Flagship scientific applications on an European RISC-V long-vector accelerator: Lessons learned

European Processor Initiative

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Pablo Vizcaino, Marc Blancafort

29/04/2026, Online webinar

Outline

I) Introduction and context *(20 mins)*
   1. EPAC, RISC-V, RVV *(hardware)*
   2. CoEs and co-design *(software)*
   3. Software Development Vehicles *(methodology)*

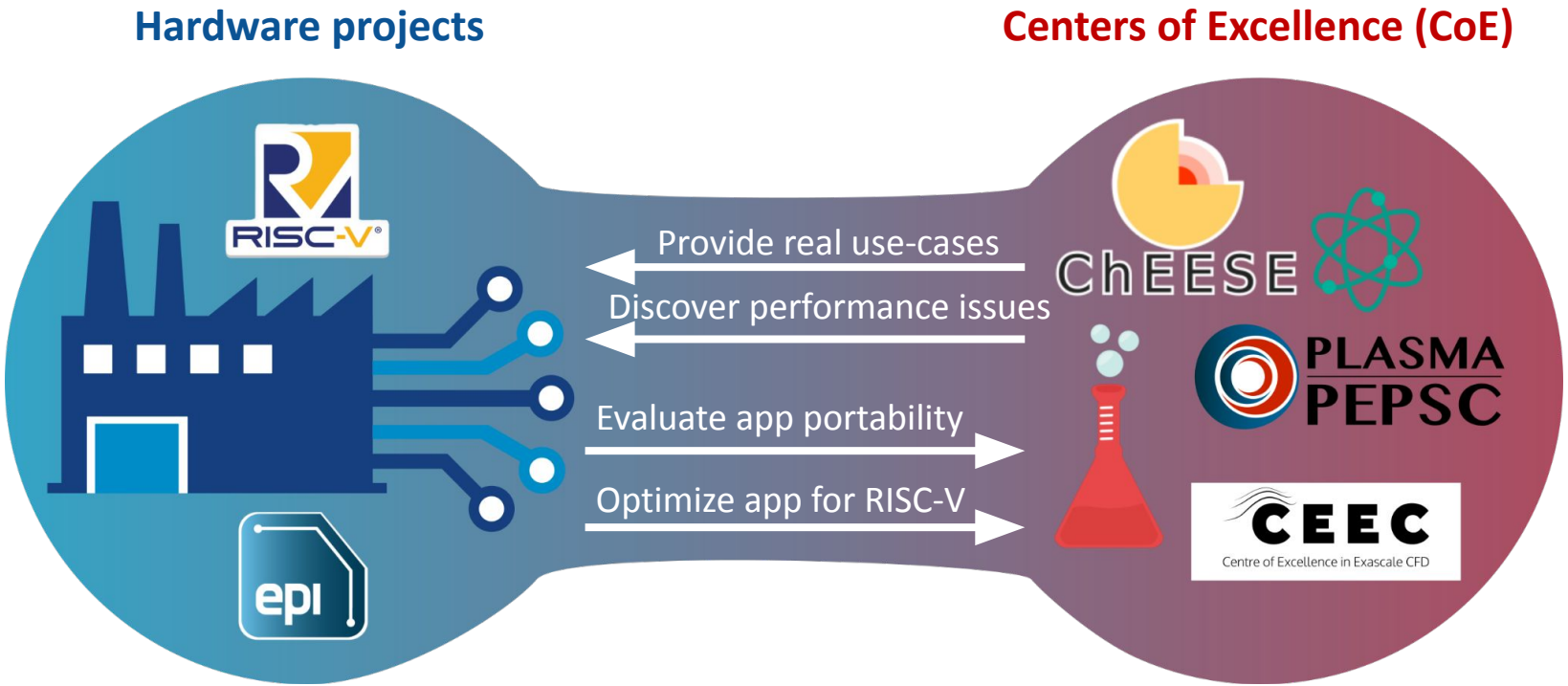
II) Examples from CEEC applications *(30 mins)*
   1. Sod2d
   2. Walberla
   3. Alya-Solidz

III) Conclusions and lessons learned *(10 mins)*

IV) Q&A

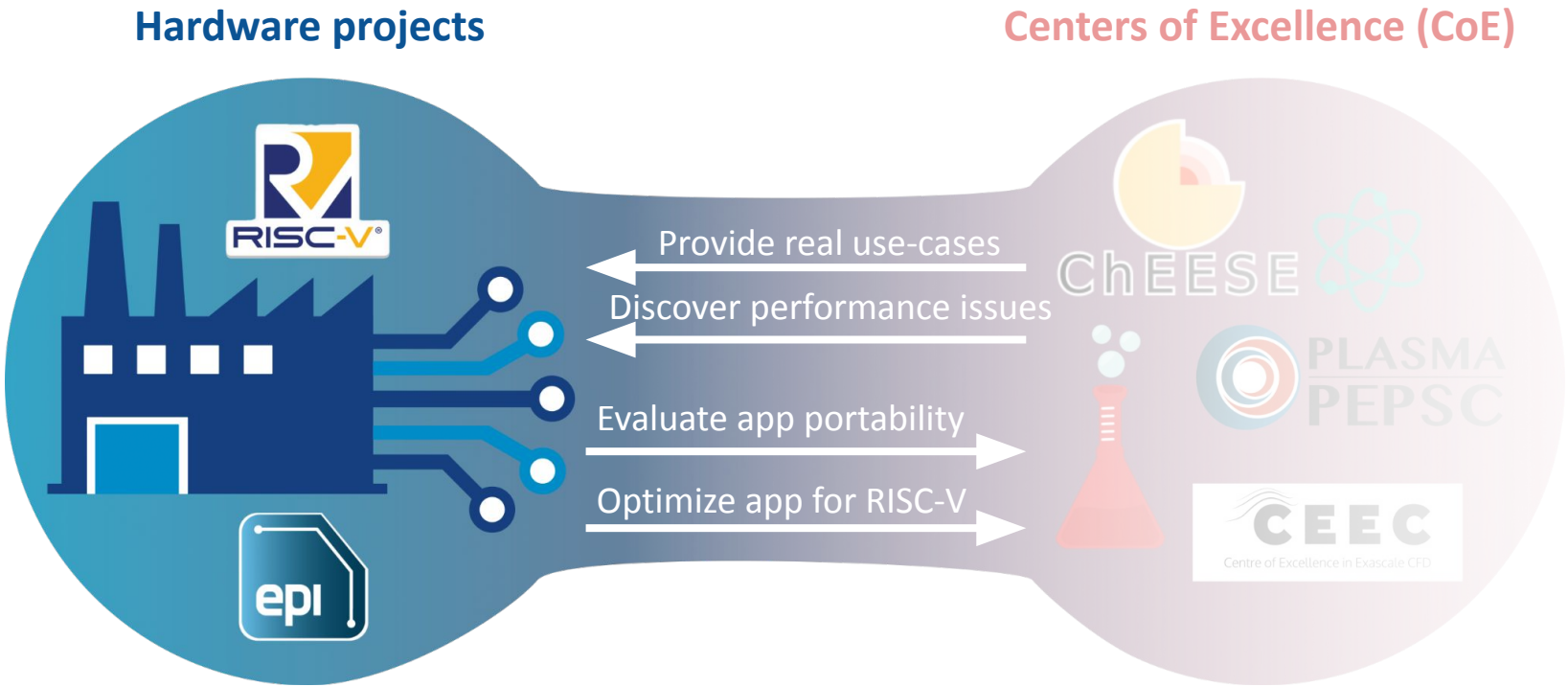
Context

Union of two research efforts currently ongoing at the Barcelona Supercomputing Center (BSC)



Context

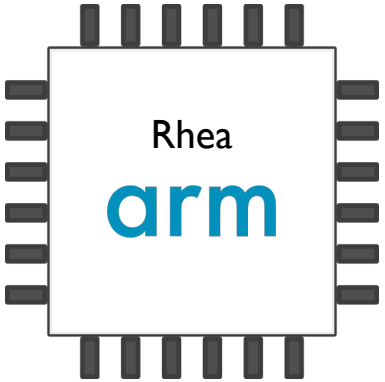
Union of two research efforts currently ongoing at the Barcelona Supercomputing Center (BSC)



Efforts bringing RISC-V to HPC

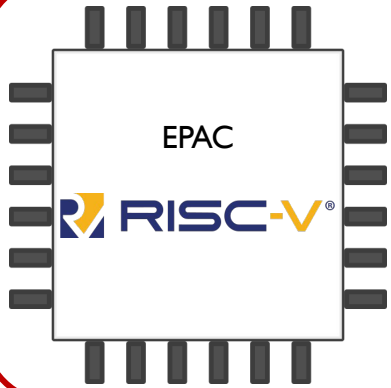
The European Processor Initiative (EPI)

- Chasing chip sovereignty in Europe
- 30 partners
- Two main research lines



RHEA

- General purpose CPU
- ARM based



European Processor Accelerator (EPAC)

- RISC-V Based
- Contains acceleration tiles
- **Long Vector** unit

What is RISC-V?

- RISC-V is an Instruction Set Architecture (ISA)
- RISC-V is open source
- ❌ RISC-V is not free hardware (implementation)
- ✅ RISC-V is a free piece of paper! (standard)

Area	Standard	Implementation
Network	Ethernet	10BaseT, 1000BaseT, …
Wireless	Wi-Fi	W-Fi 5 (802.11ac), …
Graphics	OpenGL	NVIDIA GeForce, Mesa 3D, …
Processor Architecture	**RISC-V**	SiFive Unmatched, Sophon SG2042, Bananapif3, EPAC, …

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

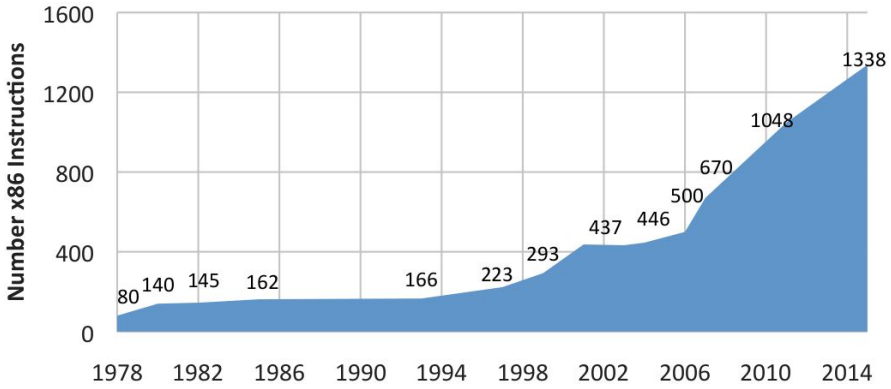
RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Modular vs incremental ISA

- **Intel x86** is incremental, each new release:
  - Maintains backward compatibility
  - Carry on new instructions

- Cores get bigger and bigger…



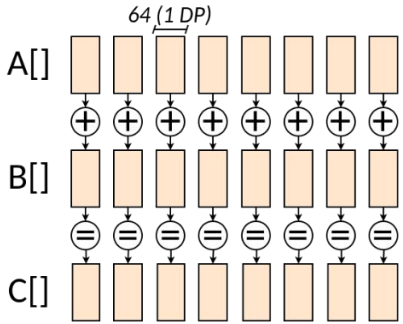
RISC-V is modular!

