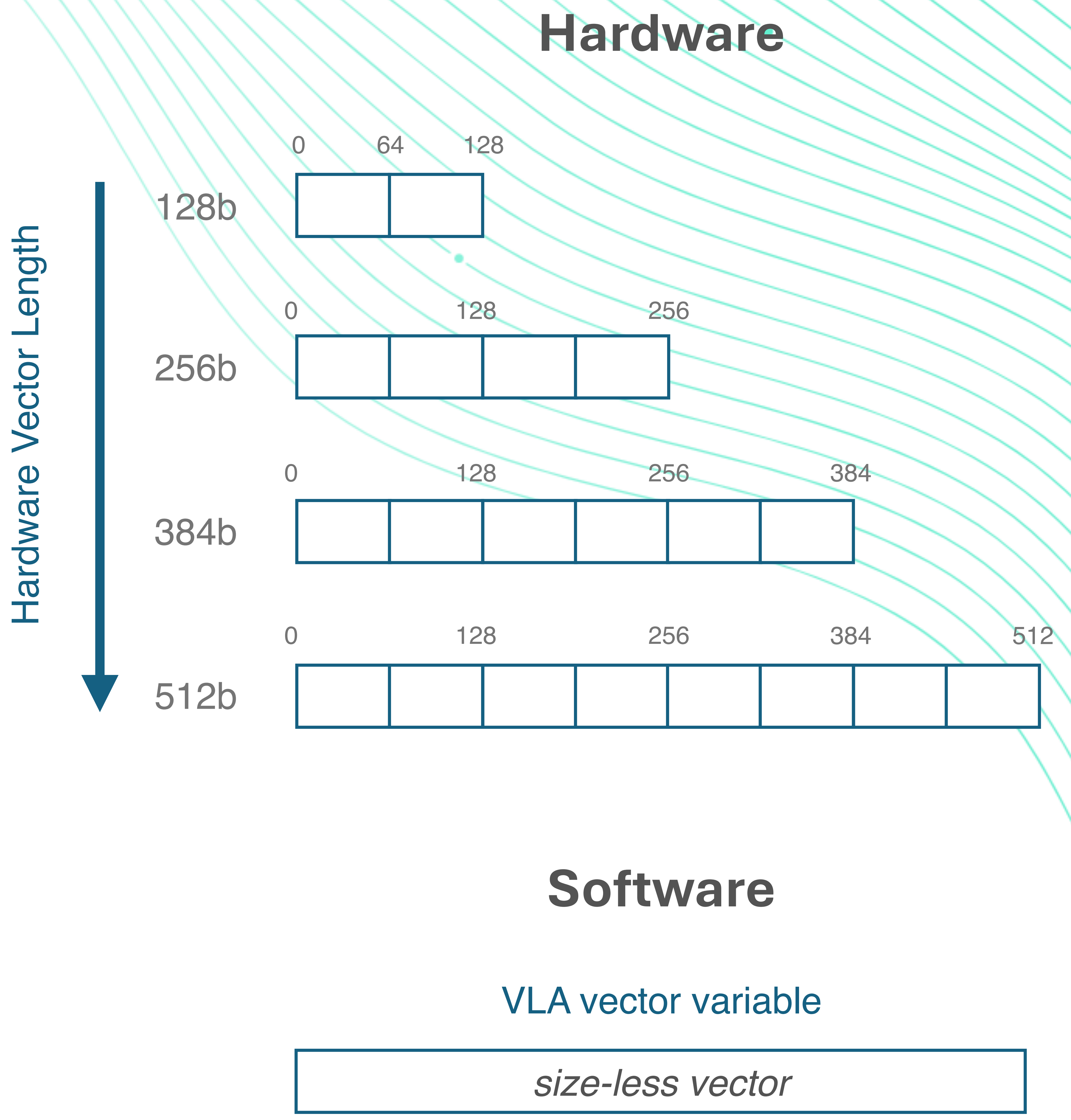
- **Supported by Deucalion Arm partition**
  - Fujitsu A64FX CPUs
  - 4 NUMA nodes with 12 cores each, 48 cores total
  - 512 bit SVE units



What is the vector length?

- **Unlike other architectures, SVE has no preferred vector length (VL)**
  - VL will depend on the ARM CPU type, ranging from 128 to 2048
  - A64FX supports 128, 256 and 512 bits

- **Code can be written to be Vector Length Agnostic (VLA)**
  - In software, vectors have no length
  - Portable across all possible VL
  - No need to recompile or rewrite code



How can I use it?

- **Compilers**
  - Auto-vectorization: GCC, clang, Fujitsu
    - Usually enabled at higher optimization levels
    - Check specific compiler flags
  - OpenMP SIMD directives
    - `#pragma omp parallel for simd`
  - Compiler directives (e.g. clang)
    - `#pragma clang loop vectorize(enable)`

- **Libraries**
  - Arm Performance Library (ArmPL)
  - Fujitsu SSL II
  - etc.

