Vectorization studies of scientific codes

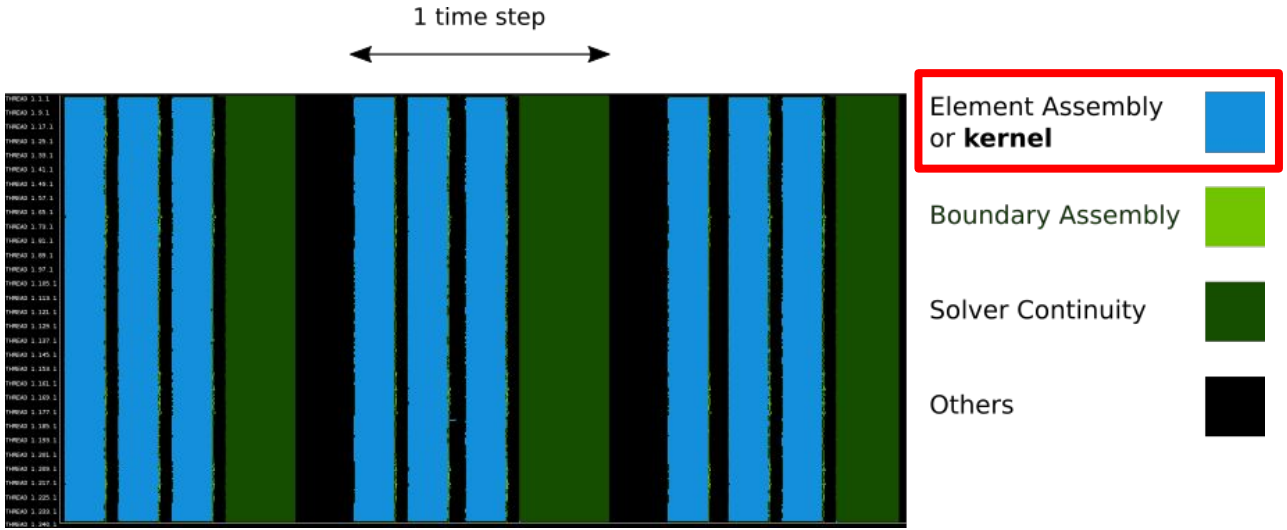
CFD code: Alya

CEEC

Centre of Excellence in Exascale CFD

Vectorization of a real CFD code (Alya)

- Alya is a modular code → We study the module called “Nastin”
- “BLOCK_SIZE”
  - Allocates data structures in a vector-friendly way
  - Values under study → [16, 64, 128, 240, 256, 512]



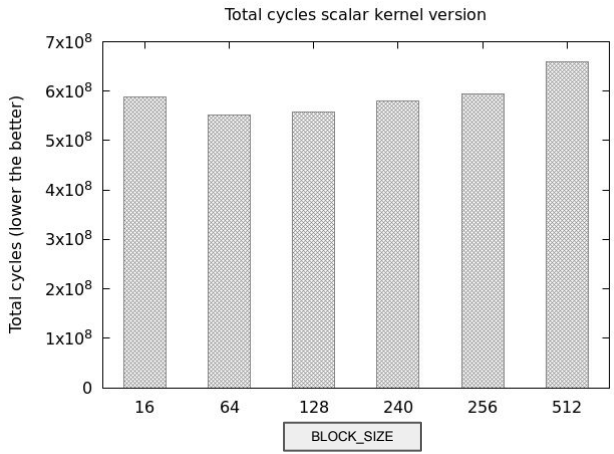
Alya mini-app

- We worked on a mini-app that mimics the behaviour of the Assembly of Alya
- We divided the mini-app in “phases”
  - Mini-app phases are regions of codes with one or more loops
  - We are interested in loops because is where there is potential for vectorization
  - 8 phases identified: P1+P2+P3+P4+P5+P6+P7+P8 = mini-app
- We based our study and optimization on the autovectorization capabilities
  - No intrinsics portability is preserved

1st step: Run on commercial RISC-V platforms (no vector)

