RAVE: RISC-V Analyzer of Vector Executions

a QEMU tracing plugin

Pablo Vizcaino, Filippo Mantovani, Jesus Labarta, Roger Ferrer

RISC-V Technical Session, October 10th

European Processor Initiative

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Outline

First half:

- Background on RVV
- EPAC chip
- Simulation environment
  - Why do we need **RAVE**?
  - How does **RAVE** work?
- Performance and use cases

Second half:

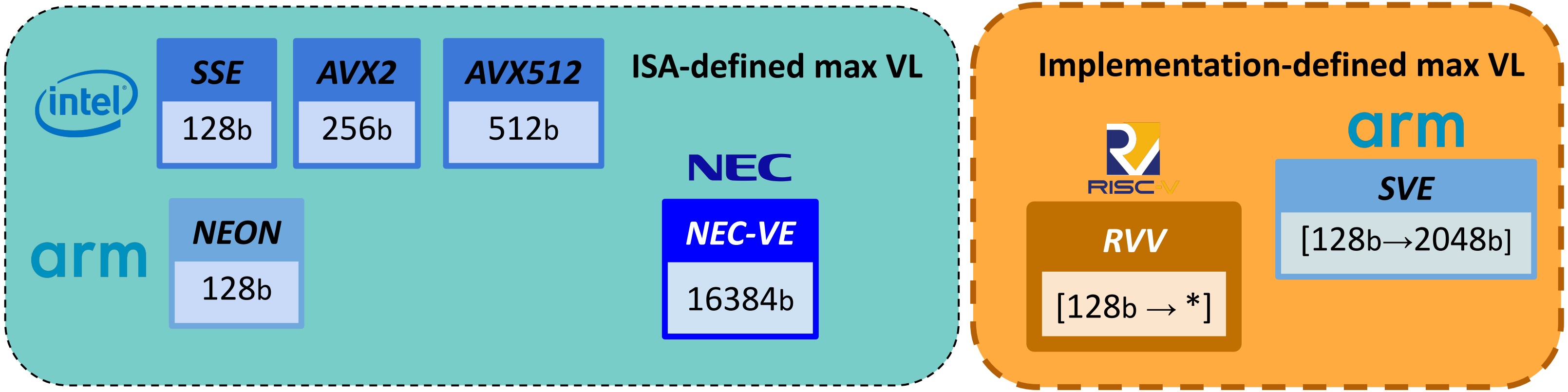
- RAVE demo

Motivation

- **RVV** (RISC-V Vector extension) is RV’s bet for High Performance Computing

Motivation

- **RVV** (RISC-V Vector extension) is RV’s bet for High Performance Computing



ISA-defined max VL

SSE 128b

AVX2 256b

AVX512 512b

NEON 128b

NEC-VE 16384b

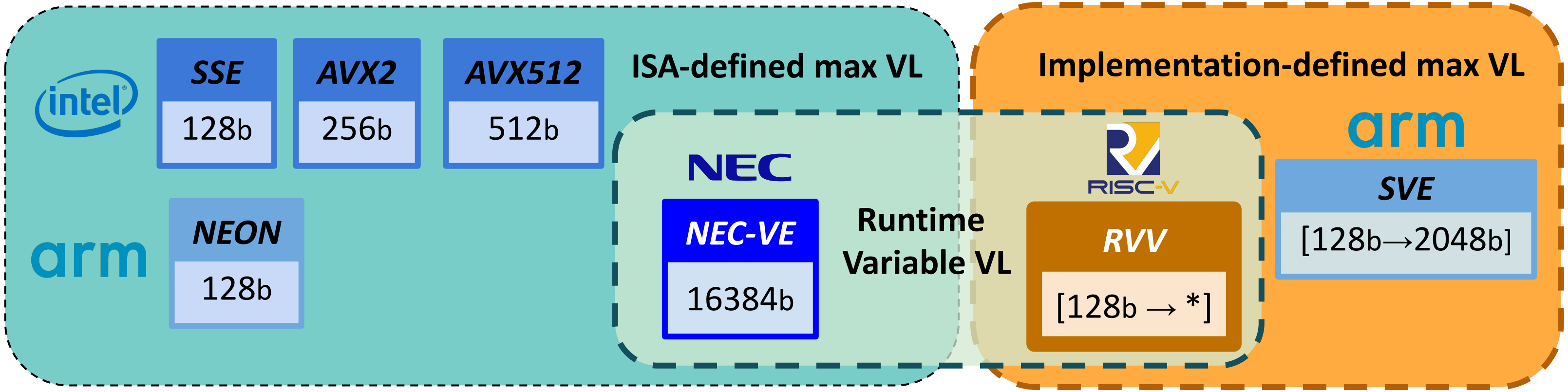
Implementation-defined max VL

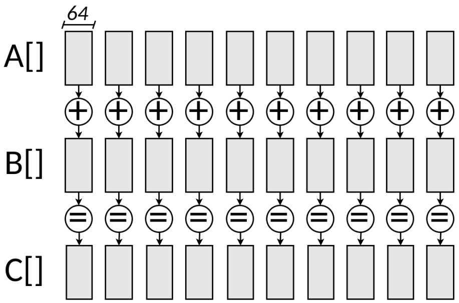
RVV [128b → *]

SVE [128b→2048b]

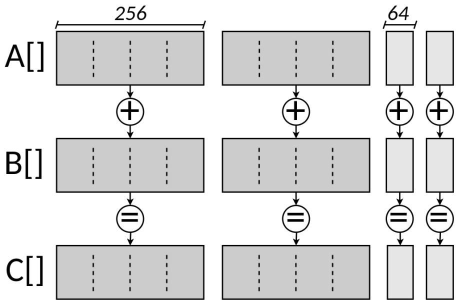
Motivation

- **RVV** (RISC-V Vector extension) is RV's bet for High Performance Computing

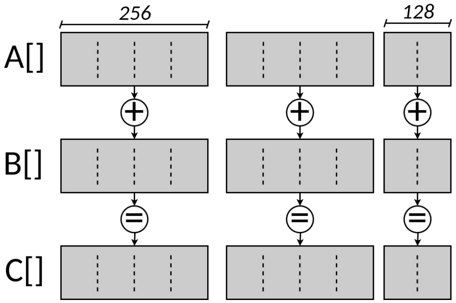




Scalar processor



SIMD (e.g., AVX2)

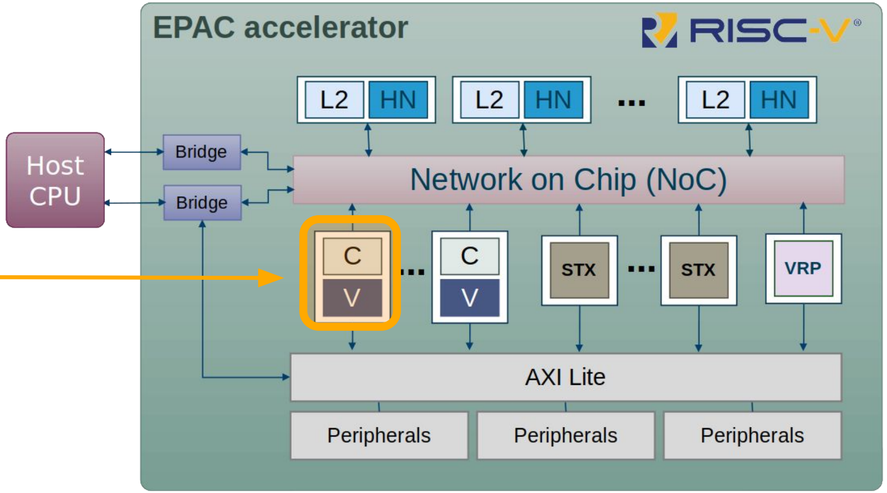


Variable VL (e.g., RVV)

Who is implementing this technology?

European Processor Initiative (**EPI**)

- **Rhea:** arm-based general purpose CPU
- **EPAC:** European Processor Accelerator
  - Based on **RISC-V**, an open ISA
  - Many tiles: VRP, STX, **VEC**

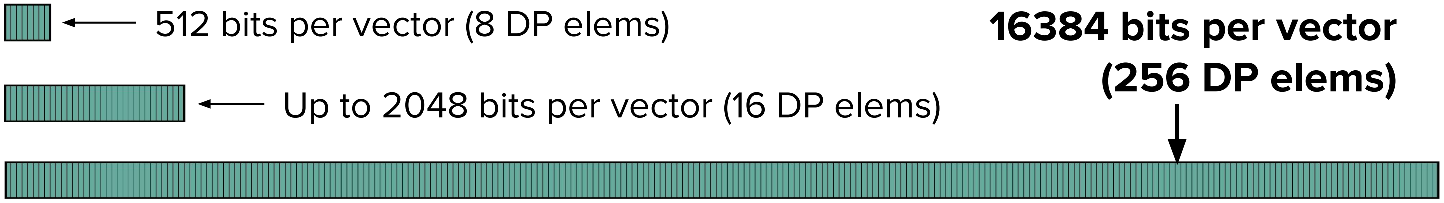


Very large vector length:

intel – AVX512

arm – SVE

NEC / epi European Processor Initiative



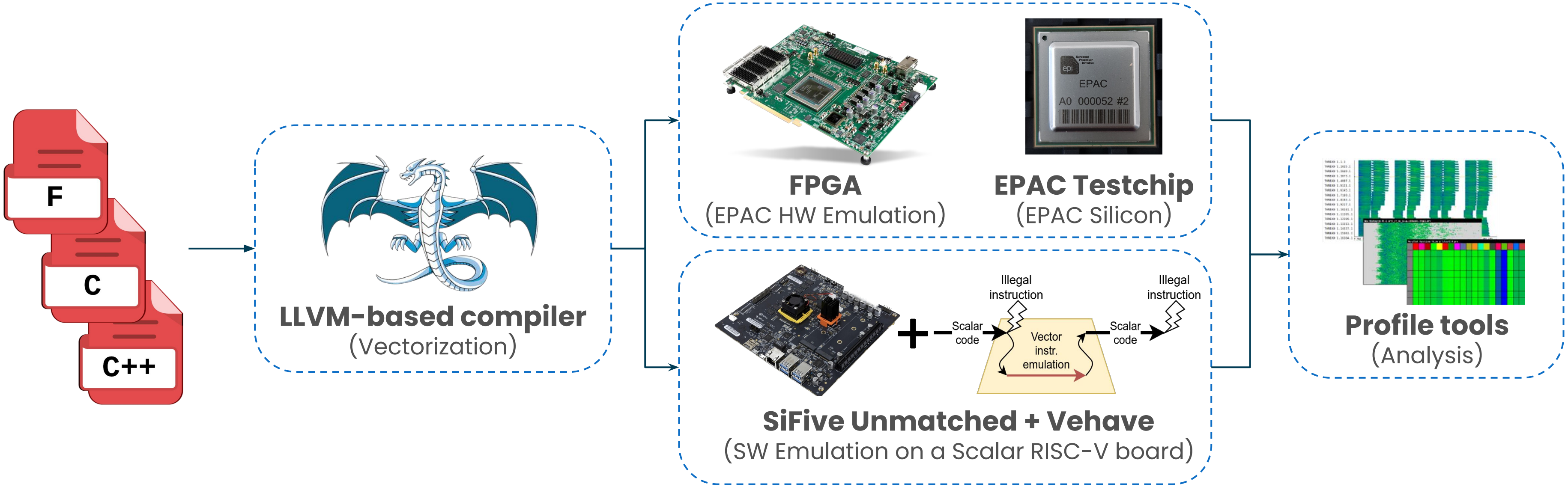
512 bits per vector (8 DP elems)

Up to 2048 bits per vector (16 DP elems)

16384 bits per vector (256 DP elems)

How can you develop code for this accelerator?

- BSC has the **Software Development Vehicles (SDV)**[1]**:**



LLVM-based compiler
(Vectorization)

FPGA
(EPAC HW Emulation)

EPAC Testchip
(EPAC Silicon)

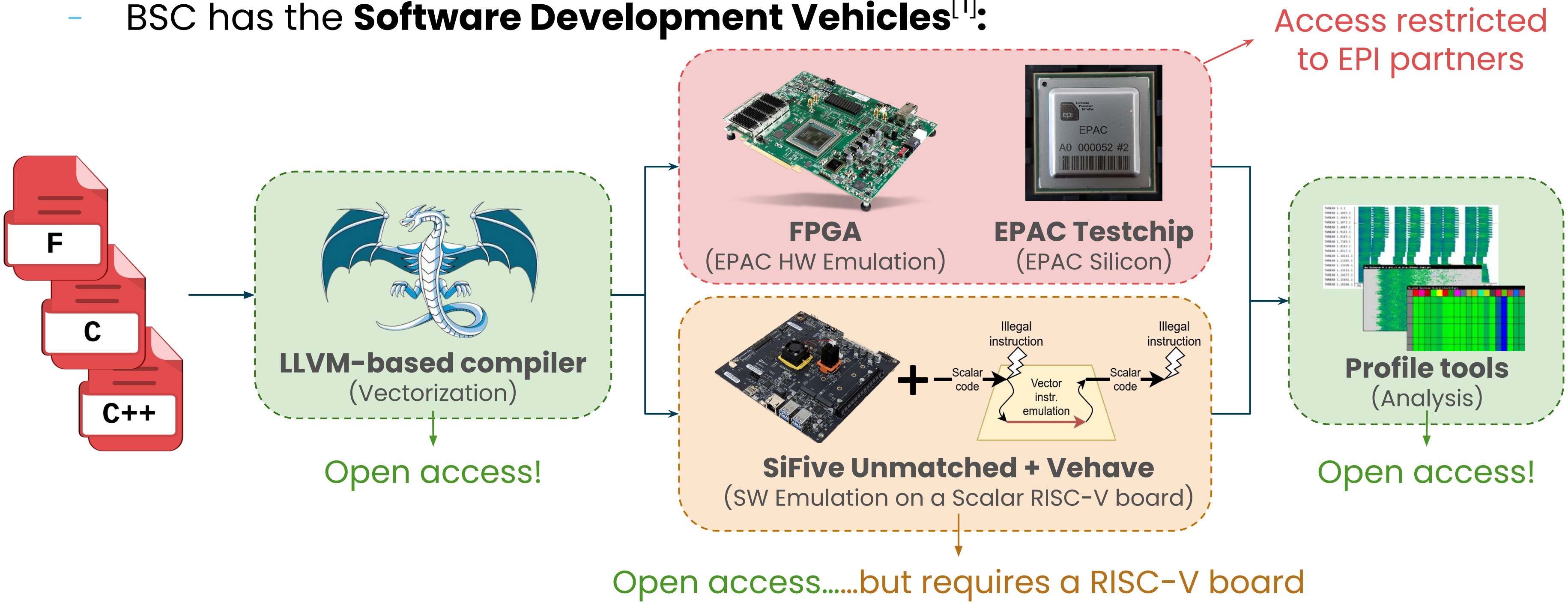
SiFive Unmatched + Vehave
(SW Emulation on a Scalar RISC-V board)

Profile tools
(Analysis)

[1] *Filippo Mantovani et al. (2023, May). Software Development Vehicles to enable extended and early co-design: a RISC-V and HPC case of study. In International Conference on High Performance Computing (pp. 526-537). Cham: Springer Nature Switzerland.*

How can you develop code for this accelerator?

- BSC has the **Software Development Vehicles**[1]:



Access restricted to EPI partners

F

C

C++

LLVM-based compiler (Vectorization)

Open access!

FPGA (EPAC HW Emulation)

EPAC Testchip (EPAC Silicon)

SiFive Unmatched + Vehave (SW Emulation on a Scalar RISC-V board)

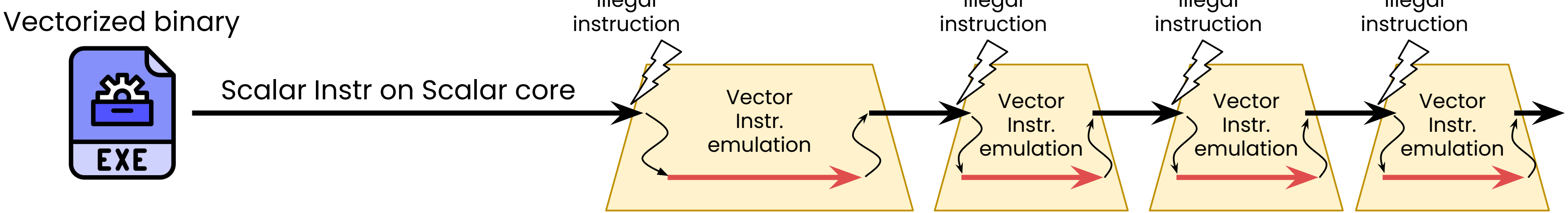
Open access……but requires a RISC-V board

Profile tools (Analysis)

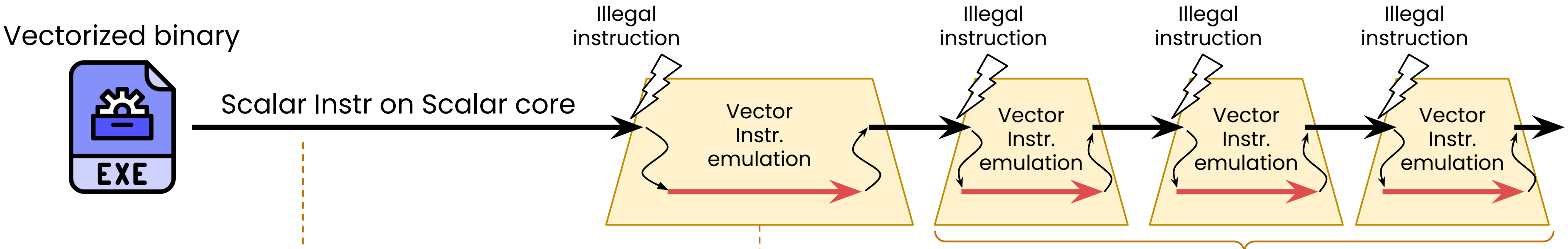
Open access!