Name	Description	Version	Status	Instruction Count
RV32I	Base Integer Instruction Set - 32-bit	2.1	Frozen	49
RV32E	Base Integer Instruction Set (embedded) - 32-bit, 16 registers	1.9	Open	Same as RV32I
RV64I	Base Integer Instruction Set - 64-bit	2.0	Frozen	14
RV128I	Base Integer Instruction Set - 128-bit	1.7	Open	14
	Extension			
M	Standard Extension for Integer Multiplication and Division	2.0	Frozen	8
A	Standard Extension for Atomic Instructions	2.0	Frozen	11
F	Standard Extension for Single-Precision Floating-Point	2.0	Frozen	25
D	Standard Extension for Double-Precision Floating-Point	2.0	Frozen	25
G	Shorthand for the base and above extensions	n/a	n/a	n/a
Q	Standard Extension for Quad-Precision Floating-Point	2.0	Frozen	27
L	Standard Extension for Decimal Floating-Point	0.0	Open	Undefined Yet
C	Standard Extension for Compressed Instructions	2.0	Frozen	36
B	Standard Extension for Bit Manipulation	0.90	Open	42
J	Standard Extension for Dynamically Translated Languages	0.0	Open	Undefined Yet
T	Standard Extension for Transactional Memory	0.0	Open	Undefined Yet
P	Standard Extension for Packed-SIMD Instructions	0.1	Open	Undefined Yet
V	Standard Extension for Vector Operations	0.7	Open	186
N	Standard Extension for User-Level Interrupts	1.1	Open	3
H	Standard Extension for Hypervisor	1.0	Frozen	2
S	Standard Extension for Supervisor-level Instructions	1.12	Open	7

Vector computing

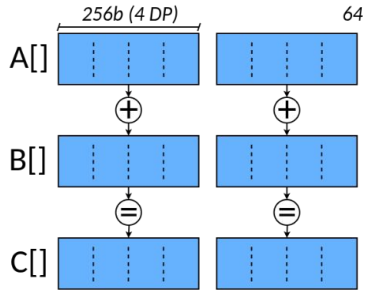
Scalar:

One instruction → one op/result



Vector/SIMD:

One Instruction → multiple ops/results



Many SIMD extensions in the market

intel	***SSE*** 128b	***AVX2*** 256b	***AVX512*** 512b
arm	***NEON*** 128b	***SVE*** [128b→2048b]
NEC	***NEC-VE*** 16384b	
RISC-V	***RVV*** [128b → *]	

RISC-V Vector Extension (RVV): What's interesting?

European Processor Initiative

ISA specific

Advanced vector instructions
gather/scatter instructions, masking, reductions, segmented loads, …

→ **Versatility!**
Allows vectorizing complex and sparse workloads

Implementation specific

VLEN
Size of vector registers, in bits [128b:*]

→ **Scalability!**
Same ISA, vastly different use-cases (embedded vs HPC)

Runtime variable

VL *(Vector Length)*
#Elements operated in an instruction
From *0* to *VLEN/sizeof(data)*
Dynamically changes

→ **Maleability!**
No restriction for VL *(2^x, even, …)*
Vector-length agnostic code
Avoid scalar tails in loops

⚠ VL can be smaller than VLEN depending on code pattern:
→ Not as efficient

SEW, LMUL, … ← Look them up!

Going wide

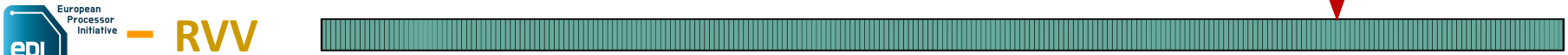
EPAC long vectors: Leverage RVV **scalability,** use a **very long VLEN** *(max. length of vectors)*

intel – **AVX512** ← 512 bits per vector (8 DP elems)

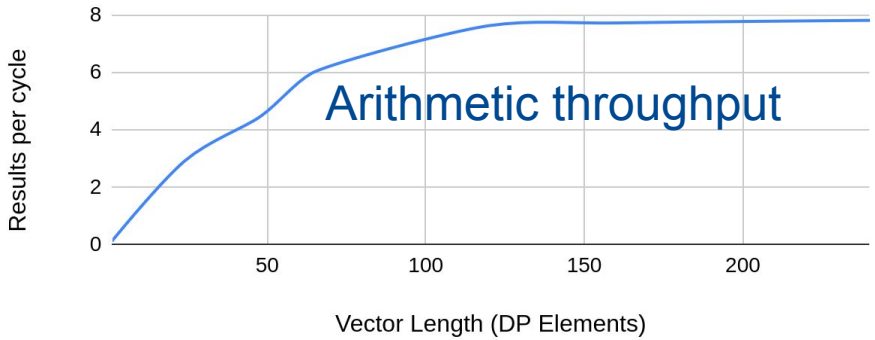
arm – **SVE** ← Up to 2048 bits per vector (16 DP elems)

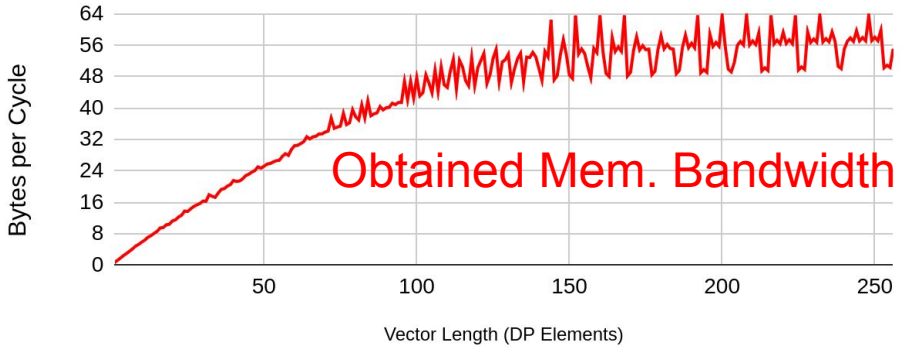
European Processor Initiative – **RVV**

16384 bits per vector (256 DP elems)



It's important to fill up the vectors!



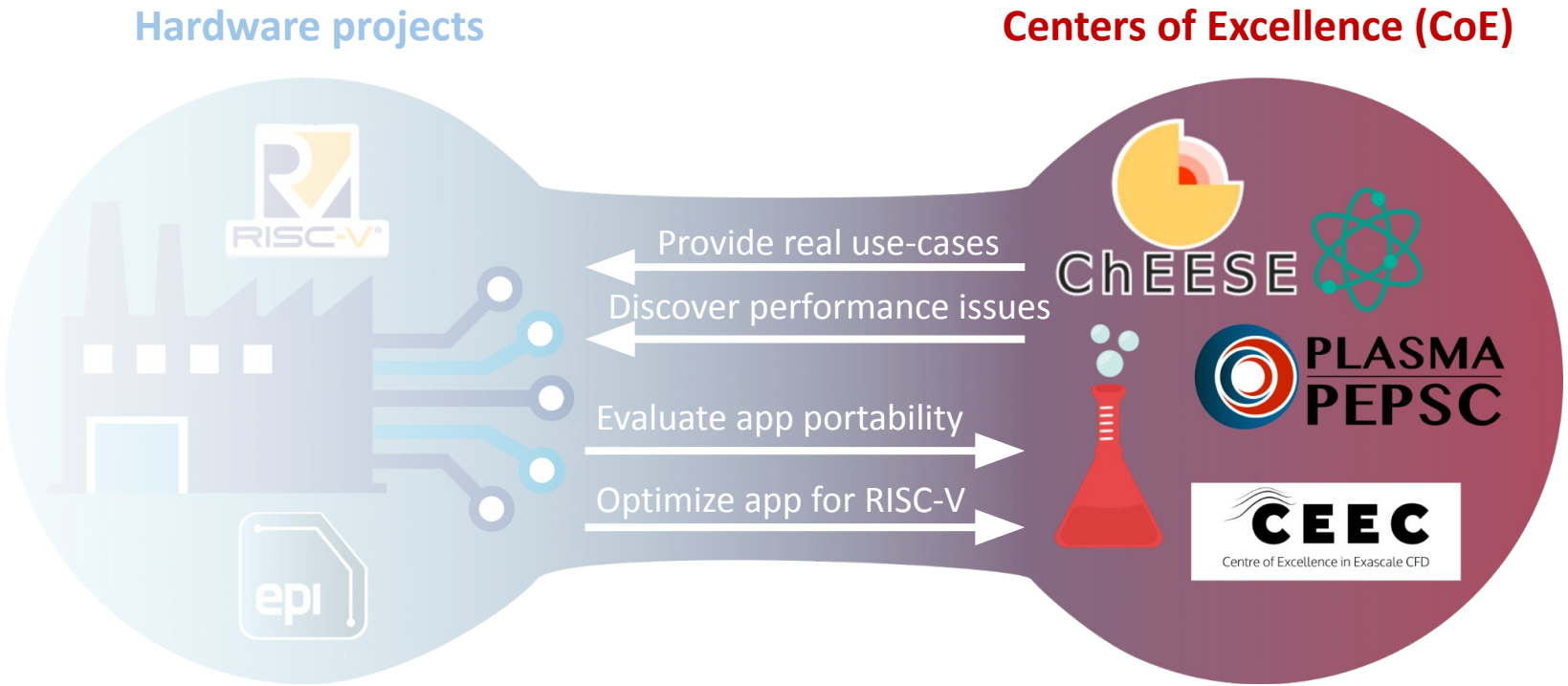


Take-Home messages (RVV):

- European Processor Initiative (**EPI**) project developed the **EPAC** RISC-V chip
- **EPAC** implements the RISC-V vector extension (**RVV**)
- **RVV** supports:
  - Implementation-specific VLEN (max VL) ⟶ **EPAC** chooses 256 DP elements
  - **Variable VL** ⟶ vl-agnostic code, but instructions might not fill the vectors

Context

Union of two research efforts currently ongoing at the Barcelona Supercomputing Center (BSC)



Context: CoE

- Centers of Excellence (CoE) are hubs focusing efforts on specific science fields
  - Collaboration between multiple institutions
  - Work packages on optimization, parallelization, performance characterization, porting, …