- **Intrinsics**
  - Arm C Language Extensions (ACLE) for SVE

Programming with ACLE

- **ACLE extends C/C++ with Arm-specific features**
  - Predefined macros
  - Data types
  - Intrinsic functions

- **Use with SVE code**
  - Nearly 1 intrinsic per SVE instructions
  - Include the relevant header

    `#include “arm_sve.h”`
  - Set the appropriate compiler target

    `-march=armv8-a+sve`

- **Intended for developers wanting to**
  - Hand-tune SVE code
  - Adapt or hand-optimize applications and libraries
  - Access low-level Arm features

```c
// Compile with -march=armv8-a+sve
#include "arm_sve.h"

// Serial version
void mul2(float * a, int N ) {
    for( int i = 0; i < N; i++ ) {
        a[i] *= 2.0;
    }
}

// SVE version
void vmul2(float * a, int N ) {
    for( int i = 0; i < N; i+= svcntw() ) {
        svbool_t pred = svwhilelt_b32(i, N);
        svfloat32_t va = svld1(pred, &a[i]);
        va = svmul_x(pred, va, 2.0);
        svst1(pred, &a[i], va);
    }
}
```

SVE data types

```
#include "arm_sve.h"
```

- **SVE datatypes**
  - Represent the **size-less** vectors used for SVE intrinsics
  - Support for different integer/floating-point bit widths

- **Integer types**
  - `svint8_t, svint16_t, svint32_t, svint64_t`
  - `svuint8_t, svuint16_t, svuint32_t, svuint64_t`

- **Floating-point types**
  - `svfloat32_t, svfloat64_t`

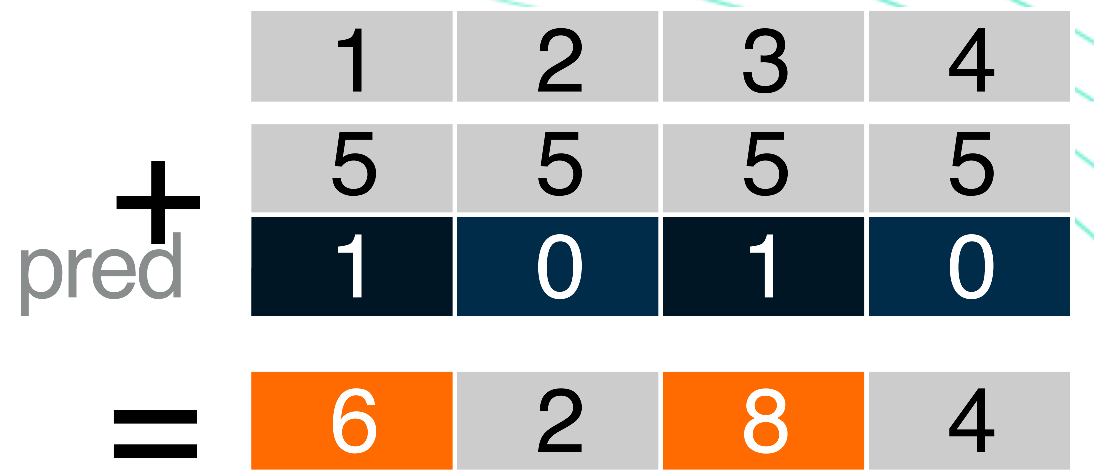
- **Predicate (mask) types**
  - `svbool_t`

```
void vmul2(float * a, int N ) {
    for( int i = 0; i < N; i+= svcntw() ) {
        svbool_t pred = svwhilelt_b32(i, N);
        svfloat32_t va = svld1(pred, &a[i]);
        va = svmul_x(pred, va, 2.0);
        svst1(pred, &a[i], va);
    }
}
```

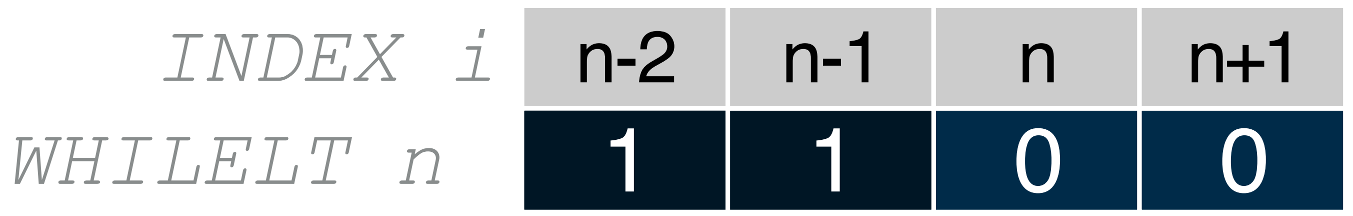
What are Predicates?

- **All SVE instructions include a *predicate***
  - Also know as ***masks*** on other architectures
  - Indicates which lanes (vector elements) are active in the operation
- **Used for per-lane predication**
  - Operation is only applied to specific vector elements
  - e.g. implement `(t)?a:b` vectorially
- **Used for vectorizing loops that are not a multiple of VL size**
  - The predicate elements will be all true for initial iterations
  - Last iteration only acts on the values inside the loop range
- **Specific instructions for predicate initialization**
  - `svptrue_b32()` - Initialize all lanes to true (for 32 bit lanes)
  - `svwhilelt_b32(i,N)` - Initialize lanes where `i+lane_id<N` to true, false otherwise

```
svfloat32_t svadd[_f32]_x(
   svbool_t pred,
   svfloat32_t op1,
   svfloat32_t op2
)
```