- Phases taking longer (6,3,7,4) correspond to compute intensive regions
- Phases lasting less (5,2,8,1) are memory bound regions
- BLOCK_SIZE parameter has almost no influence on the execution (5% coefficient of variation)

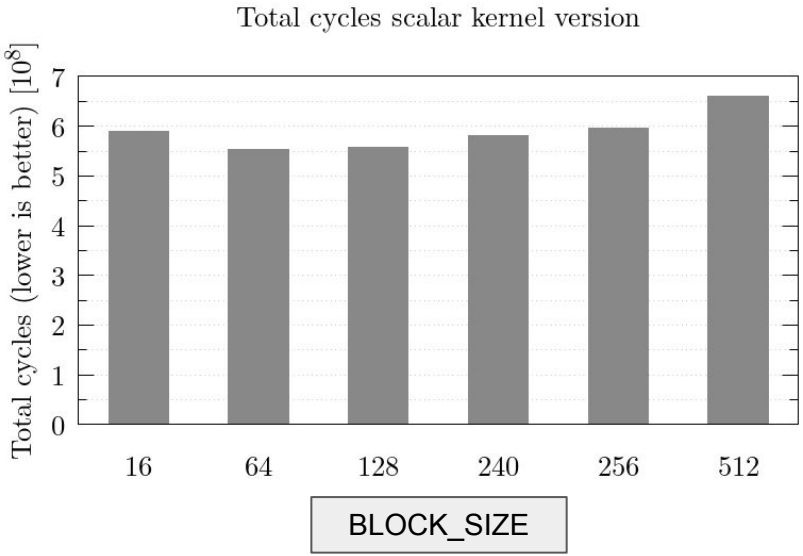
Phase	1	2	3	4	5	6	7	8
% of total cycles	1,29%	3,33%	19,80%	14,45%	3,49%	40,99%	14,68%	1,96%



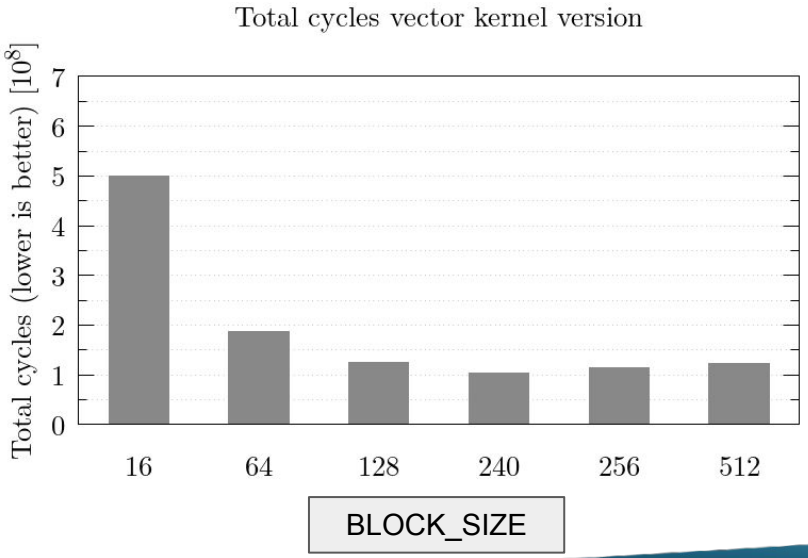
Commercial RISC-V platform (scalar CPU)

1st step: Enabling auto-vectorization

- Auto-vectorization results without touching any line of code
- BLOCK_SIZE parameter strongly influences when executing with vectors



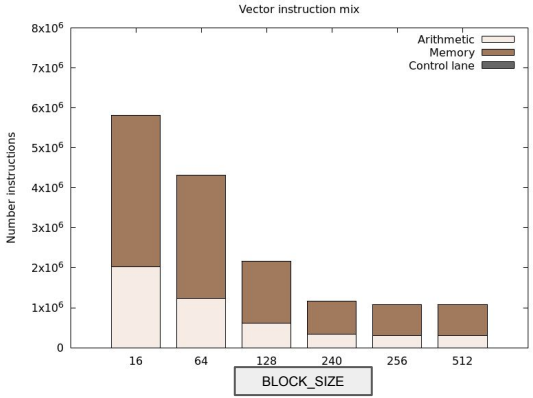
Enabling Compiler Auto-vec



2nd step: Emulation supporting RVV (RAVE)