- Our research group participates in the Co-design work packages of:

PLASMA PEPSC → **Plasma** physics, 5 applications

CEEC Centre of Excellence in Exascale CFD → **Computational Fluid dynamics**, 6 applications

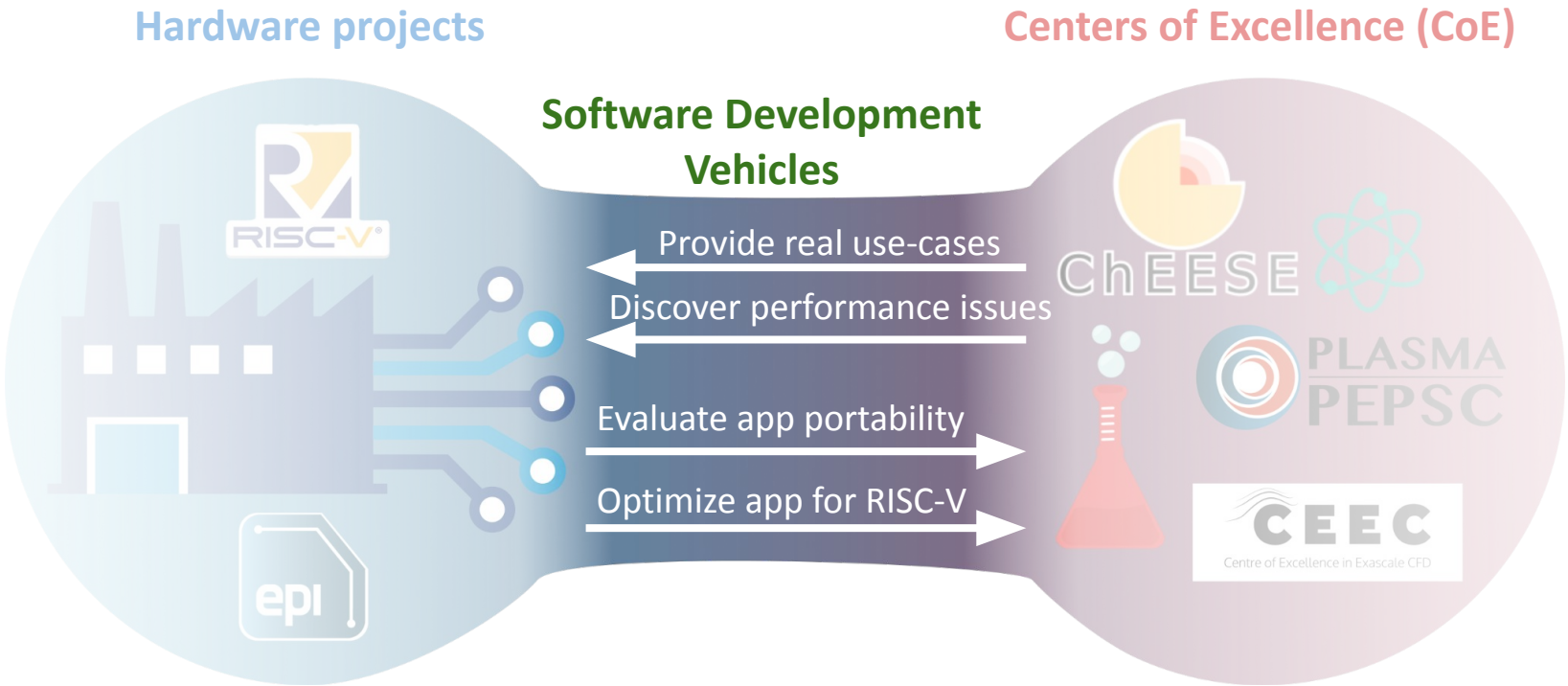
ChEESE Center of Excellence for Exascale in Solid Earth → Geohazards and **Solid-Earth** physics, 10 codes

Goal of Co-design:

1. Is the EPI prototype ready for HPC codes?
2. Are the codes ready to take advantage of the prototype?

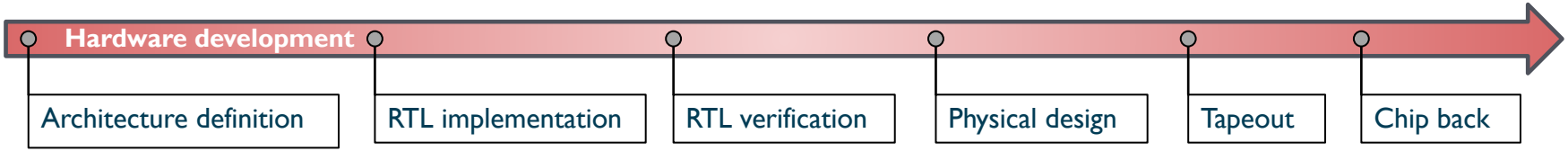
Context

Union of two research efforts currently ongoing at the Barcelona Supercomputing Center (BSC)

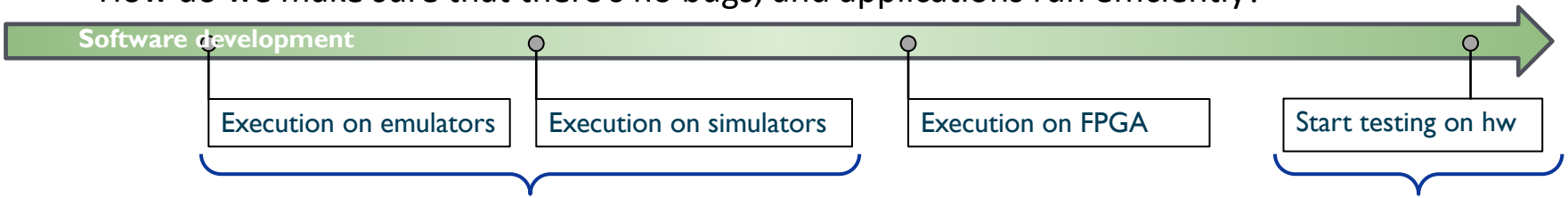


Writing software for hardware that doesn't exist *(yet)*

- Hardware development requires a lot of time: years!



- How do we make sure that there's no bugs, and applications run efficiently?



Usually constrained to a **few hundred** instructions
- **HPC apps execute billions of instructions!**

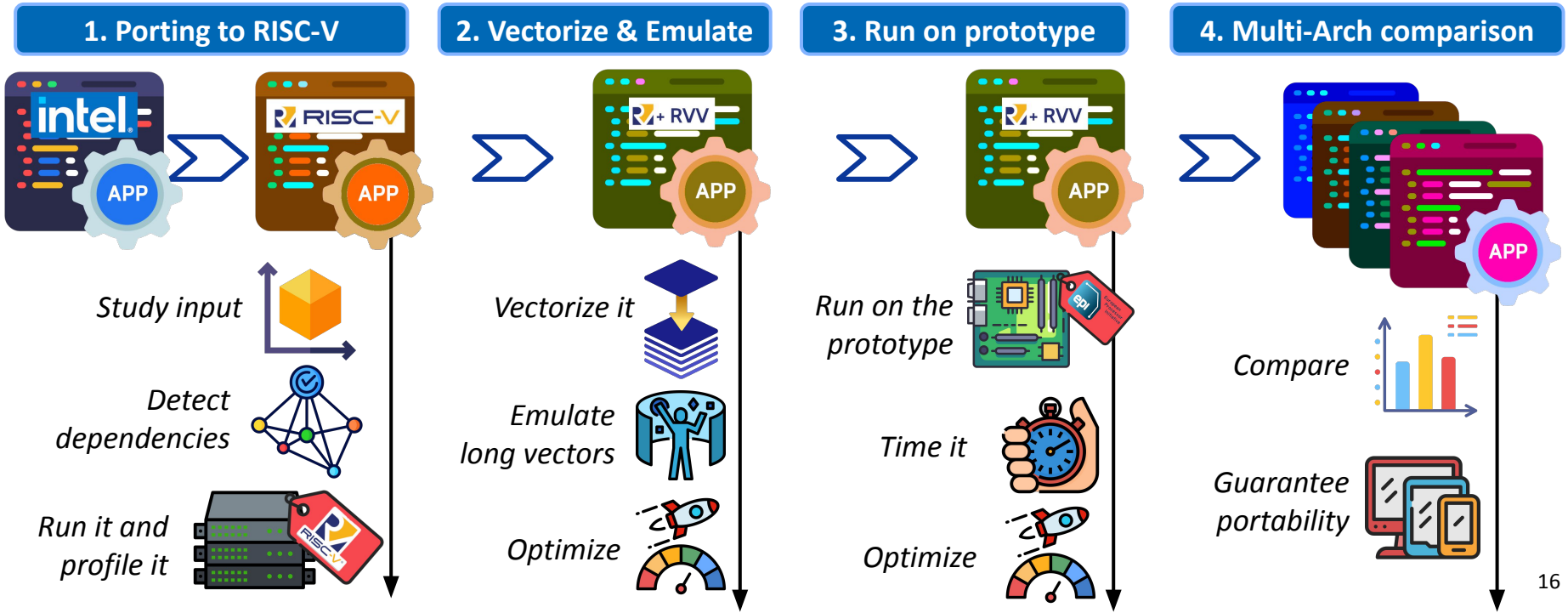
Fast testing… but too late to make HW changes!