[1] *Filippo Mantovani et al. (2023, May). Software Development Vehicles to enable extended and early co-design: a RISC-V and HPC case of study. In International Conference on High Performance Computing (pp. 526-537). Cham: Springer Nature Switzerland.*

Another issue with simulation (Vehave)



Another issue with simulation (Vehave)



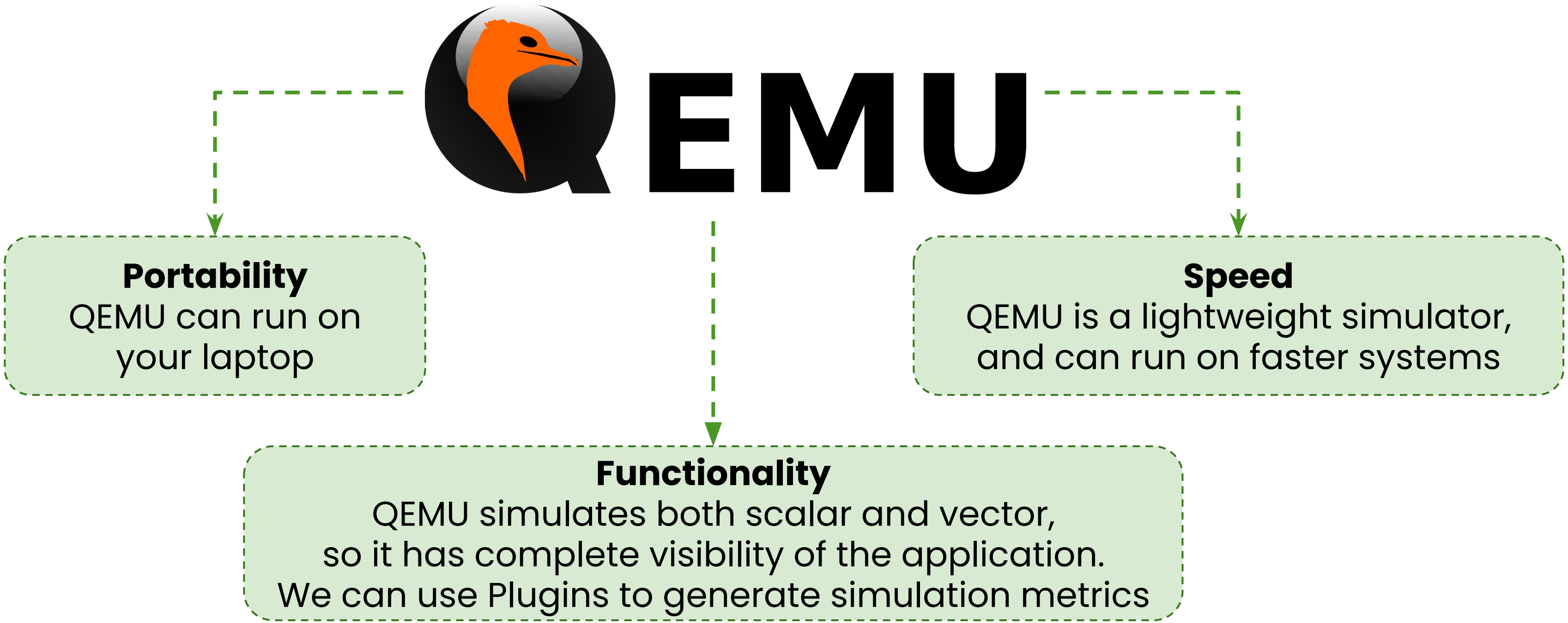
Portability
Vehave requires RISC-V hardware to run the scalar instructions

Speed
Entering the O.S. on each vector instruction is expensive

Functionality
Vehave only has visibility of vector regions (i.e. it cannot generate scalar instruction metrics)

What is the alternative?

At the start of the EPI project, QEMU did not support RVV, but now it does!



Portability
QEMU can run on your laptop

Speed
QEMU is a lightweight simulator, and can run on faster systems

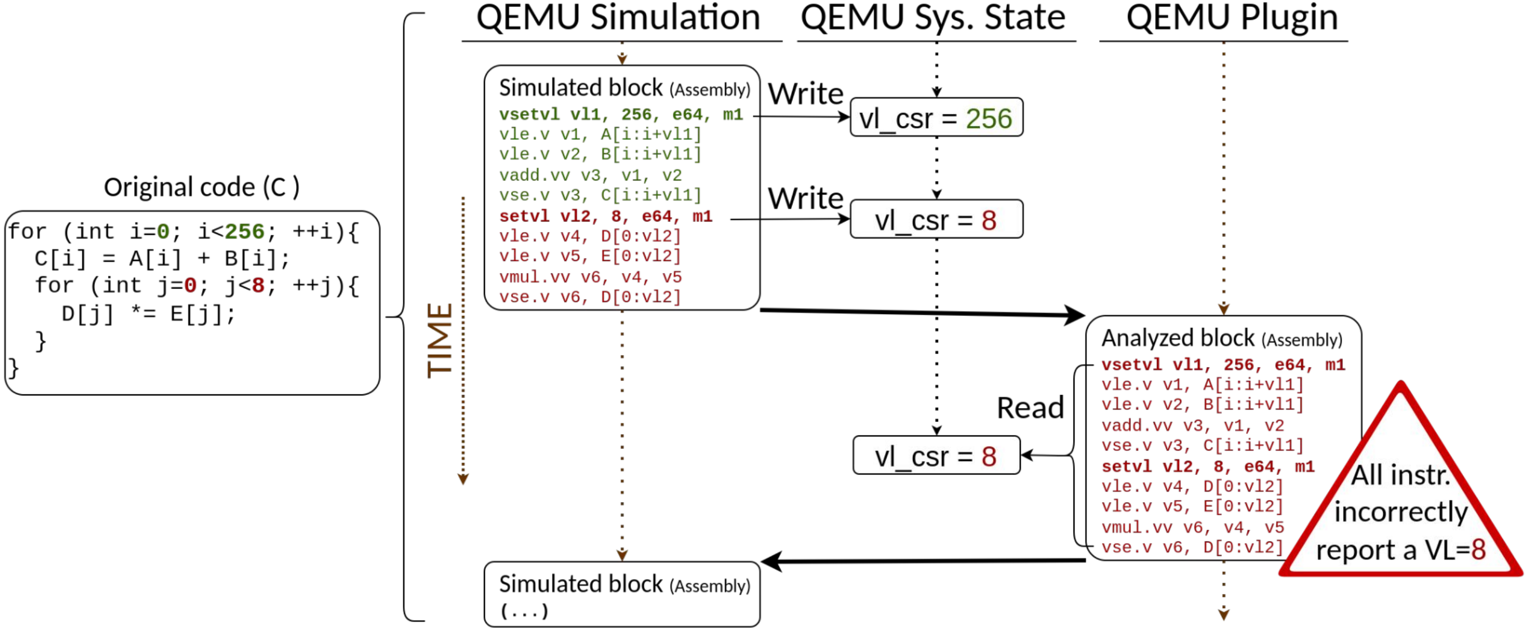
Functionality
QEMU simulates both scalar and vector, so it has complete visibility of the application. We can use Plugins to generate simulation metrics

Modifications to QEMU

1. Changing the ***RV_VLEN_MAX*** parameter in `./target/riscv/cpu.h` :
   a. By default, it's **1024** bits ⟶ we change it to **16384** bits (VLEN of EPAC)

Modifications to QEMU

1. Changing the ***RV_VLEN_MAX*** parameter in `./target/riscv/cpu.h` :
    a. By default, it's **1024** bits ⟶ we change it to **16384** bits (VLEN of EPAC)
2. Changing the size of the translation block to **1** instruction:



Original code (C)

```
for (int i=0; i<256; ++i){
  C[i] = A[i] + B[i];
  for (int j=0; j<8; ++j){
    D[j] *= E[j];
  }
}
```

Simulated block (Assembly)

```
vsetvl vl1, 256, e64, m1
vle.v v1, A[i:i+vl1]
vle.v v2, B[i:i+vl1]
vadd.vv v3, v1, v2
vse.v v3, C[i:i+vl1]
setvl vl2, 8, e64, m1
vle.v v4, D[0:vl2]
vle.v v5, E[0:vl2]
vmul.vv v6, v4, v5
vse.v v6, D[0:vl2]
```

Analyzed block (Assembly)

```
vsetvl vl1, 256, e64, m1
vle.v v1, A[i:i+vl1]
vle.v v2, B[i:i+vl1]
vadd.vv v3, v1, v2
vse.v v3, C[i:i+vl1]
setvl vl2, 8, e64, m1
vle.v v4, D[0:vl2]
vle.v v5, E[0:vl2]
vmul.vv v6, v4, v5
vse.v v6, D[0:vl2]
```

Simulated block (Assembly)

```
(...)
```

RAVE plugin: RISC-V Analyzer of Vector Executions

- We design a QEMU plugin to profile and analyze the vectorization:

Matching Vehave functionalities

- Print the sequence of executed vector instructions
- For each instruction, report:
  - Vector Length (VL) and
  - Element Width (8,16,32,64b)
  - Length Multiplier (LMUL)
  - Used Registers
  - ....
- Provide an API called from the application code
- Create a PRV execution trace

RAVE plugin: RISC-V Analyzer of Vector Executions

- We design a QEMU plugin to profile and analyze the vectorization:

Matching Vehave functionalities

- Print the sequence of executed vector instructions
- For each instruction, report:
  - Vector Length (VL) and
  - Element Width (8,16,32,64b)
  - Length Multiplier (LMUL)
  - Used Registers
  - ….
- Provide an API called from the application code
- Create a PRV execution trace