BLOCK_SIZE	Phase 1	2	3	4	5	6	7	8
16	0,00%	0,00%	1,84%	0,00%	0,00%	0,95%	24,64%	0,00%
64	0,00%	0,00%	12,73%	17,37%	17,86%	21,58%	25,87%	0,00%
128	0,00%	0,00%	16,05%	16,80%	17,94%	20,39%	25,23%	0,00%
240	0,00%	0,00%	15,31%	16,45%	16,82%	19,90%	23,90%	0,00%
256	0,00%	0,00%	15,36%	16,21%	15,88%	19,78%	24,23%	0,00%
512	0,00%	0,00%	16,65%	18,19%	18,47%	21,82%	26,20%	0,00%

30,00%
15,00%
0,00%



Analysis of % of vector instructions:

- Higher BLOCK_SIZE helps the compiler to insert more vector instructions
- Higher BLOCK_SIZE reduces the total number of vector instructions
- 70% of vector instructions are memory type

3rd step: Run on EPAC mapped into FPGA

BLOCK_SIZE	Phase 1	2	3	4	5	6	7	8
16	0,00%	0,00%	15,72%	0,00%	0,00%	7,66%	73,30%	0,00%
64	0,00%	0,00%	72,59%	76,62%	57,73%	86,85%	77,70%	0,00%
128	0,00%	0,00%	81,94%	79,36%	64,01%	88,96%	79,59%	0,00%
240	0,00%	0,00%	83,69%	83,08%	70,75%	90,61%	81,94%	0,00%
256	0,00%	0,00%	83,76%	83,03%	71,29%	90,26%	82,83%	0,00%
512	0,00%	0,00%	85,74%	87,59%	80,61%	91,14%	88,50%	0,00%

100,00%
75,00%
50,00%
25,00%
0,00%

Analysis of % of vector cycles:

- High vCPI → we are computing several elements per instruction (GOOD)
- AVL == BLOCK_SIZE → the more elements we process per vector instruction, the less vector instructions we execute (GOOD)

BLOCK_SIZE	vCPI	AVL	Number vector instructions
16	9.71	16	14.3×10^5
64	23.39	64	19.1×10^5
128	28.56	128	9.6×10^5
240	41.19	240	5.1×10^5
256	43.10	256	4.7×10^5
512	45.30	256	4.7×10^5

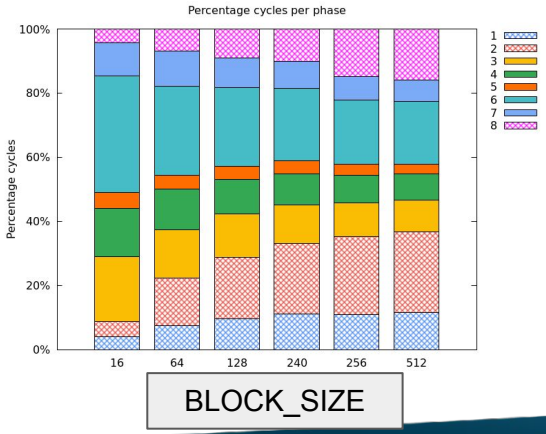
vCPI, AVL and # vector instructions phase 6

3rd step: Run on EPAC mapped into FPGA

BLOCK_SIZE	Phase 1	2	3	4	5	6	7	8
16	0,00%	0,00%	15,72%	0,00%	0,00%	7,66%	73,30%	0,00%
64	0,00%	0,00%	72,59%	76,62%	57,73%	86,85%	77,70%	0,00%
128	0,00%	0,00%	81,94%	79,36%	64,01%	88,96%	79,59%	0,00%
240	0,00%	0,00%	83,69%	83,08%	70,75%	90,61%	81,94%	0,00%
256	0,00%	0,00%	83,76%	83,03%	71,29%	90,26%	82,83%	0,00%
512	0,00%	0,00%	85,74%	87,59%	80,61%	91,14%	88,50%	0,00%