Simulators / Emulators usually require advanced knowledge of hardware architecture

We can't expect application scientists to be hardware experts

The Software Development Vehicles (SDV)

- Within the EPI project, we developed an **infrastructure** and an **analysis methodology**
  - Designed to ease software evaluation for early prototypes

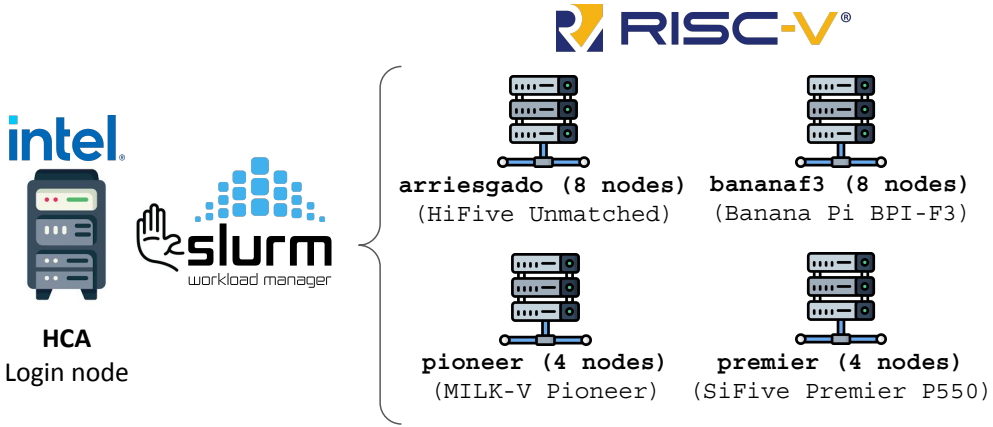


Step 1: Port to RISC-V and profiling

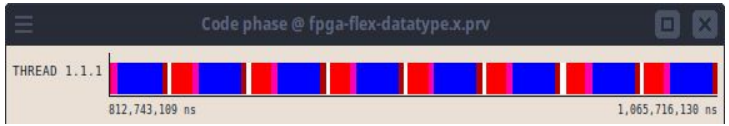
- Detect incompatible dependencies:

```
pioneer$ clang HPCG.c -o HPCG.x
/usr/bin/ld: /tmp/main-4cc29c.o: in function `conj_grad':
main.c:(.text+0x3a): undefined reference to `mkl_sparse_d_mv'
/usr/bin/ld: main.c:(.text+0x5e): undefined reference to `spmv_mkl_kernel'
/usr/bin/ld: main.c:(.text+0x98): undefined reference to `mkl_sparse_d_create_csr'
```

- Understand the application structure when running on commercial boards:

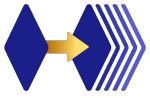


Extrae/Paraver Profiling



Pressures_cbrt	Temperatures	Volumes	Delta	Pressures_vec
895,459.12	102,526.43	410,032.38	110,140.25	85,646.75

Step 2.1: Vectorization with the EPI compiler



- The **EPI Compiler** can compile (and vectorize!) code for **RVV0.7.1** and **RVV1.0** *(C, C++, Fortran)*

Automatic (no hints)

```
for (int i=0; i<N; ++i){
  B[i] = A[i];
}
```

Guided (using pragmas)

```
#pragma clang loop vectorize(enable)
for (int i=0; i<N; ++i){
  B[i] = A[i];
}
```

```
friendly.c:56:2: remark: vectorized loop (vectorization width: v
friendly.c:65:22: remark: loop not vectorized: call instruction
                  array[i*M + j] = (rand()%1024) / 1024.0;
```

Intrinsics (explicit vectorization)

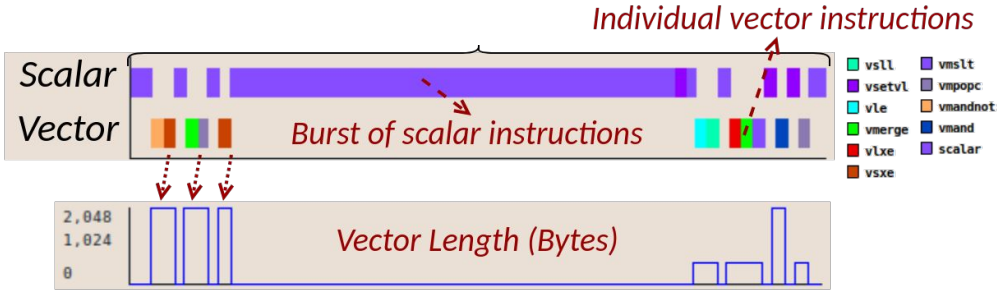
```
for (int i=0; i<N; ){
  long vl = __builtin_epi_vsetvl(N-i,e64,m1);
  __epi_1xf64 va = __builtin_epi_vload_1xf64(&A[i],vl);
  __builtin_epi_vstore_1xf64(&B[i], va, vl);
  i += vl;
}
```

Step 2.2: RAVE Emulation

Dynamic vector code analysis (on x86 host):

```
├─ Counters:
│   ├─Moved bytes: 144167
│   │   ├─ Scalar: 79 (0.05 %)
│   │   └─ Vector: 144088 (99.95 %)
│   └─Instructions: 339
│       ├─ Scalar instr: 196 (57.82 %)
│       ├─ Vsetvl instr: 22 (6.49 %)
│       └─ Vector instr: 121 (35.69 %)
│           ├─ SEW 8: 0
│           ├─ SEW 16: 0
│           ├─ SEW 32: 0
│           └─ SEW 64: 121 (100.00%) [avg VL: 233.91 elems]
│               ├─ Arith: 33 (27.27%)  [avg VL: 233.91 elems]
│               │   ├─ FP: 33 (100.00 %)
│               │   └─ INT: 0
│               ├─ Memory: 77 (63.64%) [avg VL: 233.91 elems]
│               │   ├─ unit: 77 (100.00 %)
│               │   ├─ strided: 0
│               │   ├─ indexed: 0
│               │   └─ whole-register: 0
│               ├─ Mask: 11 (9.09%)  [avg VL: 233.91 elems]
│               └─ Other: 0
```

Emulation instruction traces:



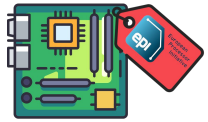
Prop. of Scalar and Vector instructions

	Init	BU	TD	Bit->Q (c)	Q->Bit (e)
qemu_scalar	0.62	0.63	1.00	0.81	0.74
qemu_vector	0.38	0.37	0.00	0.19	0.26

Assembly of instructions (In phase Init)

	vsub	vsll	vsetvl	vle	vmerge	vmv	vsxe	vmseq	scalar
qemu_scalar	-	-	640	-	-	-	-	-	641
qemu_vector	128	128	-	256	128	128	128	128	-

Step 3: Run on real EPAC Hardware



European Processor Initiative

We can access the **EPAC** implemented in an **FPGA** like a “normal” SLURM node:

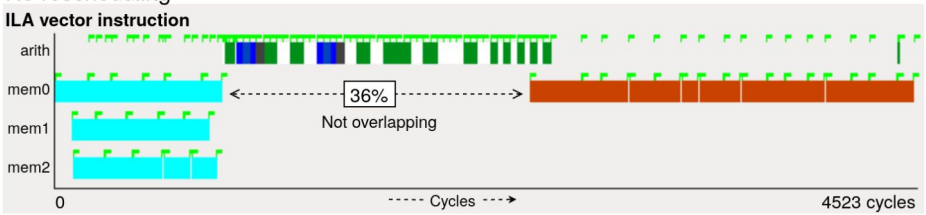
```
hca $ salloc -p fpga-epac -t 01:00:00
...

hca $ ssh fpga-epac-2
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux
5.7.0-yarvt-sdv3-minimal-network+ riscv64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
```

FPGA - SDV

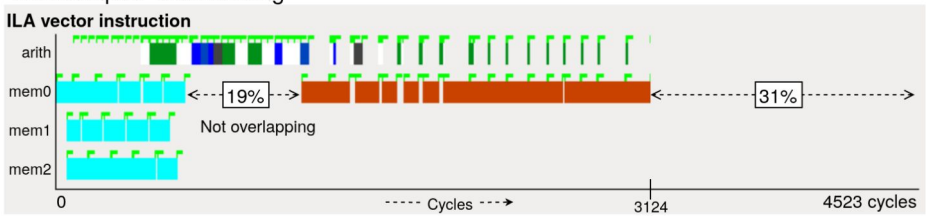
```
fpga-epac-2 $ cat /proc/cpuinfo
isa         : rv64imafdcv
mmu         : sv39
uarch       : epi,avispado
fpga-epac-2 $ ./run_your_cool_app.bin
```

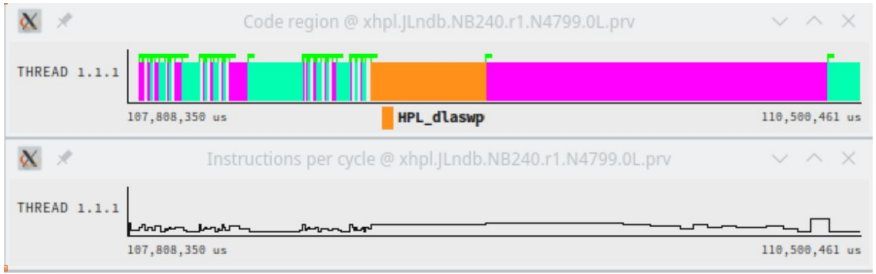
… And get execution/timing traces

No rescheduling



With compiler rescheduling





Step 4: Cross-Architecture comparisons

- Leverage that we run on a standard environment (Ubuntu, SLURM, …)
- Compare performance and speedups against SoTA Machines:

Marenostrum5 (GPP partition)

Intel Sapphirerapids
3 GHz clock
x86 + AVX512

Marenostrum5 (NGP partition)

NVIDIA Grace Superchip
3.3 GHz clock
arm + SVE (128b)

NEC SX-Aurora TSUBASA

Vector Engine 20-B
1.6 GHz clock
NEC vector extension (16Kb)

Take-Home messages (SDV):

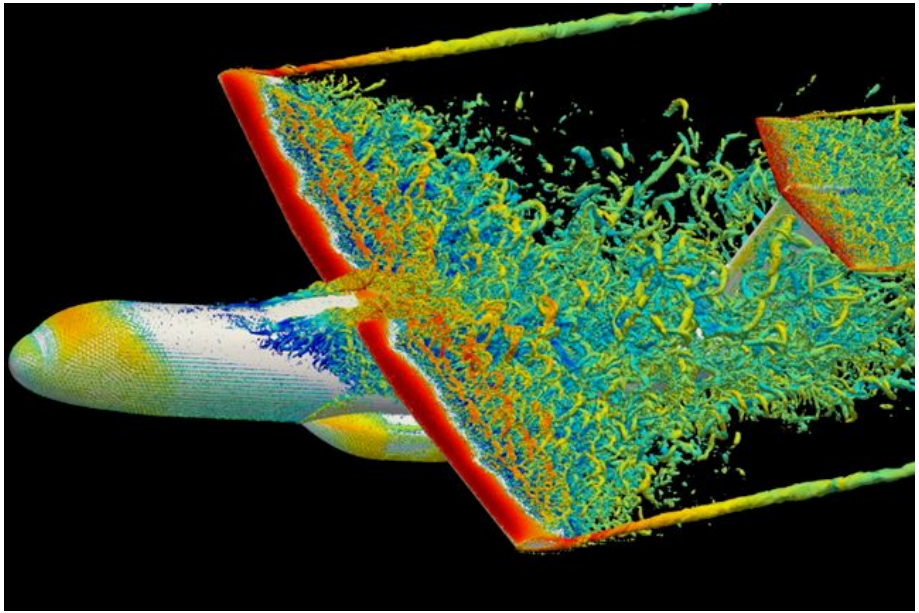
- **SDV** methodology helps optimizing applications before the hardware is available
  - Used to co-design CoE Apps

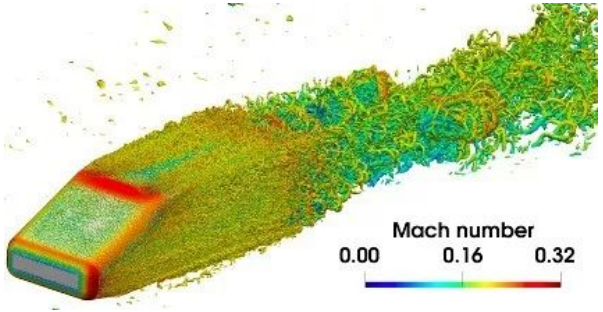
- **Four steps**:
  1. Porting and scalar profiling
  2. **Automatic** compiler **vectorization**, emulate, and study vectorization (**RAVE**)
  3. Measure performance on the **FPGA** / **Testchip**
  4. Compare against other HPC platforms

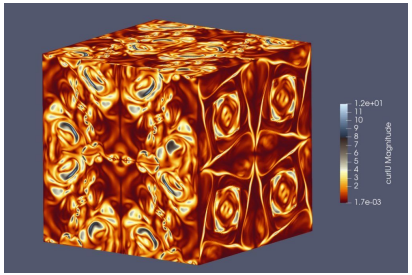
Let's see some examples!