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



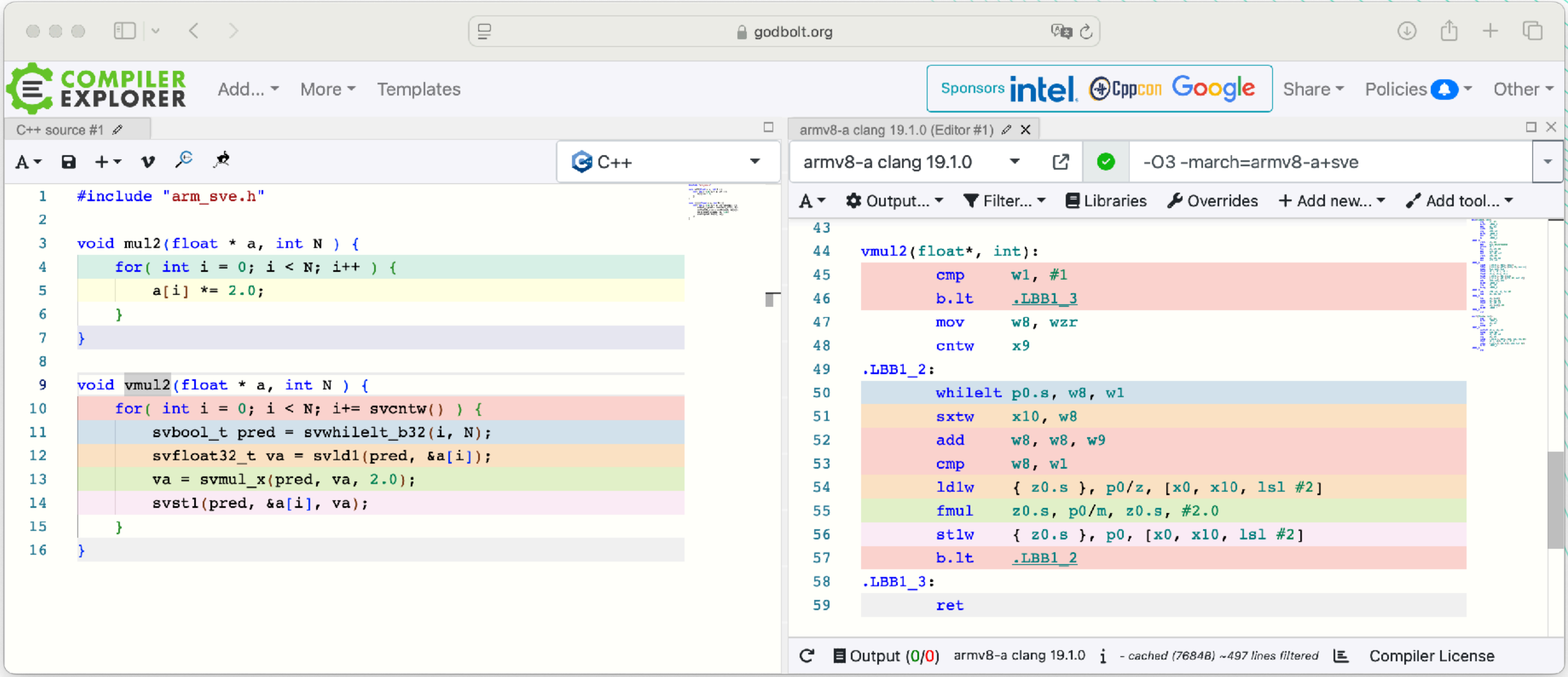
Annotated example - a[:] = 2.0 * a[:]

```
1 // Compile with -march=armv8-a+sve
2 #include "arm_sve.h"
3
4 // Serial version
5 void mul2(float * a, int N ) {
6     for( int i = 0; i < N; i++ ) {
7         a[i] *= 2.0;
8     }
9 }
10
11 // SVE version
12 void vmul2(float * a, int N ) {
13     for( int i = 0; i < N; i+= svcntw() ) {
14         svbool_t pred = svwhilelt_b32(i, N);
15         svfloat32_t va = svld1(pred, &a[i]);
16         va = svmul_x(pred, va, 2.0);
17         svst1(pred, &a[i], va);
18     }
19 }
```

SVE version

- **line 13** - Advance iteration by the number of vector elements - `svcntw`()
- **line 14** - Set the predicate to all lanes from `i` to less than N
- **line 15** - Load `svfloat32_t` vector variable **va** from `&a[i]` using the predicate. This prevents reading from beyond the limit of array `a`.
- **line 16** - Multiply **va** by scalar value `2.0`. The inactive predicate lanes will not be changed.
- **line 17** - Store the result `va` at `&a[i]` using the predicate. This prevents writing beyond the limit of array `a`.

Try it in Compiler Explorer - https://godbolt.org



SVE intrinsic functions

https://developer.arm.com/architectures/instruction-sets/intrinsics

- **Example arithmetic operations**
  - Addition - `svfloat32_t svadd[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)`
  - Multiplication - `svfloat32_t svmul[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)`
  - Division - `svfloat32_t svdiv[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)`
  - Fused multiply-add - `svfloat32_t svmad[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)`