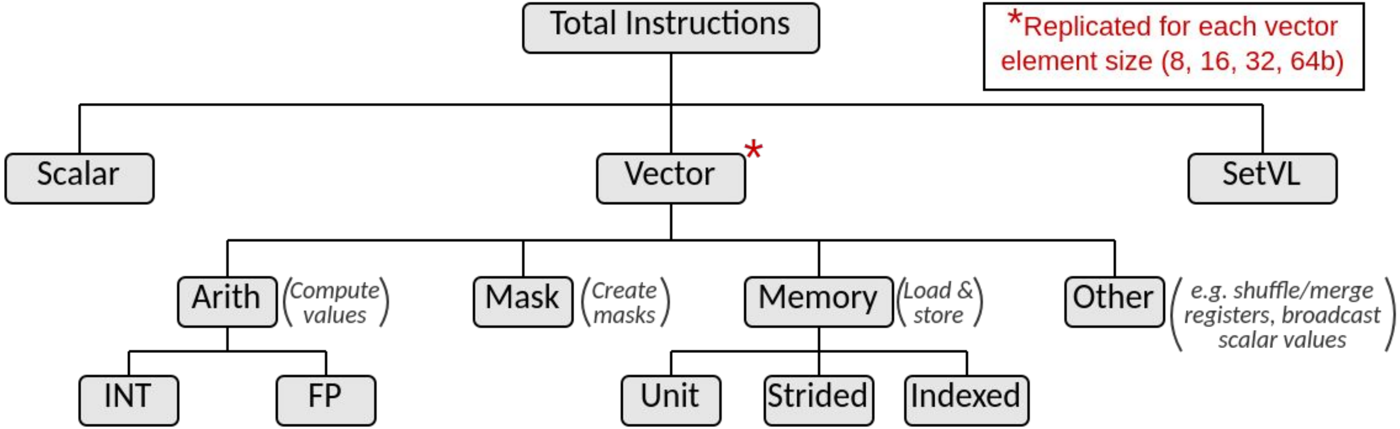
+

Extend them in RAVE

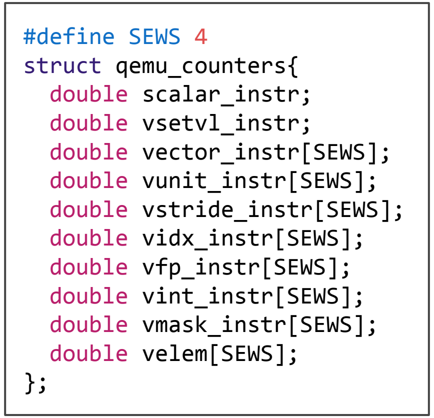
- Count the scalar instructions between vector ones
- Compute vectorization metrics on each instrumented block
- Generate a report at the end of the emulation to summarize these vector metrics

Vectorization metrics

- We classify all instructions following this diagram:



*Replicated for each vector element size (8, 16, 32, 64b)

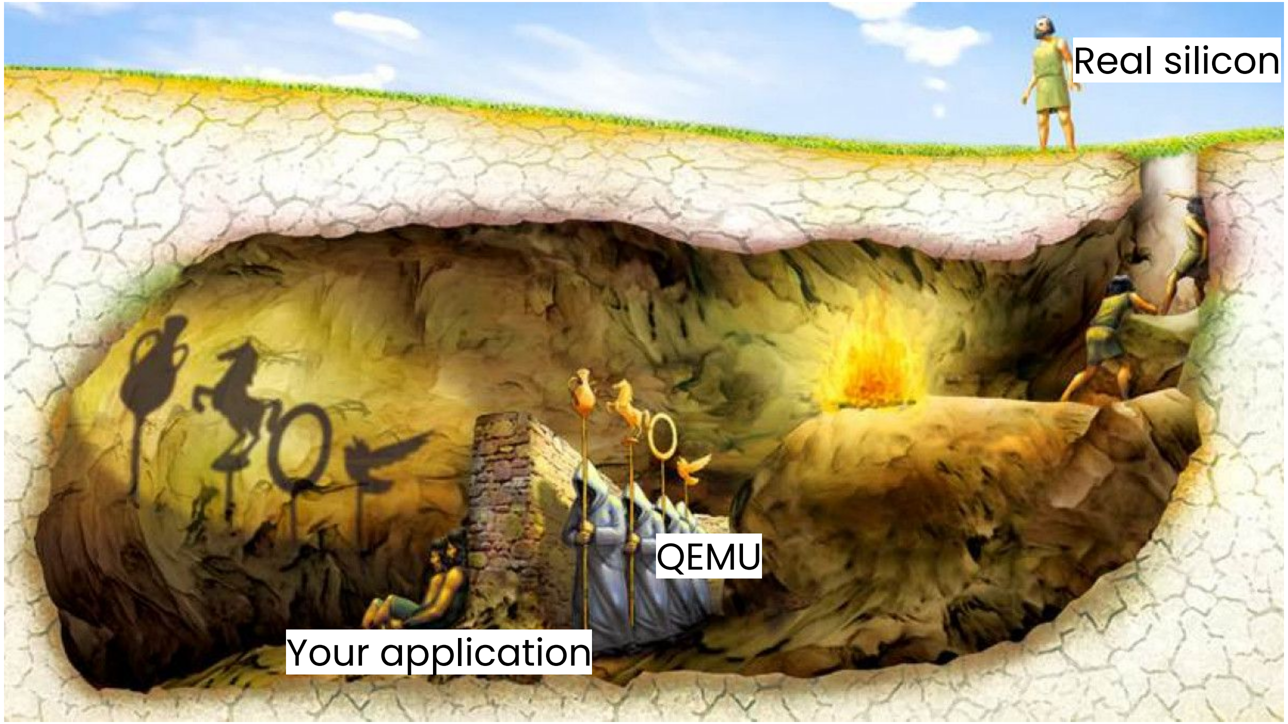


```
#define SEWS 4
struct qemu_counters{
  double scalar_instr;
  double vsetvl_instr;
  double vector_instr[SEWS];
  double vunit_instr[SEWS];
  double vstride_instr[SEWS];
  double vidx_instr[SEWS];
  double vfp_instr[SEWS];
  double vint_instr[SEWS];
  double vmask_instr[SEWS];
  double velem[SEWS];
};
```

- We derive metrics such as:
  - ***VectorMix*** = *Vector / Total_Instructions*
  - **V. Flop/Byte** = Vector.*Arith.FP* / *Vector.Memory*

Controlling the trace with the RAVE API

- Emulation is transparent to applications, so they cannot call RAVE directly



Controlling the trace with the RAVE API

- Emulation is transparent to applications, so they cannot call RAVE directly
- RAVE/QEMU only sees simulated instructions
  - We must communicate through them…
  - … but without changing the program behavior

Controlling the trace with the RAVE API

- Emulation is transparent to applications, so they cannot call RAVE directly
- RAVE/QEMU only sees simulated instructions
    - We must communicate through them…
    - … but without changing the program behavior → Use instructions writing to x0 (which is hardcoded to 0)

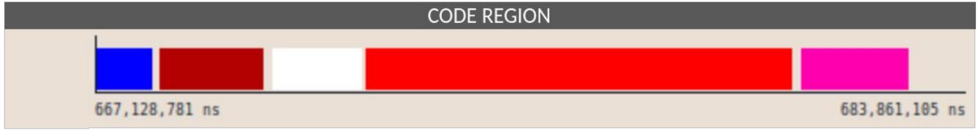
User Function	Instruction	Description
rave_start_trace()	li x0, -3	After this instruction, tracing is enabled
rave_stop_trace()	li x0, -4	After this instruction, tracing is disabled
rave_restart_trace()	li x0, -2	Deletes tracing information up to this point

Controlling the trace with the RAVE API

- We also add instrumentation mechanisms, to define regions of interest.
- We work with tuples of Events and Values:

Event: Code Region

Values: *"Ini"*, *"Compute"*, …



```
#define SEWS 4
int main(){
  rave_name_event(1000,"Code Region")
  rave_name_value(1000, 1, "Ini")
  rave_name_value(1000, 2, "Compute")

  double array1[256], array2[256], array3[256];

  rave_event_and_value(1000, 1)
  ini_vectors(array1, array2, array3);
  rave_event_and_value(1000, 0)

  rave_event_and_value(1000, 2)
  for(int i=0; i<256; ++i)
     array3[i] += array1[i] + array2[i];
  rave_event_and_value(1000,0)
};
```

Define event 1000 = "Code Region"
Value 1 = "Ini"
Value 2 = "Compute"

Enclose first region with value 1 ("Ini")

Enclose second region with value 2 ("Compute")

Controlling the trace with the RAVE API

These functionalities are also encoded in instructions writing to x0:

User Function	Instruction	Description
rave_event_and_value(e,v)	or x0, src1, src2	event **e** and value **v** are read from src1 and src2.
rave_name_event(e,name)	and x0, src1, src2	src1 contains the event **e**, and src2 is equal to -1
	li x0, -1 lui x0, name[0], lui x0, name[1], … li x0, -1	The "**li x0, -1**" instructions mark the beginning and end of the name, and then a series of **lui** instructions encode its characters one by one
rave_name_value(e,v,name)	and x0, src1, src2	src1 contains the event **e**, and src2 contains **v**
	li x0, -1 lui x0, name[0], lui x0, name[1], … li x0, -1	The value name is transmitted using the same protocol as the event name.