100,00%
75,00%
50,00%
25,00%
0,00%

- Phases 1, 2 and 8 are not vectorized (pattern colored in plot)
- Next step: focus in vectorize phase 2
  - Costing 30% of time ⚠️



Example of optimization: phase 2 aka VEC2

Problem

- Compiler unable to vectorize loop, not sure of VECTOR_DIM value

```
subroutine nsi_miniapp(VECTOR_DIM, pnode, pgaus, list_elements)
```

```
loop not vectorized: unsafe dependent memory operations in loop.
```

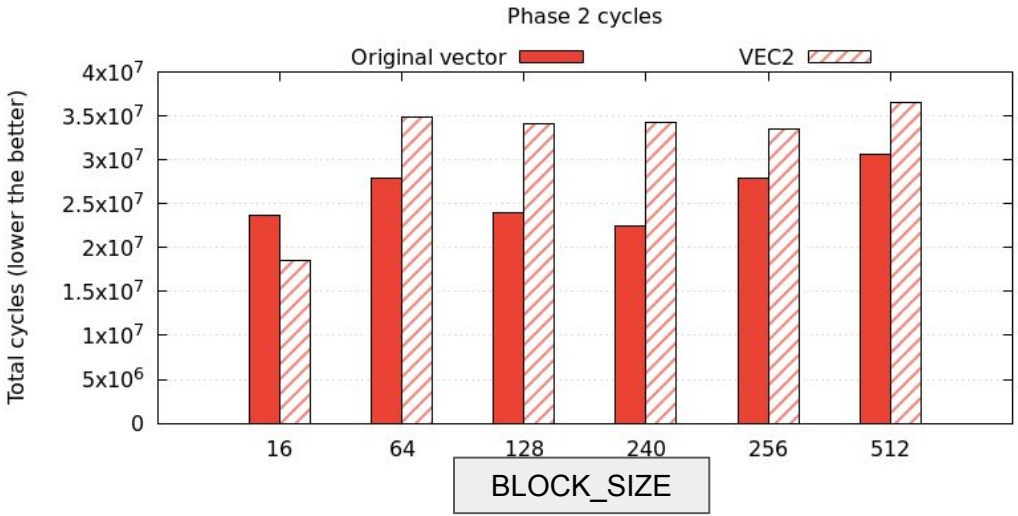
Solution

- We know VECTOR_DIM value

```
integer(ip), parameter :: VECTOR_DIM = BLOCK_SIZE
```

Optimization – VEC2

- Enabled vectorization in phase 2
  - Performance get worst instead of improving
  - AVL of vector instructions is low! ⚠️
    We are not taking advantage of the full-VL. Why?



Optimization - VEC2+VL

Problem

- pnode comes from input, we do not know its value
- Experimentally found pnode << VECTOR_DIM

```
do ivect = 1,VECTOR_DIM
  do inode = 1,pnode
    !WORK
  end do
end do
```