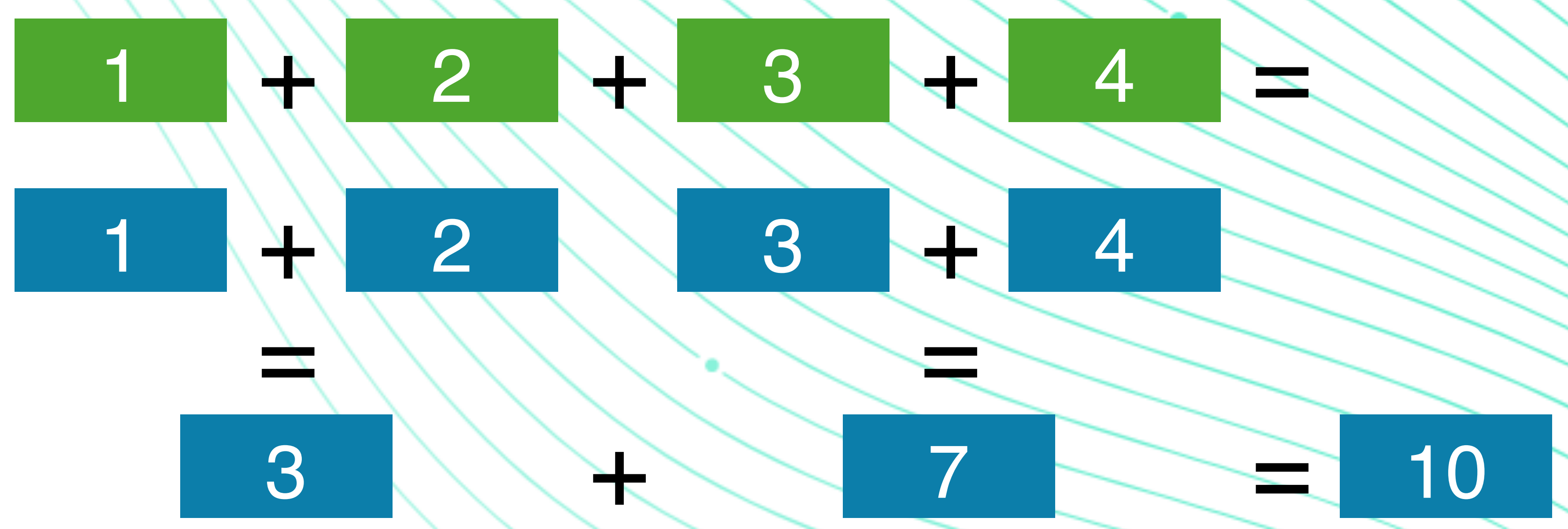
- **Example logical operations**
  - Less or equal - `svbool_t svcmple[_f32](svbool_t pg, svfloat32_t op1, svfloat32_t op2)`
  - Logical or - `svbool_t svorr[_b]_z(svbool_t pg, svbool_t op1, svbool_t op2)`

SVE horizontal reductions

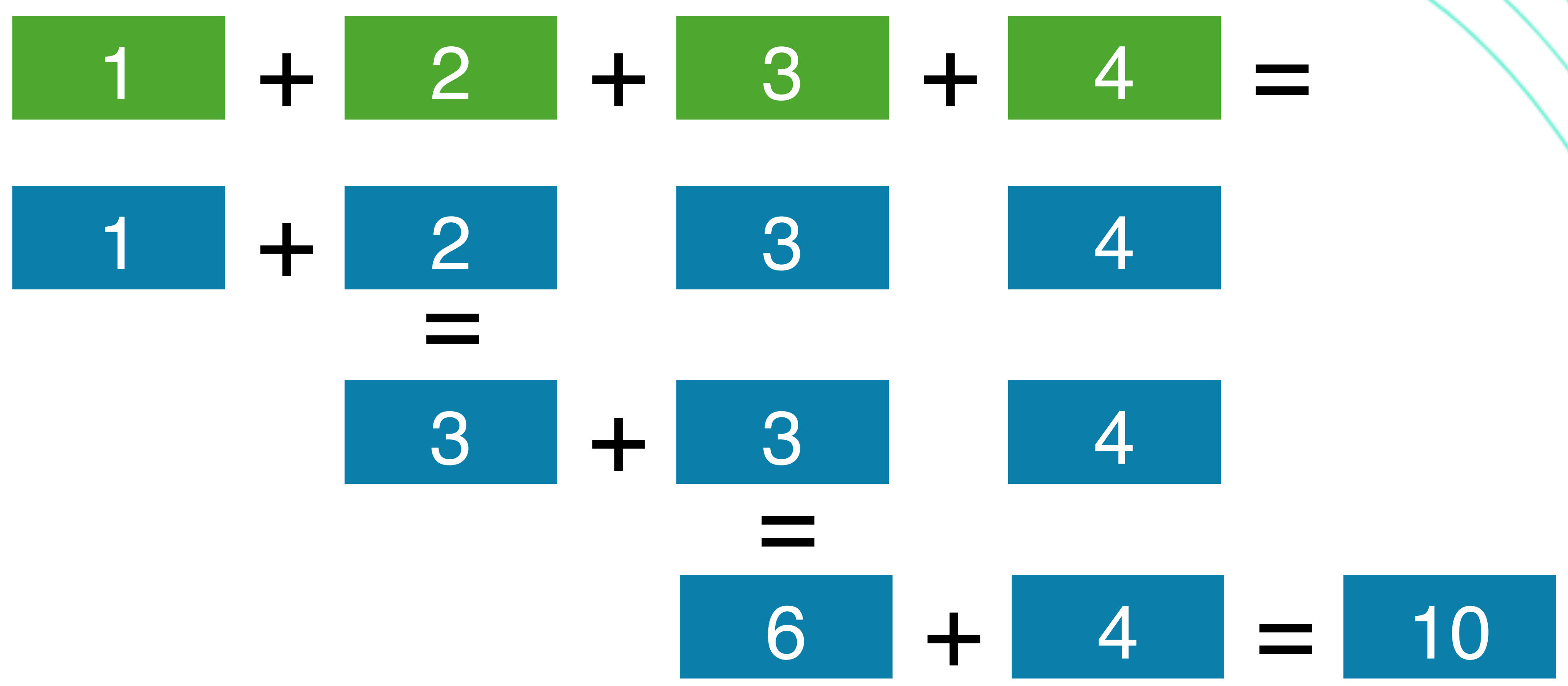
- **SVE operations that work across all lanes**
  - Combine all lane values into a scalar value
  - Addition, Maximum, Minimum