RAVE plugin flow (translation)

```
For each Instr ∈ translation_block
  disassembly ← qemu_plugin_insn_disas(Instr)
  switch( type(disassembly) )
    scalar :
      Instr_data ← fill_scalar_struct(Instr)
      Instr → set_callback(vcpu_insn_exec, Instr_data)
    vector :
      Instr_data ← fill_vec_struct(Instr)
      Instr → set_callback(vcpu_insn_exec, Instr_data)
    API/tracing :
      Function ← tracing_function(Instr)
      Instr → set_callback(Function)
```

```c
enum instr_type{SCALAR, VECTOR, VSETVL};
enum v_major_type{OTHER, ARITH, MEMORY, MASK};
enum v_minor_type{FP, INT, UNIT, STRIDE, INDEX};
struct instr_data{
  uint64_t PC;
  uint32_t paraver_code;
  char * asm_string;
  short dst, src1, src2, src3
  enum instr_type type;
  enum v_major_type v_majortype;
  enum v_major_type v_minortype;
};
```

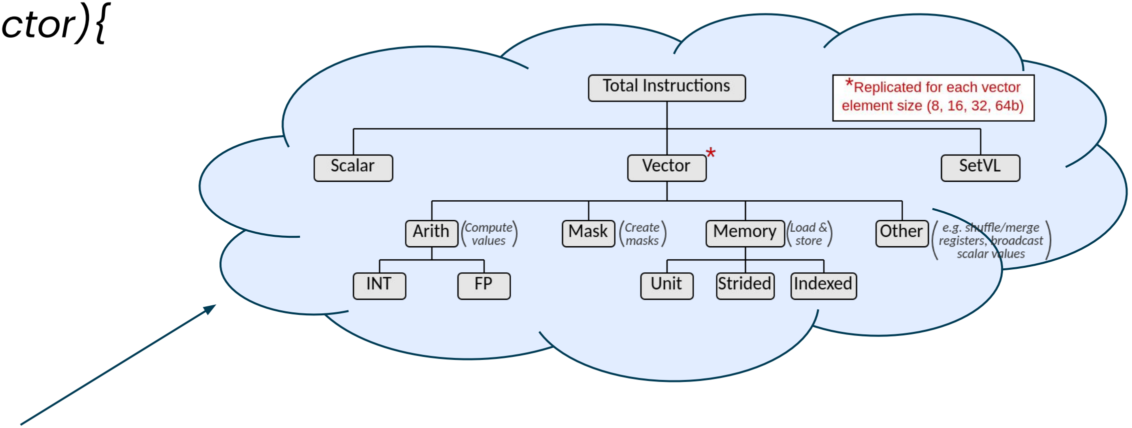
Different callbacks for **li x0, -1**, **and x0, src1, src2,** etc…

RAVE plugin flow (execution callback)

```
callback vcpu_insn_exec(Instr_data):
  //Trace and log
  If (Instr_data.major_type == Vector){
     read_cpu_state(VL, SEW)
     log_instruction(Instr_data)
     trace_instruction(Instr_data)
 }
 //counters
   ++total_instructions
   If (Instr_data.major_type == Vector) ++vector_instructions
   If (Instr_data.minor_type == V.Memory) ++vector_mem_instructions
   //(....)
```

RAVE plugin flow (execution callback)

```
callback vcpu_insn_exec(Instr_data):
  //Trace and log
  If (Instr_data.major_type == Vector){
     read_cpu_state(VL, SEW)
     log_instruction(Instr_data)
     trace_instruction(Instr_data)
  }
  //counters
   ++total_instructions
   If (Instr_data.major_type == Vector) ++vector_instructions
   If (Instr_data.minor_type == V.Memory) ++vector_mem_instructions
   //(....)
```



Total Instructions

*Replicated for each vector element size (8, 16, 32, 64b)

Scalar

Vector*

SetVL

Arith (Compute values)

Mask (Create masks)

Memory (Load & store)

Other (e.g. shuffle/merge registers, broadcast scalar values)

INT

FP

Unit

Strided

Indexed

Evaluating RAVE

We evaluate four setups/environments:

Commodity Laptop (QEMU+RAVE)

- 8-core Intel i7-8650U
- 2.1 GHz
- 16 GB

High-end AMD node (QEMU+RAVE)

- 12-core AMD Ryzen 5600G
- 3.9 GHz
- 32 GB

Unmatched RISC-V board (Vehave)

- 4-core HiFive Unmatched
- 1.2 GHz
- 16 GB

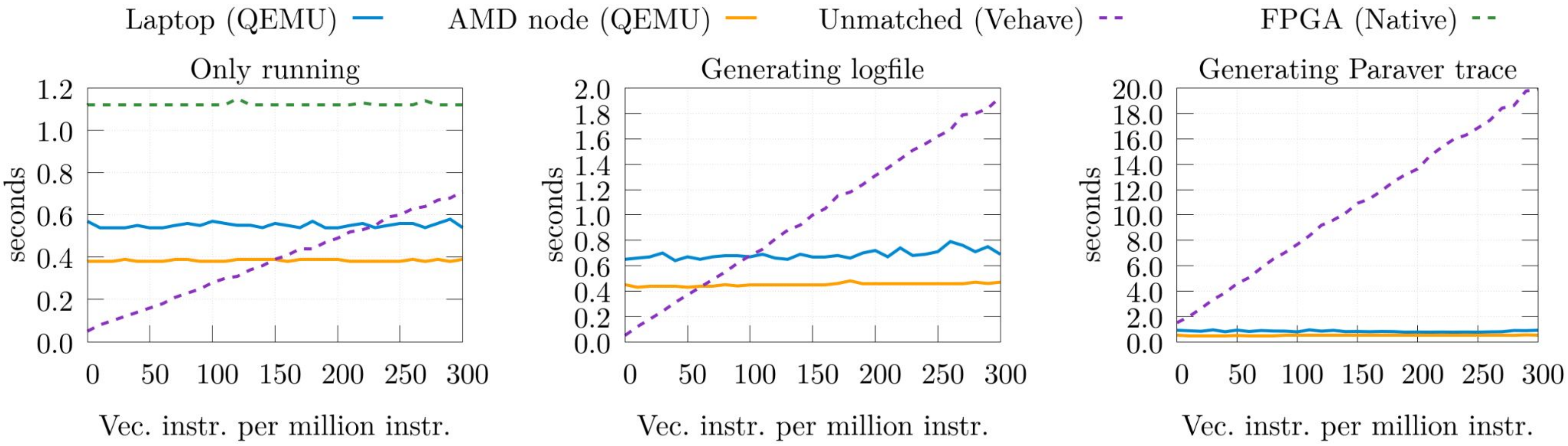
Native execution (FPGA)

- EPAC RTL on VCU128 FPGA
- 50 MHz
- 4 GB

Evaluating RAVE: Synthetic Benchmark

We measure the simulation time of a synthetic benchmark

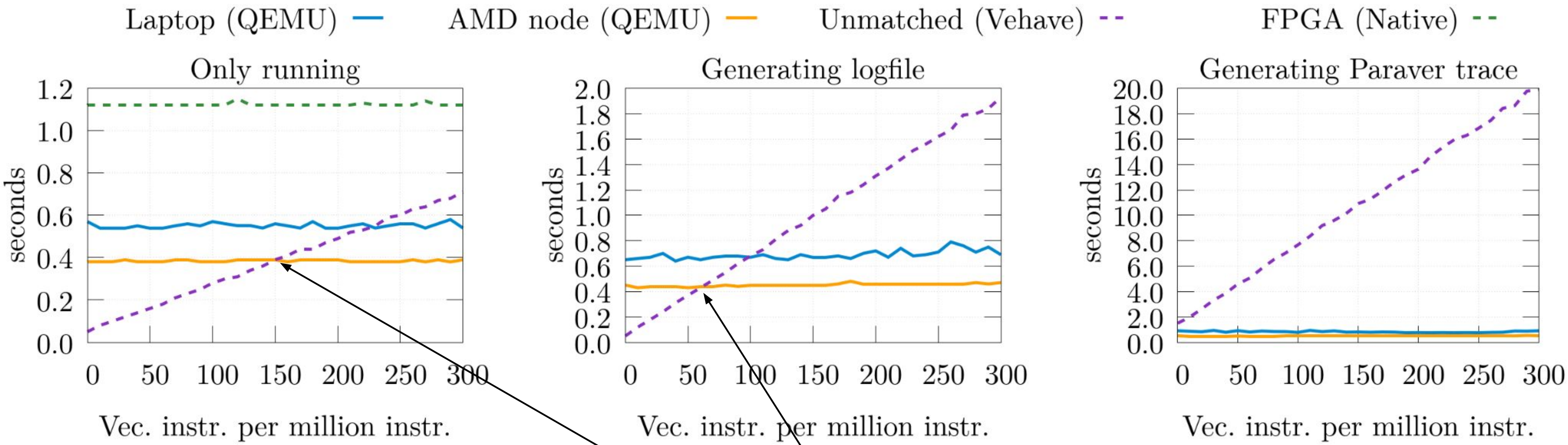
- Increasing ratio of Vec. Instructions per Million *total instructions* (Scalar+Vec)



Evaluating RAVE: Synthetic Benchmark

We measure the simulation time of a synthetic benchmark

- Increasing ratio of Vec. Instructions per Million *total instructions* (Scalar+Vec)