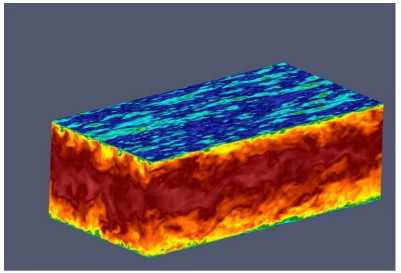
SOD2D

Spectral high-Order coDe 2 solve partial Differential equations









How is SOD2D engineered?

The algorithm is designed to be GPU-accelerated
Implemented using OpenACC for portability

The number of nodes per element can be selected with the **porder** parameter

SOD2D: A GPU-enabled Spectral Finite Elements Method for compressible scale-resolving simulations ☆

L. Gasparino [a,*], F. Spiga [b], O. Lehmkuhl [a]

[a] *Barcelona Supercomputing Center (BSC), Barcelona, Spain*
[b] *NVIDIA Corporation, Santa Clara, CA, USA*

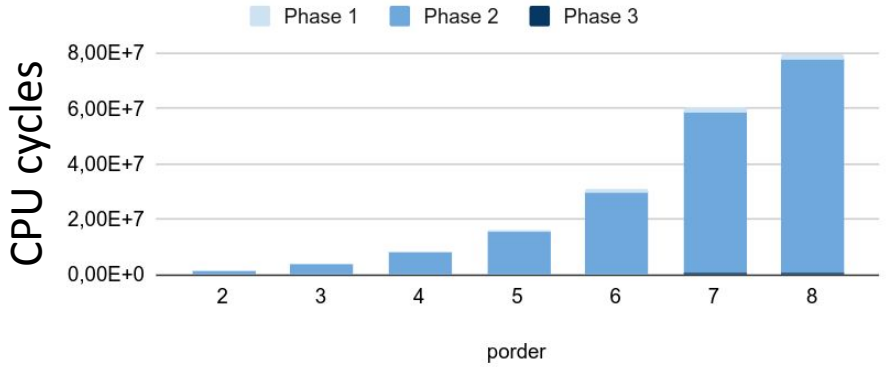
Preliminary vector analysis

CEEC Centre of Excellence in Exascale CFD

European Processor Initiative epi

Step 1: Run on scalar commercial boards →

- Three main phases (**Phase 2 dominates**)
- Input size → controlled by "**porder**" var



Step 2: Vectorize and Emulate

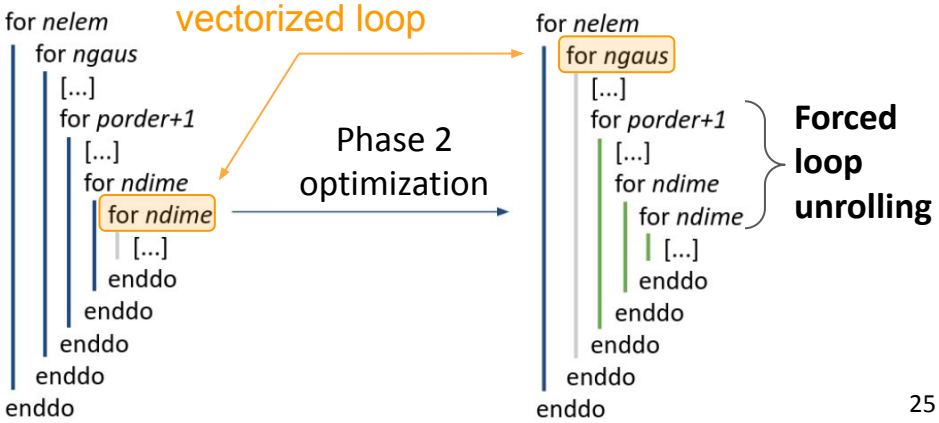
VECTOR MIX (vec/total instr)

Phase	porder						
	2	3	4	5	6	7	8
1	1,8%	1,8%	1,8%	1,1%	2,5%	1,8%	2,5%
2	11,9%	13,9%	16,0%	17,4%	18,4%	19,2%	20,0%
3	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%

VECTOR Occupancy (avg.VL / 256)

phase	porder						
	2	3	4	5	6	7	8
2	1.17	1.17	1.17	1.17	1.17	1.17	1.17

Very low VL (3/256 = 1.17%)



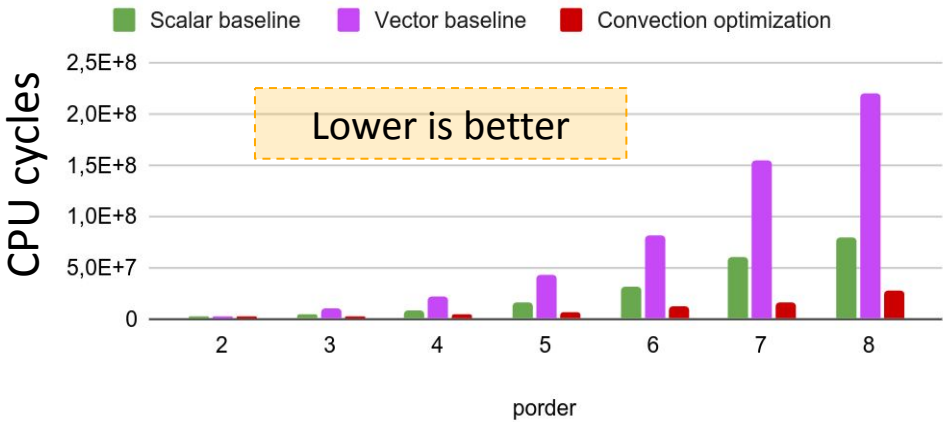
Results after optimization

Step 2: Vectorize and Emulate *(after optimization!)*

VECTOR Occupancy (%)						
porder						
2	3	4	5	6	7	8
10.55	25.00	48.83	84.38	66.99	100.00	94.92

Improved from 1% to 100% (depending on porder)

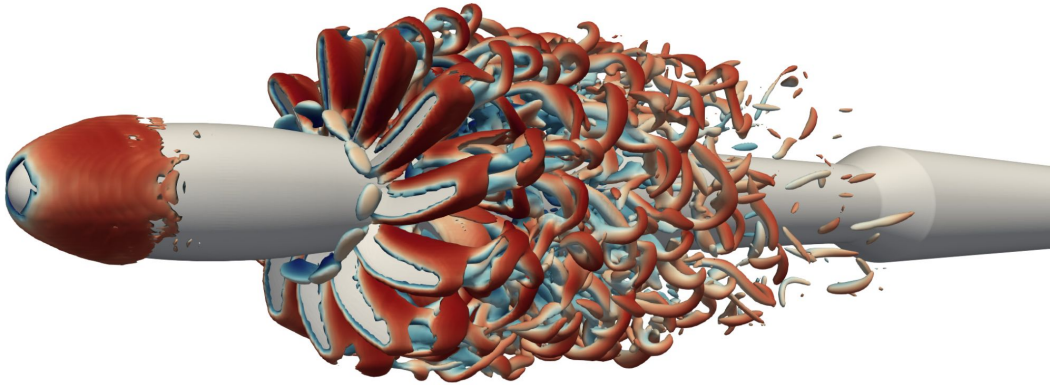
Step 3: Run on FPGA prototype

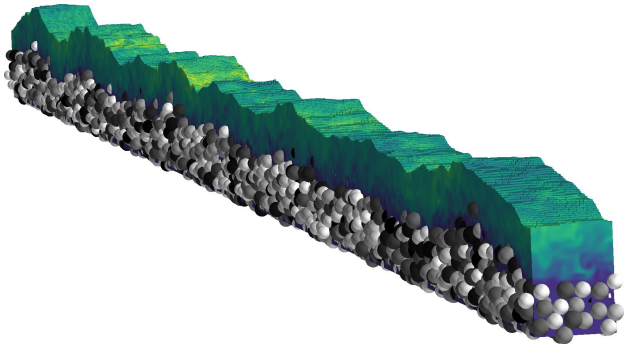


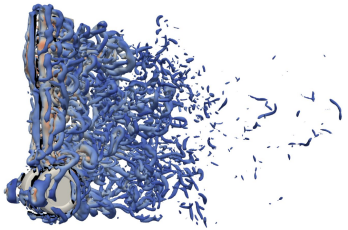
Speedup to scalar							
Implementation	porder						
	2	3	4	5	6	7	8
Scalar baseline	1,00	1,00	1,00	1,00	1,00	1,00	1,00
Vector baseline	0,49	0,41	0,38	0,37	0,38	0,39	0,36
Convection optimization	0,92	1,49	2,07	2,65	2,74	3,64	2,85

WaLBerla

Widely applicable Lattice Boltzmann from Erlangen







How is WaLBerla engineered?