- **Two flavours for addition**
  - `svaddv_f32(svbool_t pg, svfloat32_t op)`
    - Tree-based algorithm
  - `svadda_f32(svbool_t pg, float32_t initial, svfloat32_t op)`
    - Add values sequentially

Tree Algorithm



Sequential Algorithm

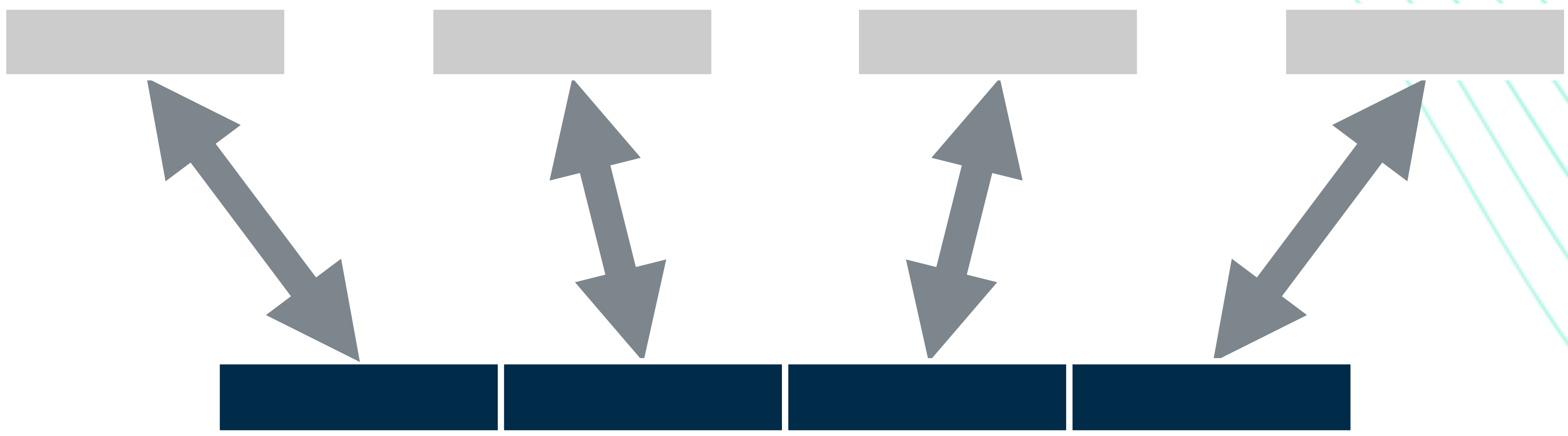


SVE memory access

- **Load / store consecutive scalar data**
  - Load - `svfloat32_t svld1[_f32](svbool_t pg, const float32_t *base)`
  - Store - `void svst1[_f32](svbool_t pg, float32_t *base, svfloat32_t data)`

- **Load / store consecutive 2 or 3 element structures**
  - This will de-interleave data on load / interleave data on store
  - Efficient, simple way to maintain compatibility with other data structures
  - Load - `svfloat32x2_t svld2[_f32](svbool_t pg, const float32_t *base)`
  - Store - `void svst2[_f32](svbool_t pg, float32_t *base, svfloat32x2_t data)`
  - `svfloat32x2_t` types can be managed with `svcreate2_f32()` / `svget2_f32()`

SVE gather / scatter operations



- **Gather operations**
  - Gather scalar values from array with indices specified by an integer vector
  - `svfloat32_t svld1_gather_[s32]index[_f32](svbool_t pg, const float32_t *base, svint32_t indices)`

- **Scatter operations**
  - Scatter scalar values from a vector to array with indices specified by an integer vector
  - `void svst1_scatter_[s32]index[_f32](svbool_t pg, float32_t *base, svint32_t indices, svfloat32_t data)`

Migrating from other SIMD architectures

- **Instructions**
  - Most instructions translate directly
  - Some are not available
    - **Implement equivalent ones**
  - Other operations exist on NEON/SVE but are missing from other SIMD architectures
    - **Replace code blocks with optimised versions**

- **Vector length**
  - Other SIMD architectures do not support VLA
  - VLA is not supported in some data structures
  - **You can set vector length at compile time**

Set vector length at compile time

- **While writing Vector Length Agnostic code can be quite useful, it may not fit all developer needs**
  - SIMD algorithms written for other architectures sometimes expect a fixed vector length
  - More importantly, you cannot use VLA dataypes inside data structures

- **You can, however, force the SVE code to a specific vector length**
  - Compile with `-msve-vector-bits=`*`nbits`*
  - The `__ARM_FEATURE_SVE_BITS` macro can be used to get *nbits*
  - Declare new types with `__attribute__((arm_sve_vector_bits(`*`nbits`*`)))`
  - The new types can be used with the standard SVE intrinsics

Set vector length at compile time - Example

```
// Compile with -msve-vector-bits=<nbits>
// where <nbits> can be 128, 256, 512, ...

#include <arm_sve.h>

constexpr int sve_vec_width = __ARM_FEATURE_SVE_BITS / 32;

typedef svfloat32_t vec_f32  __attribute__((arm_sve_vector_bits(__ARM_FEATURE_SVE_BITS)));
typedef svint32_t   vec_i32  __attribute__((arm_sve_vector_bits(__ARM_FEATURE_SVE_BITS)));
typedef svuint32_t  vec_u32  __attribute__((arm_sve_vector_bits(__ARM_FEATURE_SVE_BITS)));
typedef svbool_t    vec_mask __attribute__((arm_sve_vector_bits(__ARM_FEATURE_SVE_BITS)));

// Add 2 fixed-width SVE vectors
static inline vec_f32 vec_add( vec_f32 a, vec_f32 b ) {
    return svadd_f32_x( svptrue_b32(), a, b );
}
```