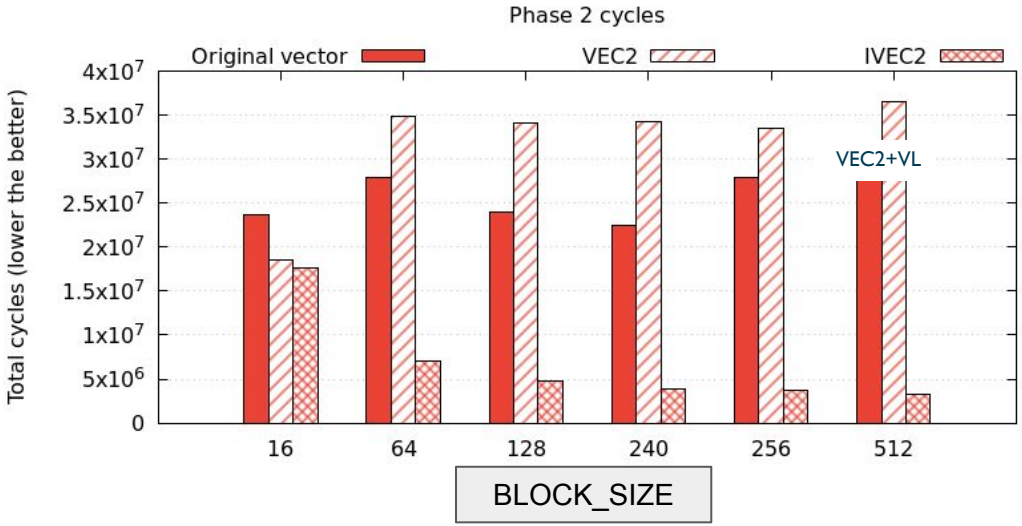
Solution

- Swap induction variables

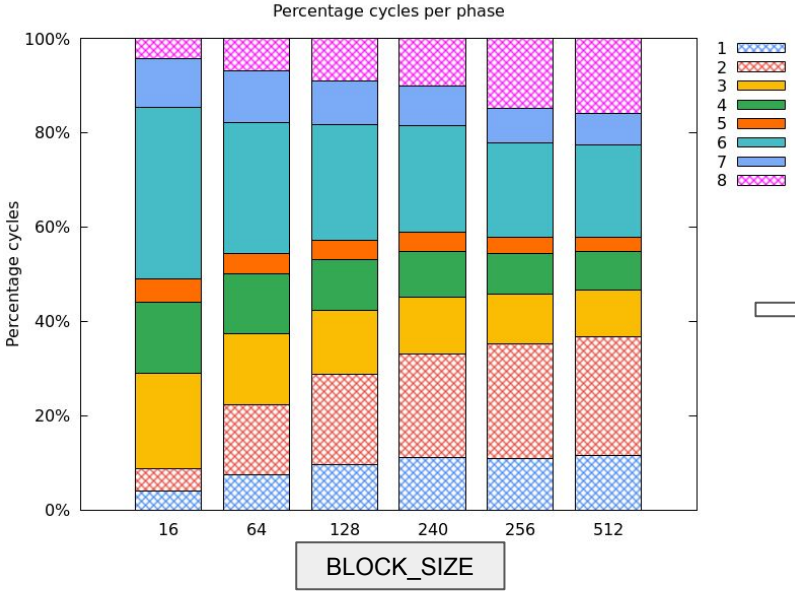
```
do inode = 1,pnode
  do ivect = 1,VECTOR_DIM
    !WORK
  end do
end do
```

Optimization VEC2+VL: results

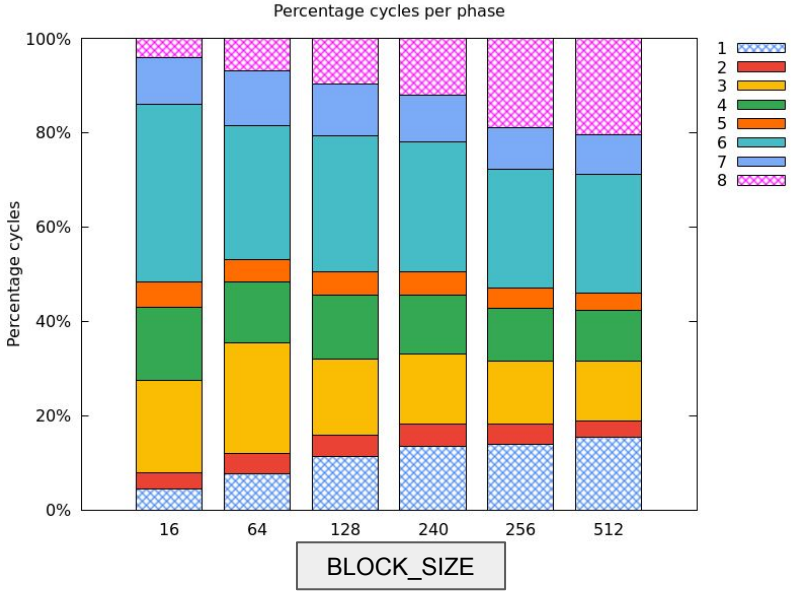
- Improved AVL vectorization in phase 2
  - Vector instructions running with average vector length == BLOCK_SIZE