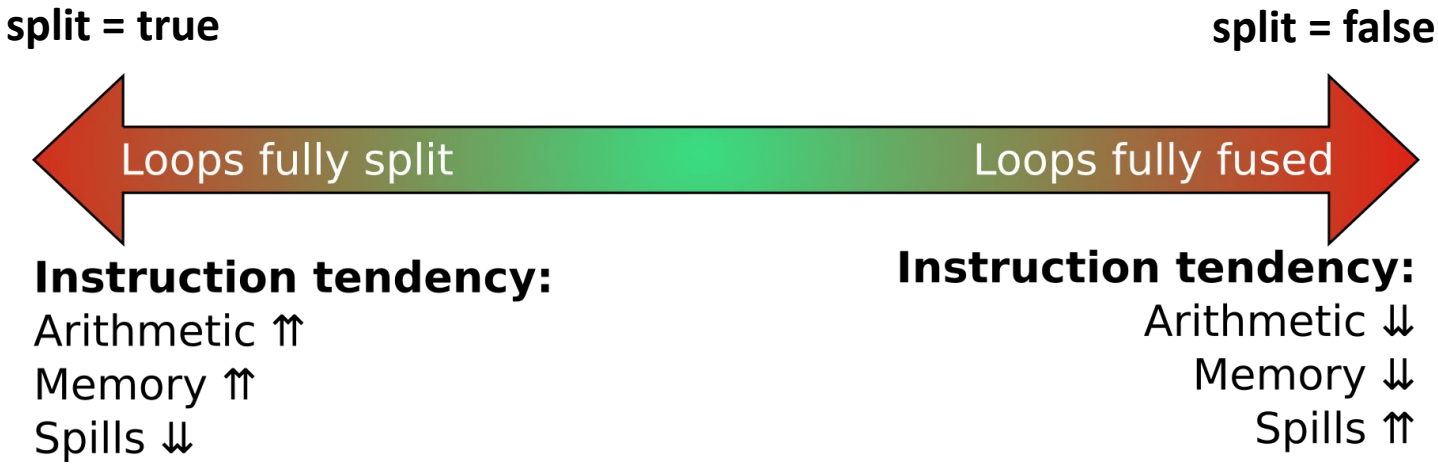
Straightforward LBM for CPU and GPU processors

- Ensures functional portability 😀
- Can lead to performance inefficiencies 🧐

  The **split** parameter 💡

```
// CPU
for (int64_t ctr_2 = 1; ctr_2 < _size_pdfs_2 - 1; ctr_2 += 1) {
    for (int64_t ctr_1 = 1; ctr_1 < _size_pdfs_1 - 1; ctr_1 += 1) {
        for (int64_t ctr_0 = 1; ctr_0 < _size_pdfs_0 - 1; ctr_0 += 1) {
            // D3Q27 stencil
        }
    }
}

// GPU
if (blockDim.x*blockIdx.x + threadIdx.x + 1 < _size_pdfs_0 - 1 &&
    blockDim.y*blockIdx.y + threadIdx.y + 1 < _size_pdfs_1 - 1 &&
    blockDim.z*blockIdx.z + threadIdx.z + 1 < _size_pdfs_2 - 1)
{
    // D3Q27 stencil
}
```

Is there a sweet point between the fully split and fully fused implementations?
How can we find it?

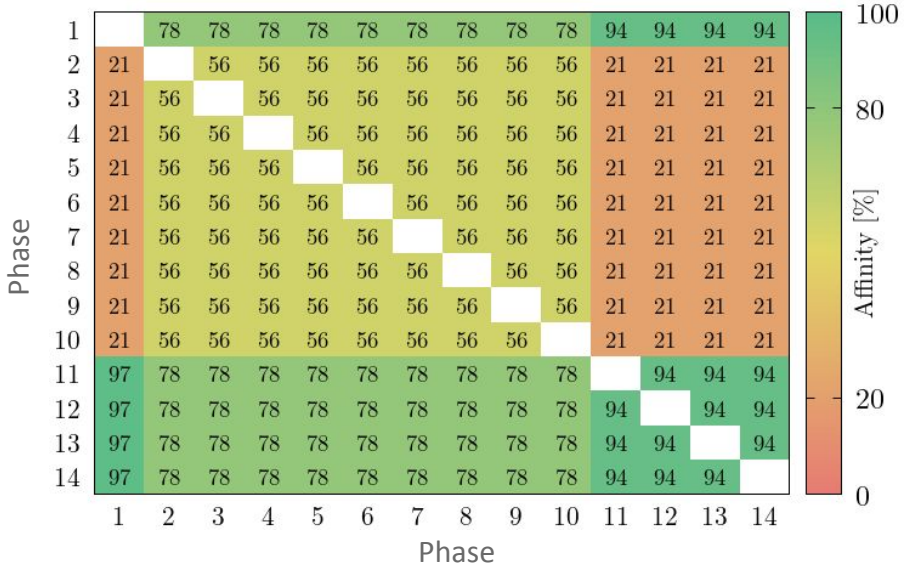


Memory overlap matrix

Tracks the memory access pattern for each phase (“loop”)

0% means memory accesses from phase *i* and phase *j* are disjoint

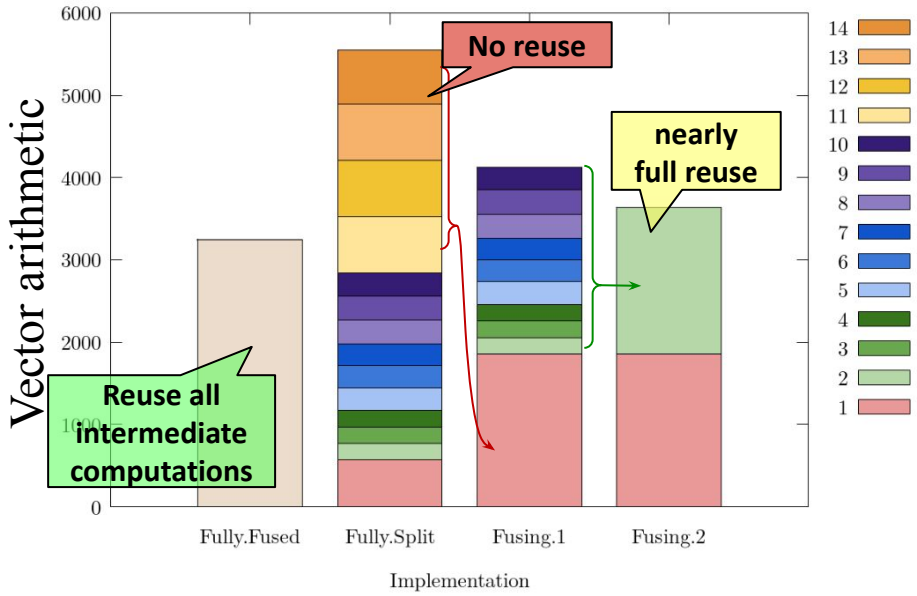
100% means memory accesses from phase *i* and phase *j* are the same

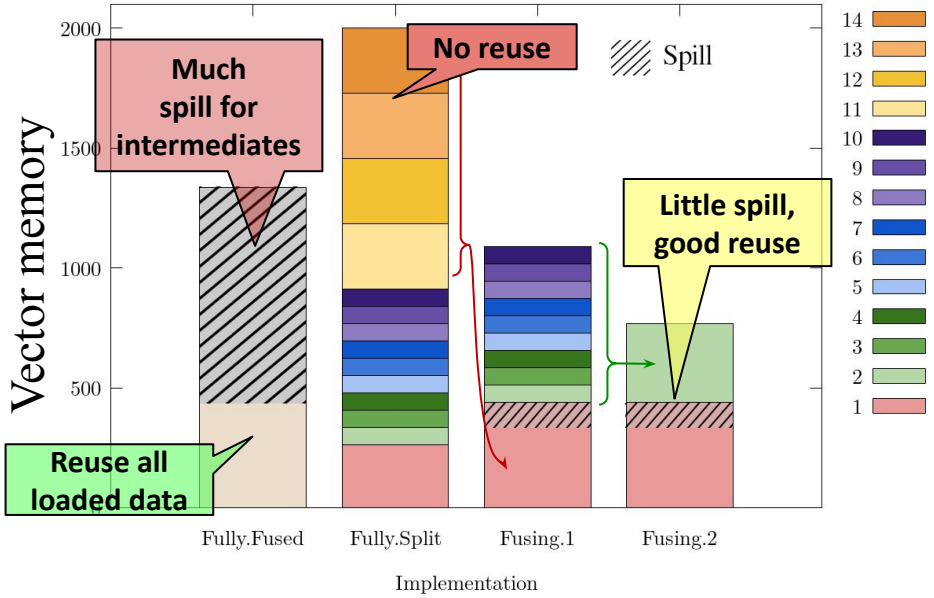


Guided-loop fusing

Step 2: Vectorize and Emulate

- Walberla has a big loop that can be broken into up to 14 phases (or fused as a single loop)



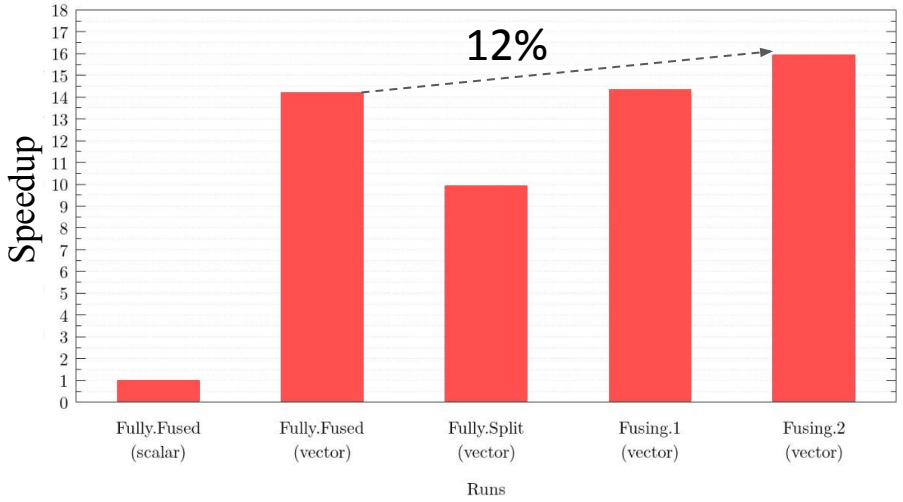


Are the extra Arithmetic/Memory worth the fewer Spills?