Migrating x86 AVX code: our approach

- **Create a unified SIMD API for use in our code**
  - Hardware agnostic versions of all used SIMD operations
- **Develop specific implementations for each specific hardware implementation**
  - Make use of each CPU architecture strengths
  - For SVE set vector length at compile time
- **Rewrite our code using these functions**
  - Select SIMD target at compile time
- **For more details see (work in progress)**
  - https://github.com/ricardo-fonseca/zpic-parallel/tree/main/openmp/simd

Migrating x86 AVX code: simple example

Helper functions (only add shown)

```
// AVX2
typedef __m256 vfloat;
static inline __m256 vec_add( __m256 a, __m256 b ) {
    return _mm256_add_ps(a,b);
}

// SVE
typedef svfloat32_t vec_f32  __attribute__((arm_sve_vector_bits(__ARM_FEATURE_SVE_BITS)));
typedef vec_f32 vfloat;
static inline vec_f32 vec_add( vec_f32 a, vec_f32 b ) {
    return svadd_f32_x( svptrue_b32(), a, b );
}
```

Hardware agnostic position advance

```
vfloat move( vfloat x0, vfloat vel, float dt ) {
   return vec_add( x0, vec_mul( vel, dt ) );
}
```

zpic@edu

- **The ZPIC code suite**
  - Open-source PIC code suite for plasma physics education
  - Fully relativistic 1D/2D EM-PIC algorithm and Electrostatic 1D/2D PIC algorithm
  - Interactive use through Jupyter notebooks

- **Testbed for deploying PIC algorithm on new platforms**
  - ZPIC-parallel supports OpenMP, CUDA, ROCm, and SYCL
  - CPU version supports AVX2, AVX512, NEON and SVE