Alya preliminary results - VEC2+VL



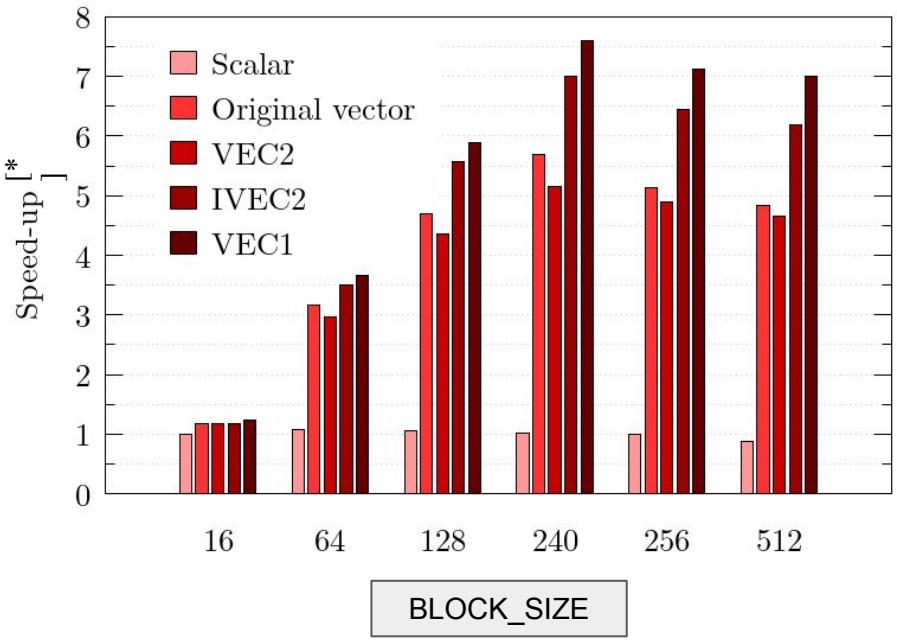
Before optimization



After optimization

Evaluation: RISC-V vector prototype

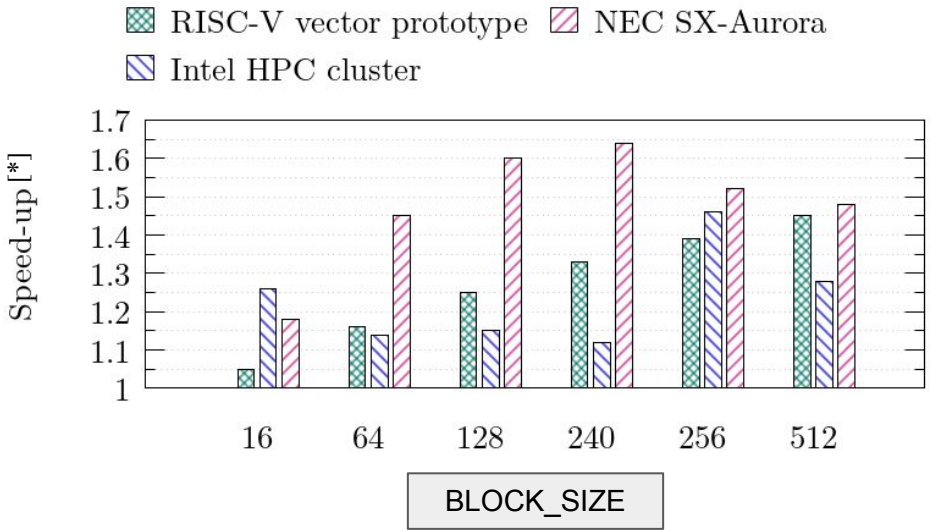
- After a detailed study and manual optimizations, we achieve a peak of 7.6x speedup (VEC1)
- Code remains portable
  No intrinsics!