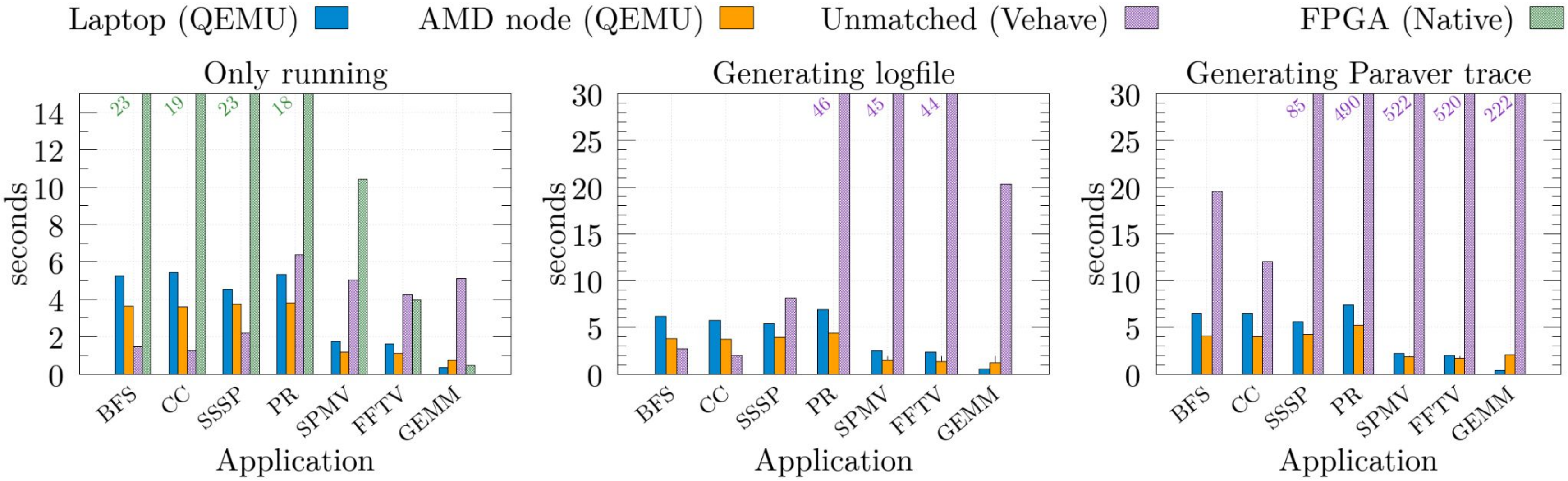
Vehave's performance degrades with more Vec. Instr (more O.S. overhead)

QEMU@AMD outperforms **Vehave** beyond 0.015% Vec. Instr (or 0.005% when logging)

Orders of magnitude speedup generating a PRV trace.

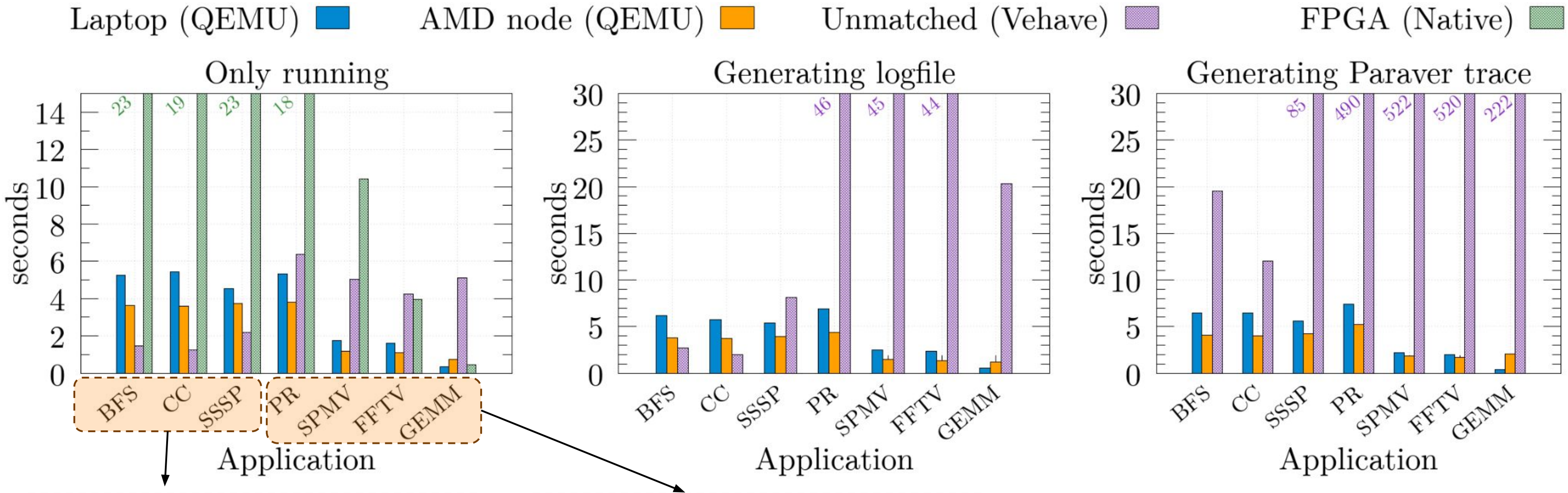
Evaluating RAVE: Real Applications

We also measure the simulation time of a real HPC kernels and applications



Evaluating RAVE: Real Applications

We also measure the simulation time of a real HPC kernels and applications



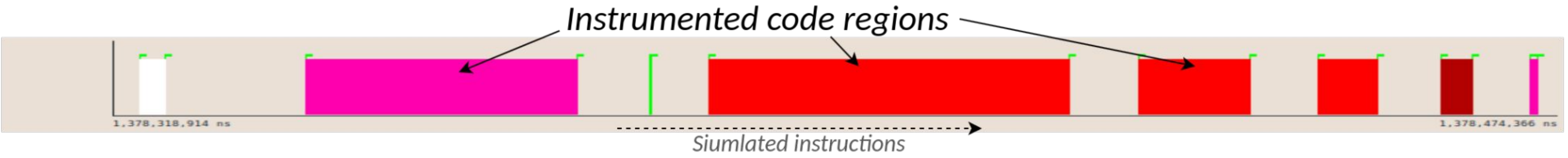
Graph applications employ many scalar instructions to read the graph from disk: **Vehave** is faster than **QEMU**

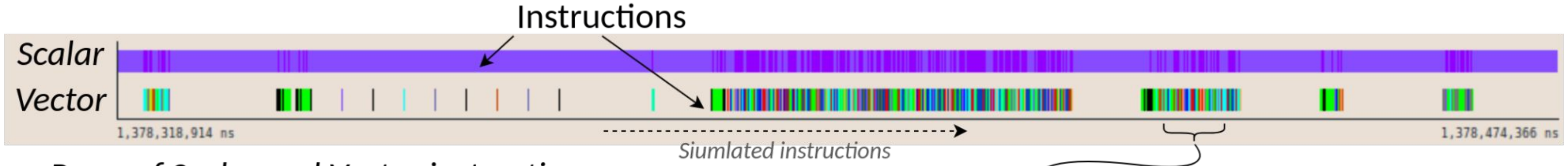
Compute-Intensive codes have more vector instructions, thus **QEMU** is faster than **Vehave**.

In all cases, QEMU generates PRV traces much faster..

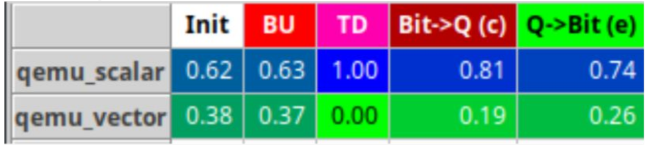
Evaluating RAVE: Use case

Beside validating the vectorized binary, we can generate Vectorization Traces:

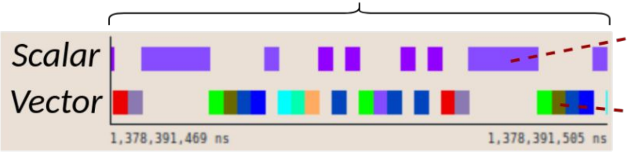




Prop. of Scalar and Vector instructions



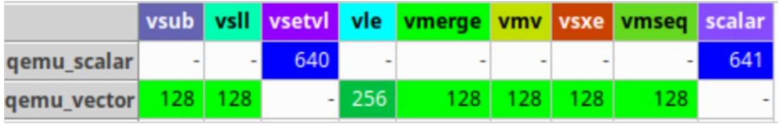
	Init	BU	TD	Bit->Q (c)	Q->Bit (e)
qemu_scalar	0.62	0.63	1.00	0.81	0.74
qemu_vector	0.38	0.37	0.00	0.19	0.26



Burst of scalar instructions

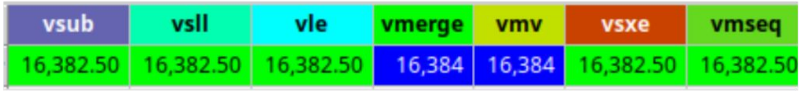
Individual vector instructions

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	-

Average bits per instruction (In phase Init)

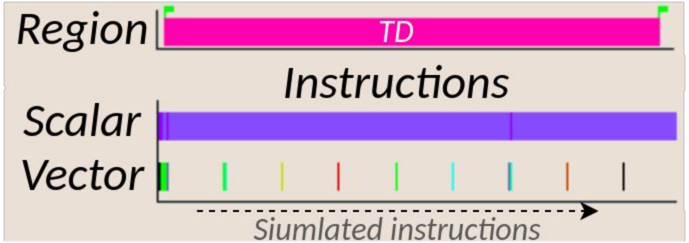


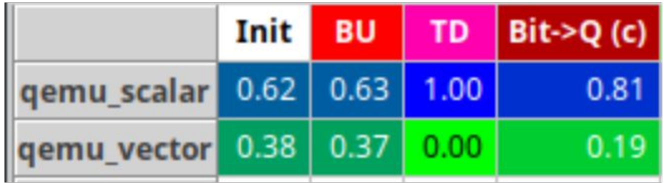
vsub	vsll	vle	vmerge	vmv	vsxe	vmseq
16,382.50	16,382.50	16,382.50	16,384	16,384	16,382.50	16,382.50