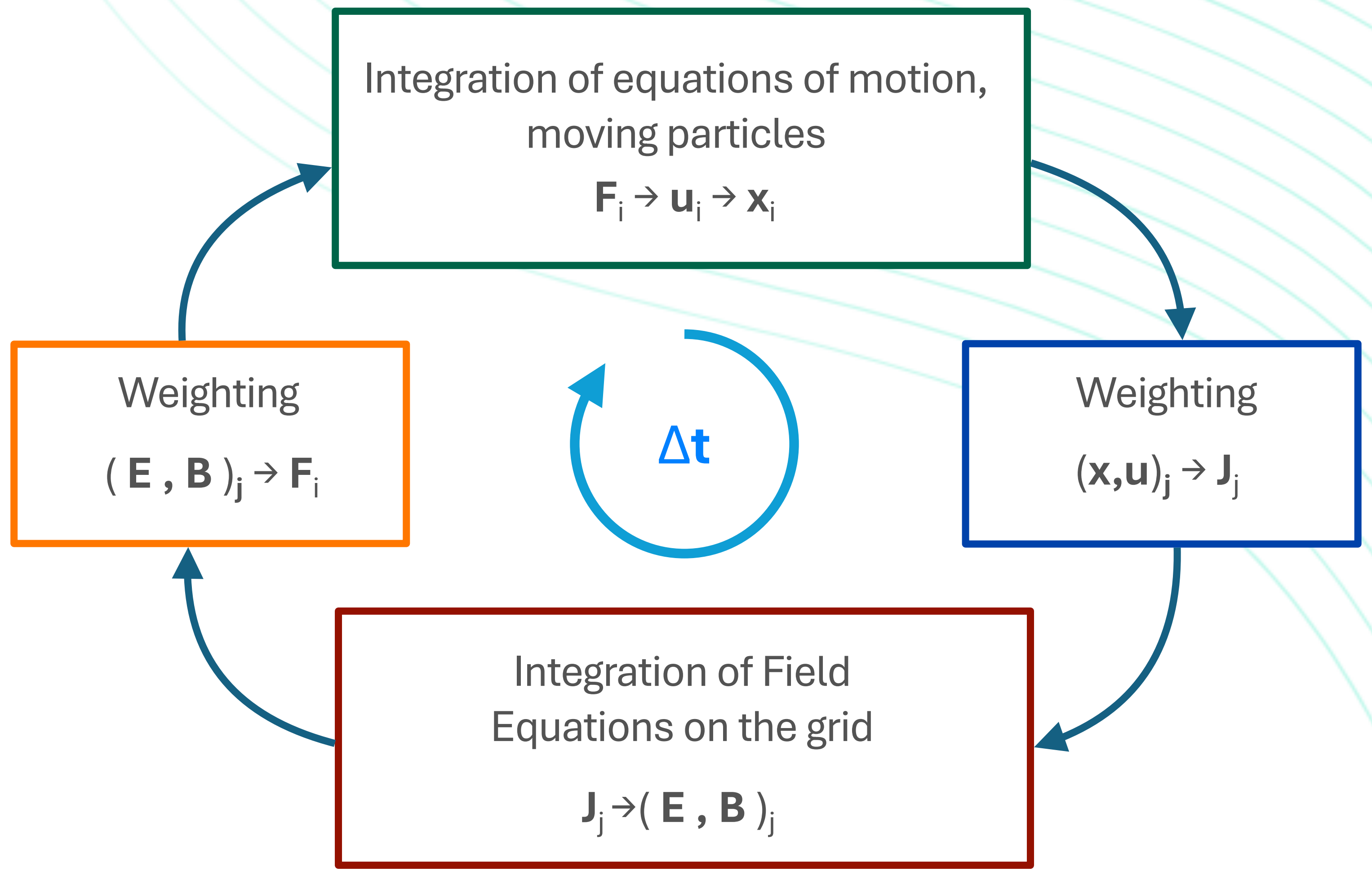
TÉCNICO LISBOA

Porting ZPIC to Arm

- **Standard OpenMP version compiles out of the box**
  - No changes required
  - Relied on compiler auto-vectorization
- **ZPIC already had explicit SIMD (x86 AVX) support**
  - Particle routines account for >90% of loop time
  - These routines were rewritten with explicit vectorization
  - For field routines we rely on the compiler for SIMD support
- **Porting to Arm required developing NEON/SVE interfaces**
  - Only the SIMD API had to be written, the main code remained the same
  - For SVE we had to set a fixed vector length

PIC Algorithm

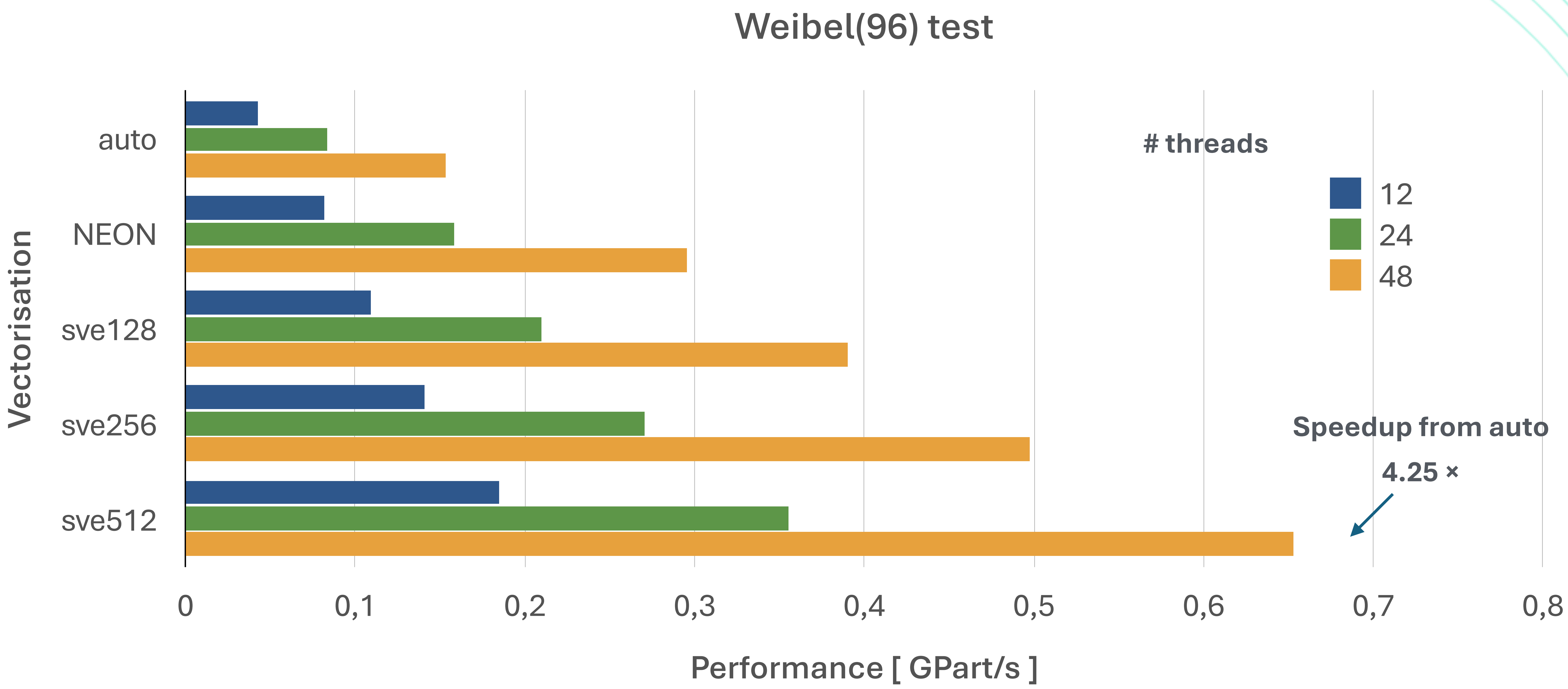
$$\frac{d\mathbf{u}}{dt} = \frac{q}{m}\left(\mathbf{E} + \frac{1}{\gamma c}\mathbf{u} \times \mathbf{B}\right)$$