Results in long vector architectures

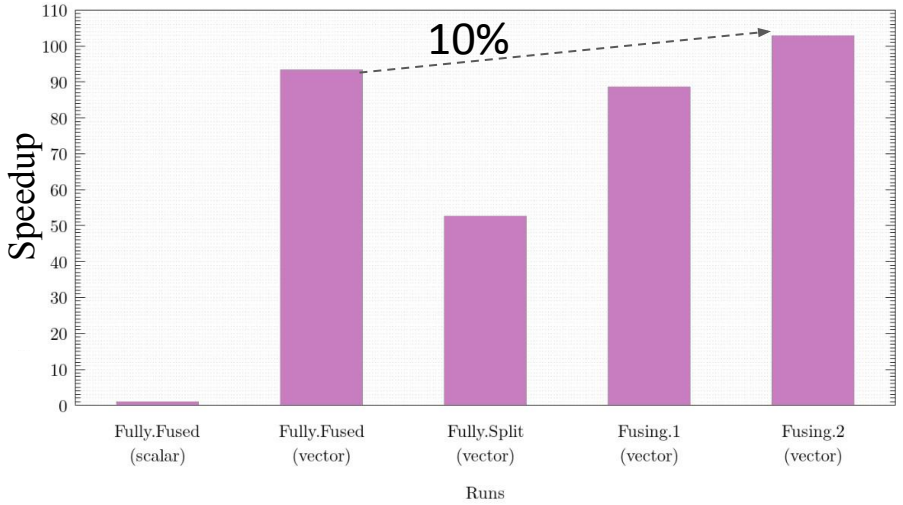
Step 3: Run on FPGA prototype

RISC-V prototype



Step 4: Cross-Architecture Comparisons

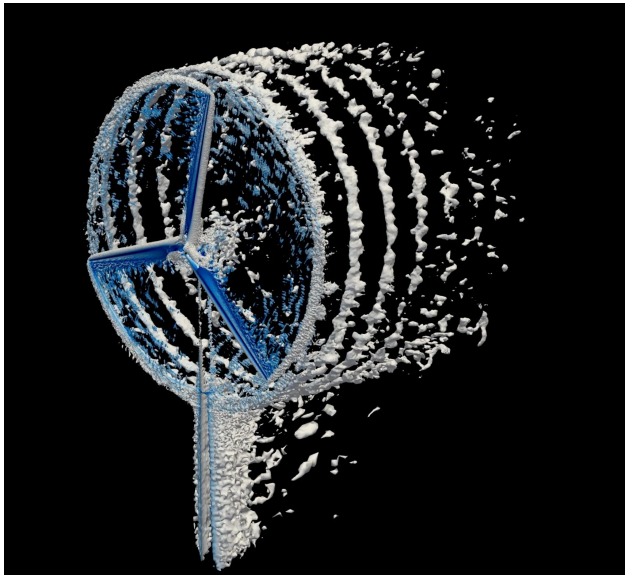
NEC SX-Aurora

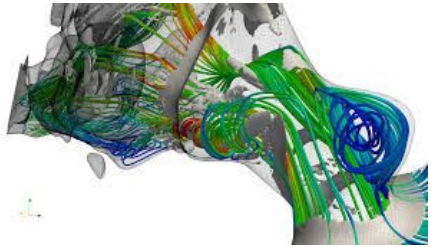


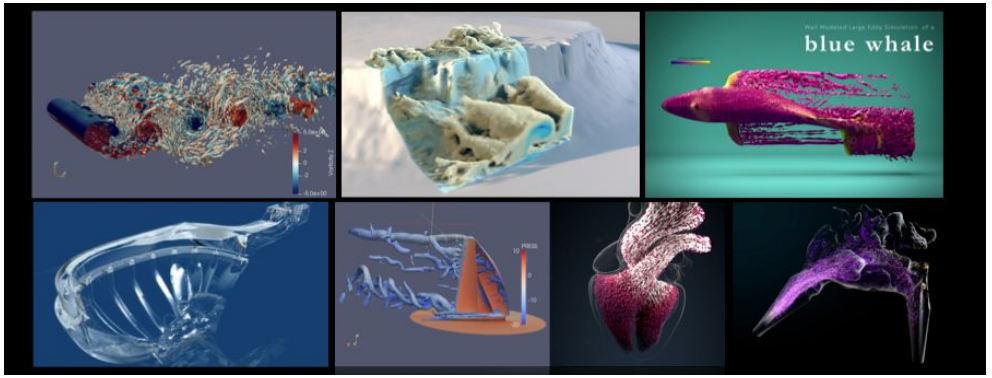
- Similar improvement of smart fusing over two long vector architectures:
  - **A few extra arithmetic/memory operations are worth fewer spills!**

Alya

Solidz module







How is Solidz engineered?

Standard FEM implementation
Loops over elements, Gauss points, tensor operation and spatial dimensions

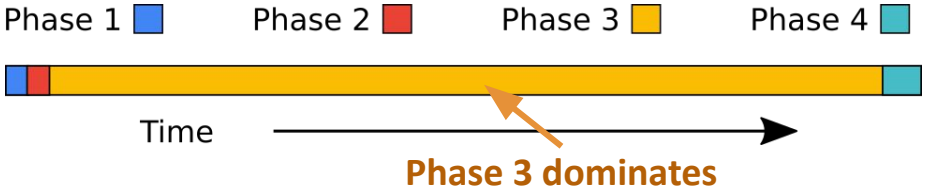
PACKING parameter: compile-time parameter that represent how many elements are processed in parallel

The idea behind the PACKING parameter 💡

- In wide vector architectures: large PACKING values
- In narrow SIMD architectures: small PACKING values

Out-of-the-box vectorization

Step 1: Run on scalar commercial boards →

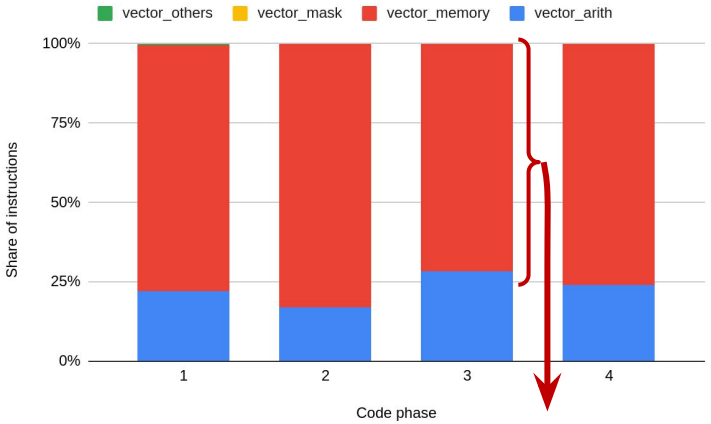


Step 2: Vectorize and Emulate

VECTOR MIX						
Phase	PACKING					
	32	64	128	240	256	512
1	1%	6%	10%	8%	8%	8%
2	6%	8%	10%	9%	8%	8%
3	31%	29%	31%	30%	30%	33%
4	7%	9%	9%	9%	9%	9%

Compile-time parameter

Good vectorization!



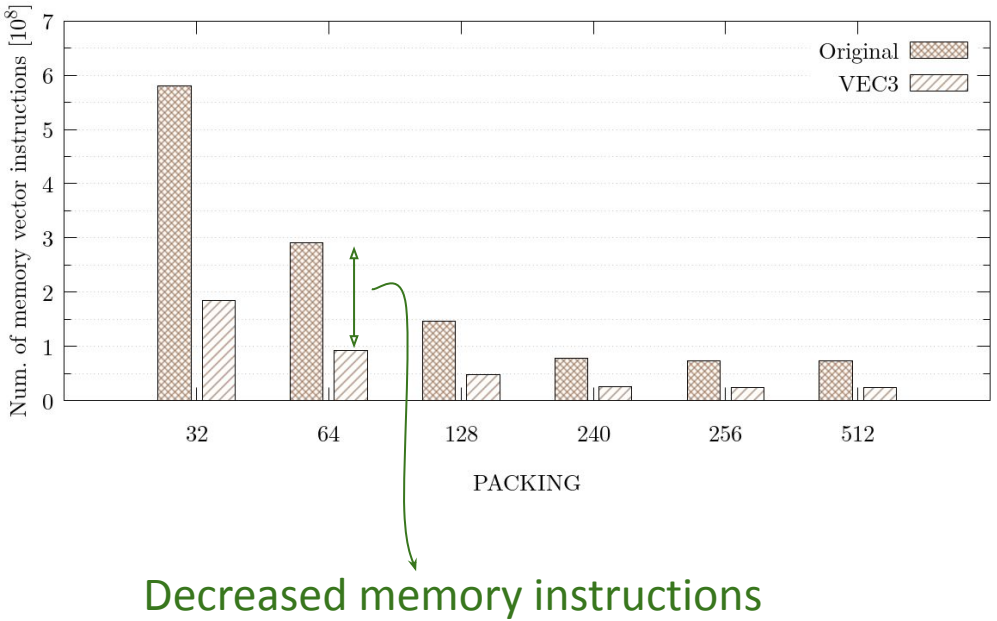
… but we observe that 72% of the vector instructions are memory loads/stores

VEC3 optimization

Step 2: Vectorize and Emulate

```
do ig = 1, ZGAUS
 do bb = 1, ZNODE
  do aa = 1, ZNODE
   do ii = 1, ZDIME
    do jj = 1, ZDIME
     ! Clear alpha
     do v2 = 1, ZVOIGT
      do v1 = 1, ZVOIGT
       do iv = 1, ZVECD
        ! Compute alpha
       enddo
      enddo
     enddo
     do iv = 1, ZVECD
      ! AXPY
     enddo
    enddo
   enddo
  enddo
 enddo
enddo
```

```
do ig = 1, ZGAUS
 do bb = 1, ZNODE
  do aa = 1, ZNODE
   do ii = 1, ZDIME
    do jj = 1, ZDIME
     ! Clear alpha
     do iv = 1, ZVECD
      do v2 = 1, ZVOIGT
       do v1 = 1, ZVOIGT
        ! Compute alpha
       enddo
      enddo
      ! AXPY
     enddo
    enddo
   enddo
  enddo
 enddo
enddo
```



We reorder the code to better reuse intermediate values

Improving vector length utilization

Step 2: Vectorize and Emulate

AvgVL / 256

Phase 4						
Metric	PACKING					
	32	64	128	240	256	512
E_v	1.62%	2.88%	3.02%	3.08%	3.09%	3.09%

Low vector-length on phase 4

We can do some loop reordering…

```
do iv = 1, ZVECD
 if (lmask_e(iv)) then
  do aa = 1, ZNODE
   do ii = 1, ZDIME
    ! WORK true
   enddo
  enddo
 else
  do aa = 1, ZNODE
   do ii = 1, ZDIME
    ! WORK false
   enddo
  enddo
 endif
enddo
```

```
do aa = 1, ZNODE
 do ii = 1, ZDIME
  do iv = 1, ZVECD
   if (lmask_e(iv)) then
    ! WORK true
   else
    ! WORK false
   endif
  enddo
 enddo
enddo
```

Unrolling loops to increase vector body work

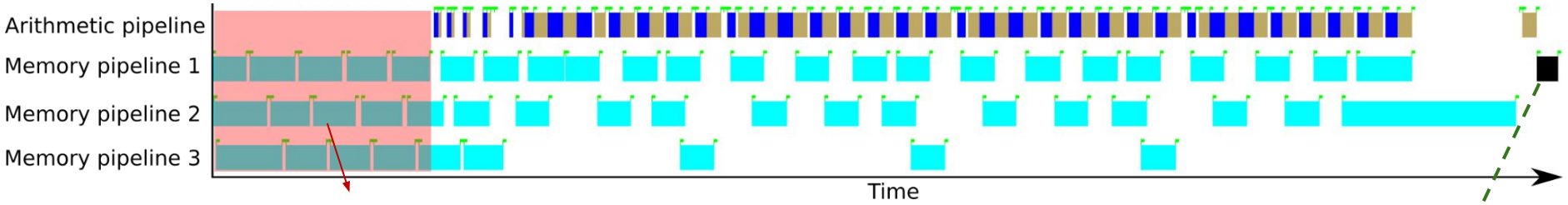
```
do ig = 1, ZGAUS
 do bb = 1, ZNODE
  do aa = 1, ZNODE
   do ii = 1, ZDIME
    do jj = 1, ZDIME
     ! Work
    enddo
   enddo
  enddo
 enddo
enddo
```

```
do ig = 1, ZGAUS
 do bb = 1, ZNODE
  do aa = 1, ZNODE
   do ii = 1, ZDIME
    ! Work_x, Work_y, Work_z
   enddo
  enddo
 enddo
enddo
```