Evaluating RAVE: Use case

Use trace insight to improve vectorization:

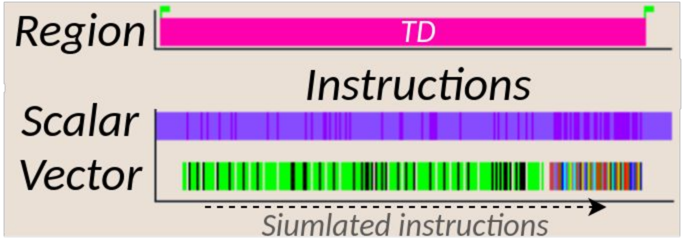
Before increasing TD Vectorization

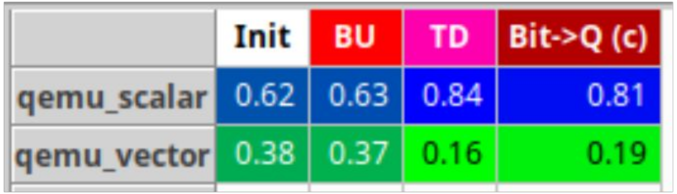




	Init	BU	TD	Bit->Q (c)
qemu_scalar	0.62	0.63	1.00	0.81
qemu_vector	0.38	0.37	0.00	0.19

After increasing TD Vectorization

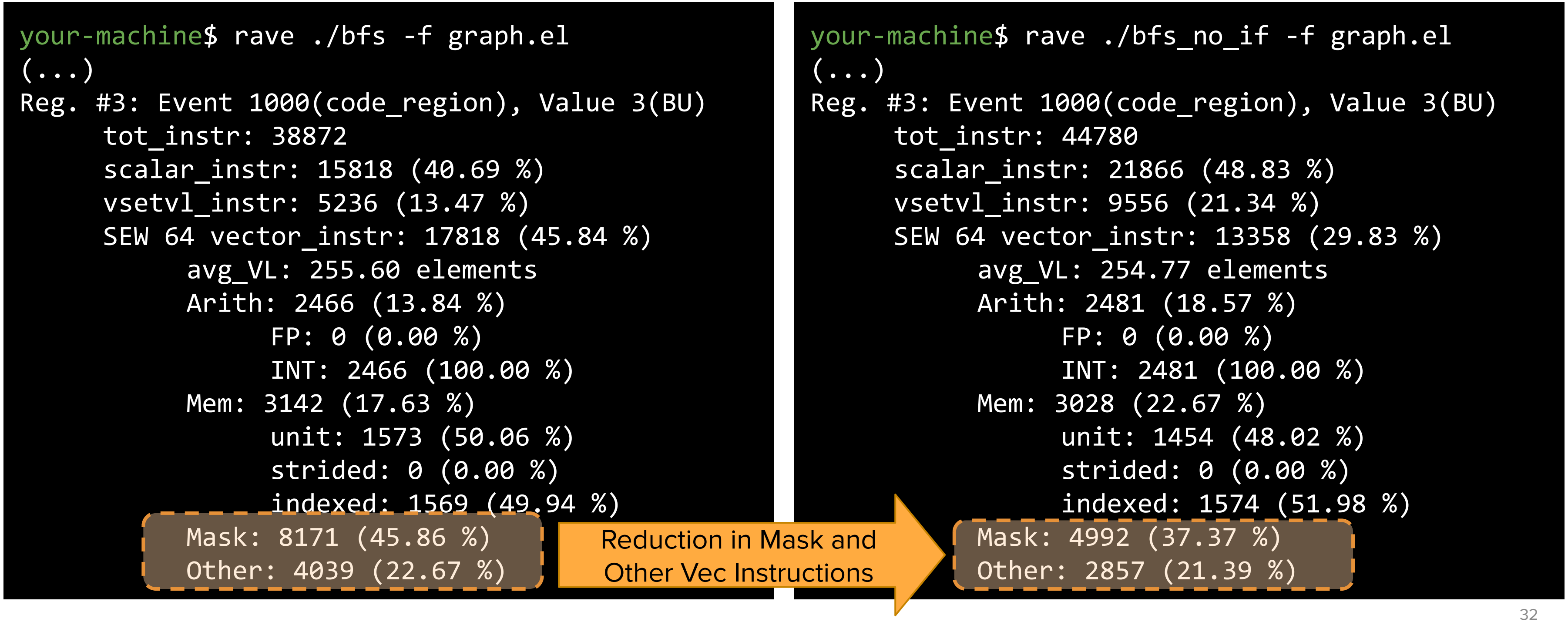




	Init	BU	TD	Bit->Q (c)
qemu_scalar	0.62	0.63	0.84	0.81
qemu_vector	0.38	0.37	0.16	0.19

Evaluating RAVE: Use case

You can also obtain vectorization metrics on a console report:



```
your-machine$ rave ./bfs -f graph.el
(...)
Reg. #3: Event 1000(code_region), Value 3(BU)
      tot_instr: 38872
      scalar_instr: 15818 (40.69 %)
      vsetvl_instr: 5236 (13.47 %)
      SEW 64 vector_instr: 17818 (45.84 %)
            avg_VL: 255.60 elements
            Arith: 2466 (13.84 %)
                  FP: 0 (0.00 %)
                  INT: 2466 (100.00 %)
            Mem: 3142 (17.63 %)
                  unit: 1573 (50.06 %)
                  strided: 0 (0.00 %)
                  indexed: 1569 (49.94 %)
            Mask: 8171 (45.86 %)
            Other: 4039 (22.67 %)
```

Reduction in Mask and Other Vec Instructions

```
your-machine$ rave ./bfs_no_if -f graph.el
(...)
Reg. #3: Event 1000(code_region), Value 3(BU)
      tot_instr: 44780
      scalar_instr: 21866 (48.83 %)
      vsetvl_instr: 9556 (21.34 %)
      SEW 64 vector_instr: 13358 (29.83 %)
            avg_VL: 254.77 elements
            Arith: 2481 (18.57 %)
                  FP: 0 (0.00 %)
                  INT: 2481 (100.00 %)
            Mem: 3028 (22.67 %)
                  unit: 1454 (48.02 %)
                  strided: 0 (0.00 %)
                  indexed: 1574 (51.98 %)
            Mask: 4992 (37.37 %)
            Other: 2857 (21.39 %)
```

Conclusions

- We developed a plugin for QEMU targeting the RISC-V Vector Extension

- We improved simulation framework of the EPI project:
  - More accessible
  - Faster
  - Increased functionality

- **RAVE** is already being used by performance analysts at BSC

- Future work include:
  - Selectively increasing the block size
  - Further speeding up the plugin
  - More functionalities (multi-core emulation, automatic instrumentation, …)