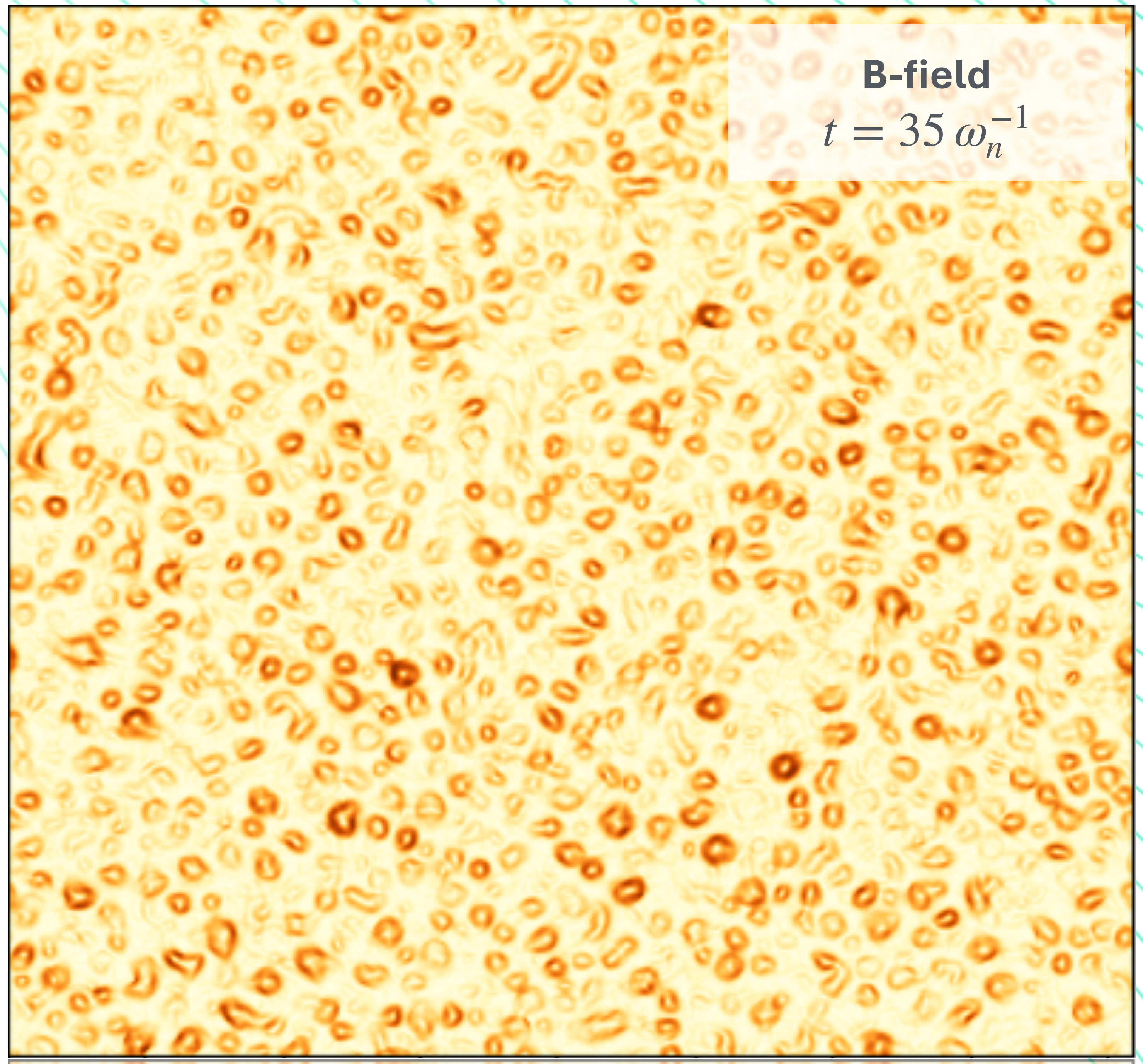


$$\frac{\partial \mathbf{E}}{dt} = c\vec{\nabla} \times \mathbf{B} - 4\pi\mathbf{j}$$

$$\frac{\partial \mathbf{B}}{dt} = -c\vec{\nabla} \times \mathbf{E}$$

Performance





- **Compiler**
  - Arm C/C++/Fortran Compiler version 24.04 (build number 9) (based on LLVM 18.1.1)
- **Neon / auto**

```
armclang++ -O3 -mcpu=a64fx
```

- **SVE512**

```
armclang++ -O3 -mcpu=a64fx -msve-vector-bits=512
```

- **Tests run on a single A64FX cpu @ Deucalion**

Simulation setup

- Collision of an electron and a positron plasma cloud
- 2D simulation in the perpendicular plane:
  - 2048^2 cells
  - 537 M particles
  - 500 time-steps

Overview

- **Using SVE in your code can lead to dramatic performance increases**
  - Make sure you are using the appropriate compiler flags / libraries
- **Directly programming for SVE using ACLE will give you access to the most advanced Arm CPU features**
  - This approach may force you to rethink your algorithms in terms of vectorization
  - It may require extensive coding, but it will generally lead to high performance gains
- **Porting from existing SIMD code for other architectures is relatively straightforward**
  - Most types of SIMD instructions have a direct equivalent
  - If VLA cannot be used in your code, you can set fixed vector lengths and go around this limitation

References

- Arm C/C++ compiler reference - Optimization
  - https://developer.arm.com/documentation/101458/2404/Optimize
- Arm Performance Libraries
  - https://developer.arm.com/Tools%20and%20Software/Arm%20Performance%20Libraries

- Introduction to SVE
  - https://developer.arm.com/documentation/102476/0100/
- SVE optimization guide
  - https://developer.arm.com/documentation/102699/0100/
- SVE and SVE2 programming examples
  - https://developer.arm.com/documentation/dai0548/c/

- Arm Intrinsics
  - https://developer.arm.com/architectures/instruction-sets/intrinsics/

EPICURE
Unlocking European-level HPC Support

Thank you!

pmo-epicure@postit.csc.fi

Follow us

Co-funded by the European Union

EuroHPC Joint Undertaking

This project has received funding from the European High Performance Computing Joint Undertaking under grant agreementNo.101139786. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union orEuroHPC Joint Undertaking. Neither the European Union nor the granting authority can be held responsible for them.