Compiler scheduling

Step 3: Run on FPGA prototype

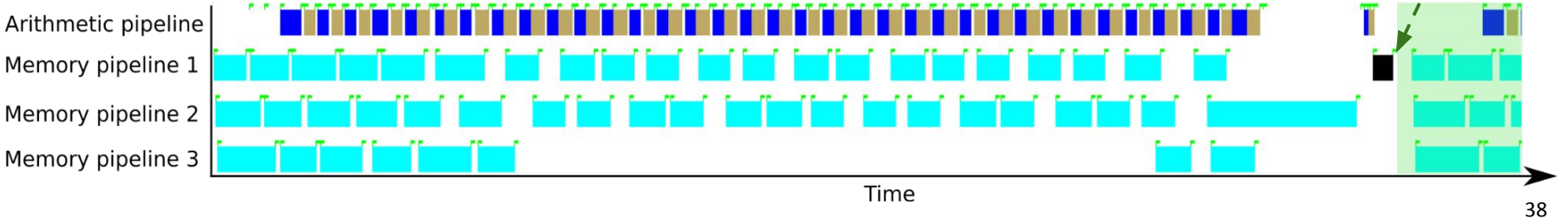
Sub-optimal instruction scheduling



Memory instructions not overlapped with arithmetic

Faster iteration

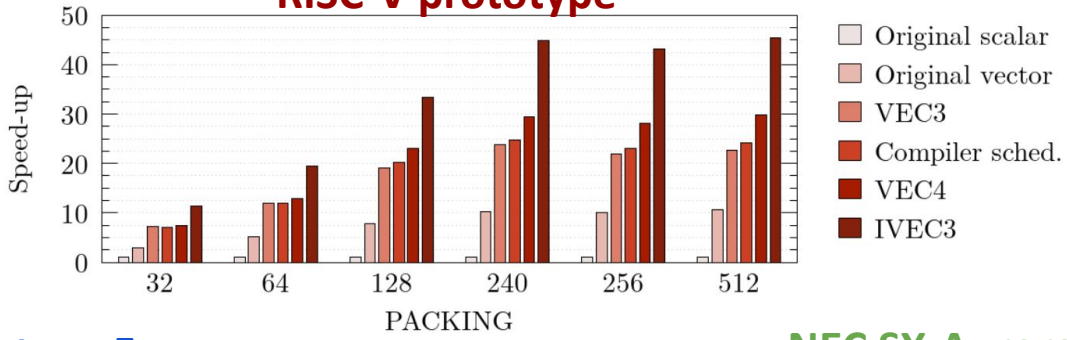
We improved the overlapping in collaboration with the compiler team



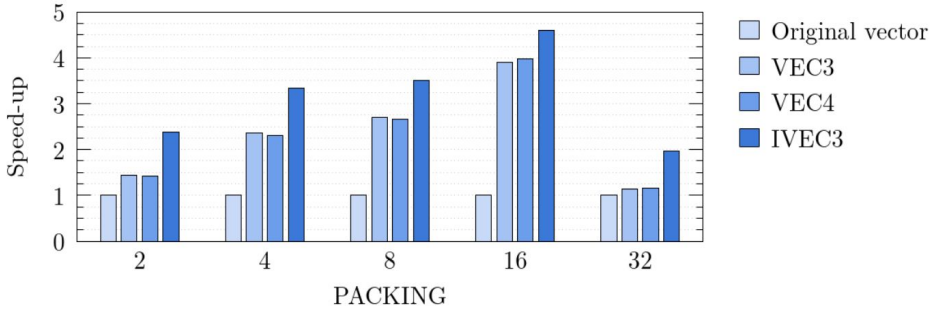
Speedups

Step 4: Cross-Architecture Comparisons

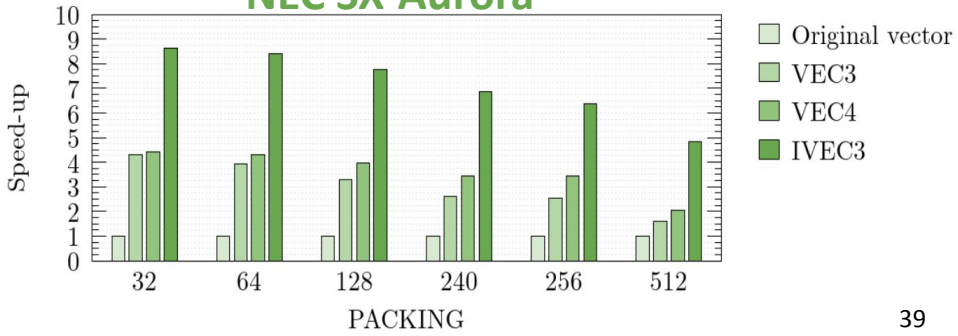
RISC-V prototype



Marenostrum5



NEC SX-Aurora



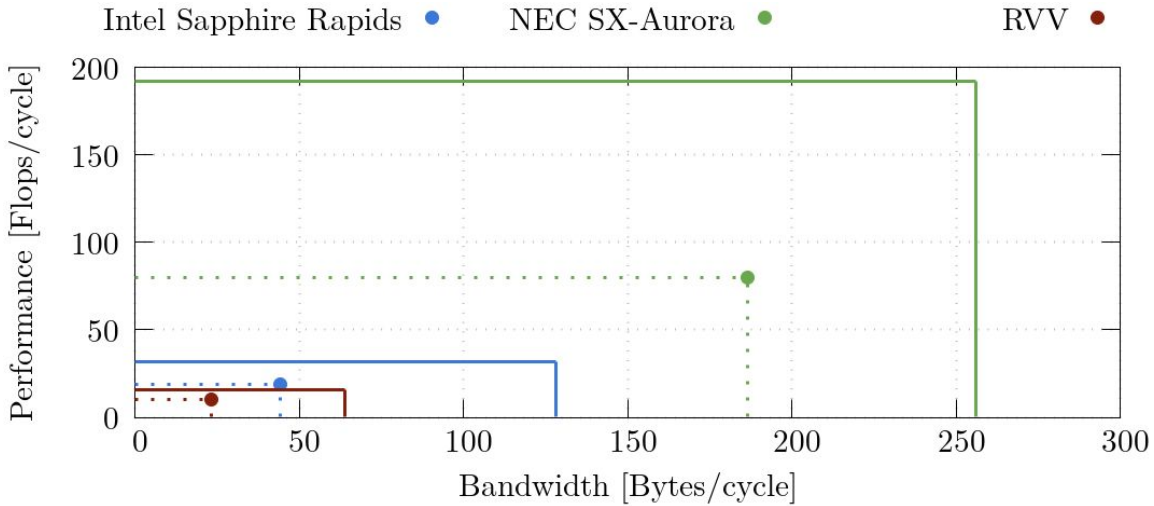
Overall efficiency

Utilization of available bandwidth and vector functional units

Intel 20%

RISC-V prototype 23%

NEC SX-Aurora 30%



Conclusions

- Revisit the goals of the **Software Development Vehicles**: *Feedback for HW* *Feedback for SW*

Provide real use-cases for the prototypes

- Evaluated full scientific applications
- Meeting point for domain scientists and hardware architects
- Prove RISC-V vector arch. are ready for HPC

Discover performance issues

- **Spill** overhead (*Walberla*)
- Subpar efficiency on **low VL** (*SOD2D*)
- Resource **under/over** -provisioning (*Alya*)

Optimize the apps for RISC-V

- Most of the findings **only** with **emulation**
- Common issues in App's loops:
  - **Loops too empty / full** → No reuse / much Spill
  - **Suboptimal nested order** → Low vector-length
- Inefficient compiler scheduler (*Alya*)

Evaluate app portability

- Ran all apps on vastly different platforms.
- Optimizations worked across architectures:
  - *Walberla* → **12%** (RV), **10%** (NEC)
  - *Alya* → **438%** (RV), **460%** (NEC) , **483%** (ARM)

Overall CoE Status

- This presentation is just a sneak peak:

	CEEC						ChEESe					PLASMA-PEPSC					
SDV Status	Alya	Flexi	Neko	NekRS	SOD2D	WaLBerla	FALL3D	HySEA	SeisSol	SPECFEM	Xshells	Alpaka3	BIT1	GENE	GENE-X	PIConGPU	Vlasiator
1) Execute on RV boards	✅	✅	✅	✅	✅	✅	✅	✅	✅	✅	✅	✅	✅	✅	✅	✅	✅
2) Emulate / Vectorize	✅	🏗️	🏗️	💤	🏗️	✅	✅	🏗️	🏗️	💤	🏗️	🏗️	🏗️	🏗️	🏗️	✅	✅
3) EPAC execution	✅	🏗️	💤	💤	🏗️	✅	✅	💤	🏗️	💤	🏗️	💤	🏗️	💤	🏗️	🚧	✅
4) Arch. Comparison	✅	💤	💤	💤	💤	✅	✅	💤	🏗️	💤	💤	💤	💤	💤	💤	🚧	🏗️

- **Future steps**:
  - Turn this table into a green ✅ field
  - Evaluate these applications on future European hardware → dare THE EUPILOT
  - Characterize and generalize performance bottlenecks for long vector architectures

Want to know more?

HW and infrastructure

Kernels and apps

Software Development Vehicles to enable extended and early co-design: a RISC-V and HPC case of study: https://arxiv.org/abs/2306.01797

Acceleration with long vector architectures: Implementation and evaluation of the FFT kernel on NEC SX-Aurora and RISC-V vector extension:http://dx.doi.org/10.1002/cpe.7424

Short reasons for long vectors in HPC CPUs: a study based on RISC-V: https://arxiv.org/abs/2309.06865

Exploiting long vectors with a CFD code: a co-design show case: https://arxiv.org/abs/2411.00815

RAVE: RISC-V Analyzer of Vector Execution QEMU tracing plugin https://arxiv.org/abs/2409.13639

Exploring RISC-V long vector capabilities: A case study in Earth Sciences: http://dx.doi.org/10.1016/j.future.2025.107932

pablo.vizcaino@bsc.es
marc.blancafort@bsc.es