[*] Speed-up defined as: scalar $BLOCK\_SIZE_{16}$ / optimized vector

Portability across other HPC platforms

- Optimizations portable to other architectures
  - “Traditional” cluster (Intel x86)
  - Long-vector architecture (NEC SX-Aurora)



[*] Speed-up defined as: vanilla vector / optimized vector

Take home message

- We leveraged the EPI Software Development Vehicles (SDVs) to study and improve vectorization of a complex CFD code (Alya) written in Fortran
- Vectorization techniques improve performance on EPAC – VEC and are portable
- Similar studies are on going for several scientific codes part of EU CoEs

CEEC
Centre of Excellence in Exascale CFD

PLASMA PEPSC

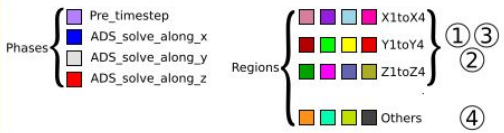
ChEESE

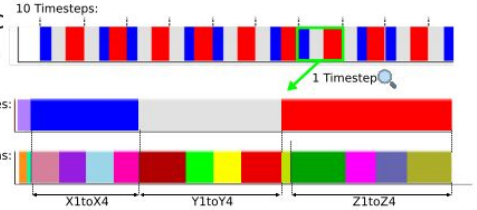
Blancafort, Marc, et al. "Exploiting long vectors with a CFD code: a co-design show case." 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2024.

MiniFALL3D

ChEESE

FALL3D is an eulerian model for atmospheric transport and deposition of passive particles.





KT_RHS function

- Fine grained execution
- Called 88k times
- Called by all ADS phases
- Cumulative 57% of Timestep

Not vector-friendly ⓪

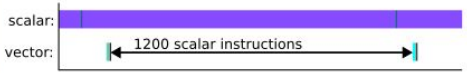
```
do i = ips, ipe
  ! Do work...
  if (i.eq.ips) flux(1) = F_m+P_m
  if (i.eq.ips) flux(2) = F_p+P_p
end do
```

Vector-friendly ①

```
flux(1) = !...
do i = ips, ipe
  ! Do work...
end do
flux(2) = !...
```

Fortran constructs

Compiler acts conservatively, it introduces extra scalar array copies.



array operations ⟶ do-loops ②

function ⟶ subroutine ③

Other Regions

Loop collapsing, do-loops, and loop swap.

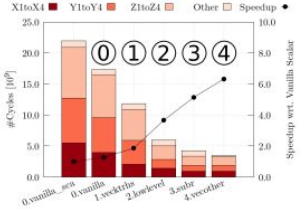
Not vectorized

```
1 CB%w(:,:,:) = (1.0_rp-stime)*my_w1(:,:,:) + stime*my_w2(:,:,:)
```

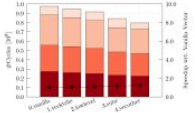
Vectorized (with VL=256) ④

```
1 ijksize = (my_kbe_1h-my_kbs_1h+1)
2   * (my_ipe-my_ips+1) * (my_jpe-my_jps+1)
3 ! 1D-to-3D pointer assignments
4 cb_w_1d(1:ijksize) => CB%w(:,:,:)
5 my_w1_1d(1:ijksize) => my_w1(:,:,:)
6 my_w2_1d(1:ijksize) => my_w2(:,:,:)
7 ! 1D do-loop
8 do i = 1, ijksize
9   cb_w_1d(i) = (1.0_rp-stime)*my_w1_1d(i) + stime*my_w2_1d(i)
10 done
```

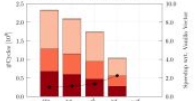
RISC-V



MareNostrum5 - x86



NEC SX-Aurora



Changes architecture agnostic

Overall performance benefits