SDV Vector Analysis Using RAVE

Pablo Vizcaino
RISC-V Technical Session, October 10th

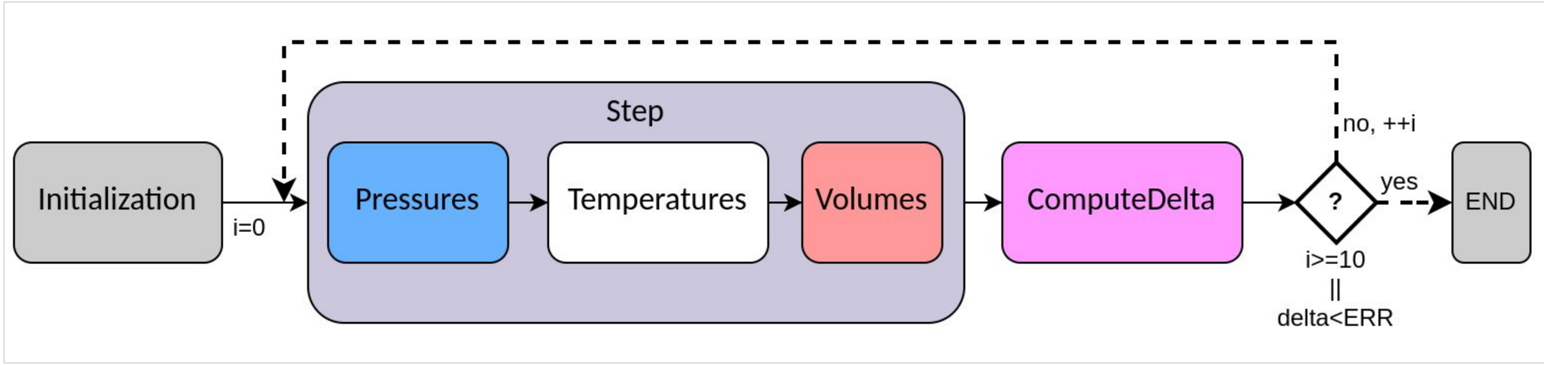
epi European Processor Initiative

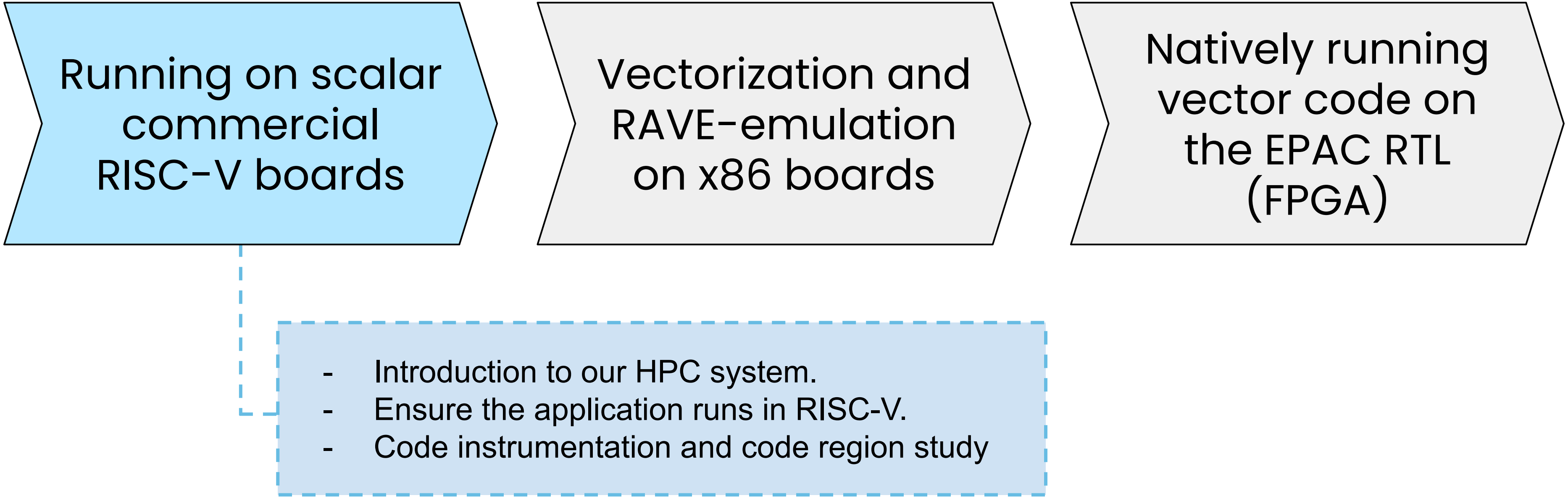
BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

The Tutorial code

- ***Main*** *function* : Initialization of 2D arrays and a loop of 10 timesteps
- Each timestep calls:
  - ***Step*** *function* : Works on three arrays, “*Pressures*”, “*Temperatures*”, and “*Volumes*”
  - ***ComputeDelta*** *function* : Computes convergence of result

- This application is not physically-meaningful but contains common operations:
  - Stencils, element-wise matrix operations, reductions, …



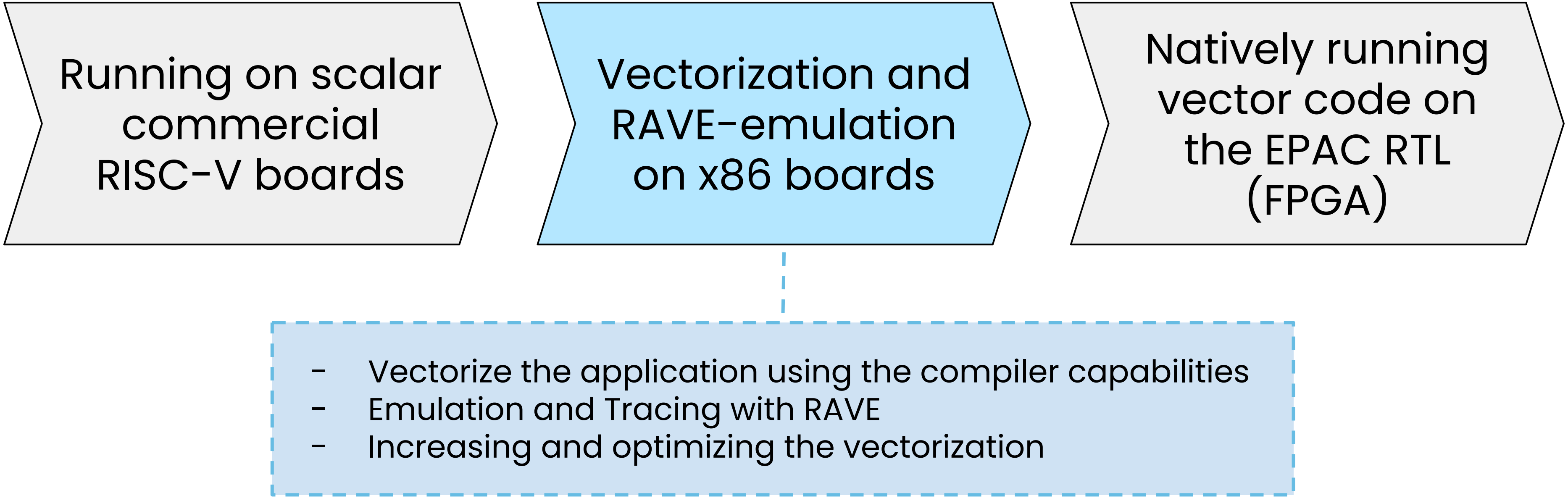


Running on scalar commercial RISC-V boards

Vectorization and RAVE-emulation on x86 boards

Natively running vector code on the EPAC RTL (FPGA)

- Introduction to our HPC system.
- Ensure the application runs in RISC-V.
- Code instrumentation and code region study



Running on scalar commercial RISC-V boards

Vectorization and RAVE-emulation on x86 boards

Natively running vector code on the EPAC RTL (FPGA)

- Vectorize the application using the compiler capabilities
- Emulation and Tracing with RAVE
- Increasing and optimizing the vectorization

2.2 Running and tracing with RAVE

- You can emulate the vectorized binary with RAVE:

```
laptop$ make reference-vec.x
laptop$ rave ./reference-vec.x
```

- You can also instrument your code and generate reports of RAVE emulations:

```
laptop$ make reference-vec-instrument.x
laptop$ RAVE_PRINT_REPORT=1 rave ./reference-vec-instrument.x
```

```
Region#38: Event 1000(code_region),Val 2(Temperatures)
  Moved bytes (Total): 2054589
    Moved bytes (scalar): 6589 (0.32 %)
    Moved bytes (vector): 2048000 (99.68 %)
  tot_instr: 210148
    scalar_instr: 194148 (92.39 %)
    vsetvl_instr: 1600 (0.76 %)
    vector_instr: 14400 (6.85 %)
      SEW 64 vector_instr: 14400 (100.00 %)
      avg_VL: 32.00 elements
      Arith: 6400 (44.44 %)
        FP: 6400 (100.00 %)
      Mem: 8000 (55.56 %)
        unit: 8000 (100.00 %)
        strided: 0 (0.00 %)
        indexed: 0 (0.00 %)
      Mask: 0 (0.00 %)
      Other: 0 (0.00 %)
```

- Or generate Parave traces:

```
laptop$ RAVE_PRV_NAME=ref-vec rave
./reference-vec-instrument.x
```

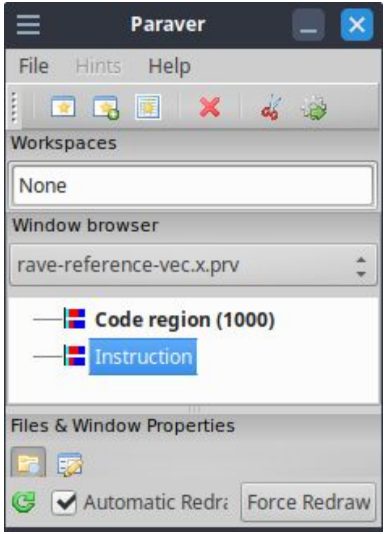
2.2 Running and tracing with RAVE

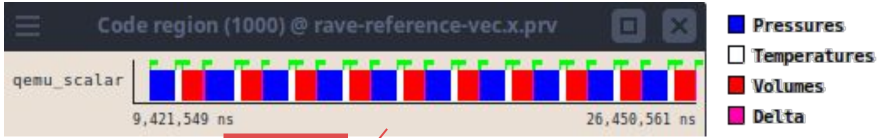
- Open the traces with Paraver:

```
laptop$ wxparaver ref-vec.prv
```

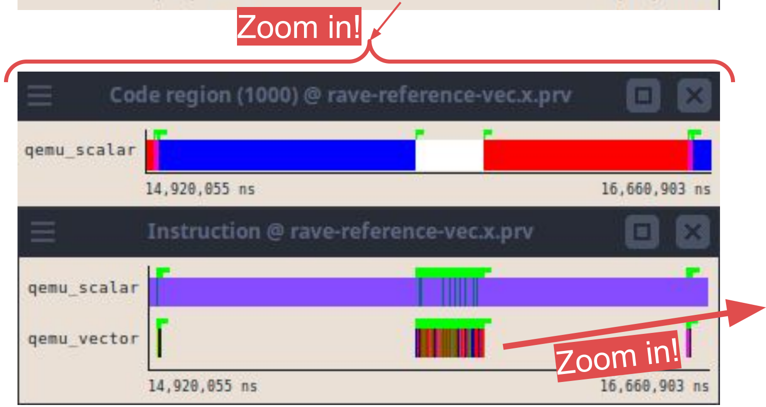
- Load the configuration files:

  paraver_cfgs/rave/per_phase_cfgs/event_1000_code_region.cfg
  paraver_cfgs/rave/Instruction_timeline.cfg

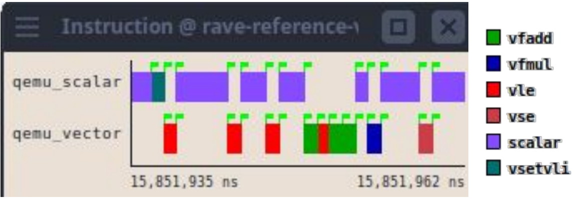




Horizontal axis is "*vec instructions*", not "*time*"

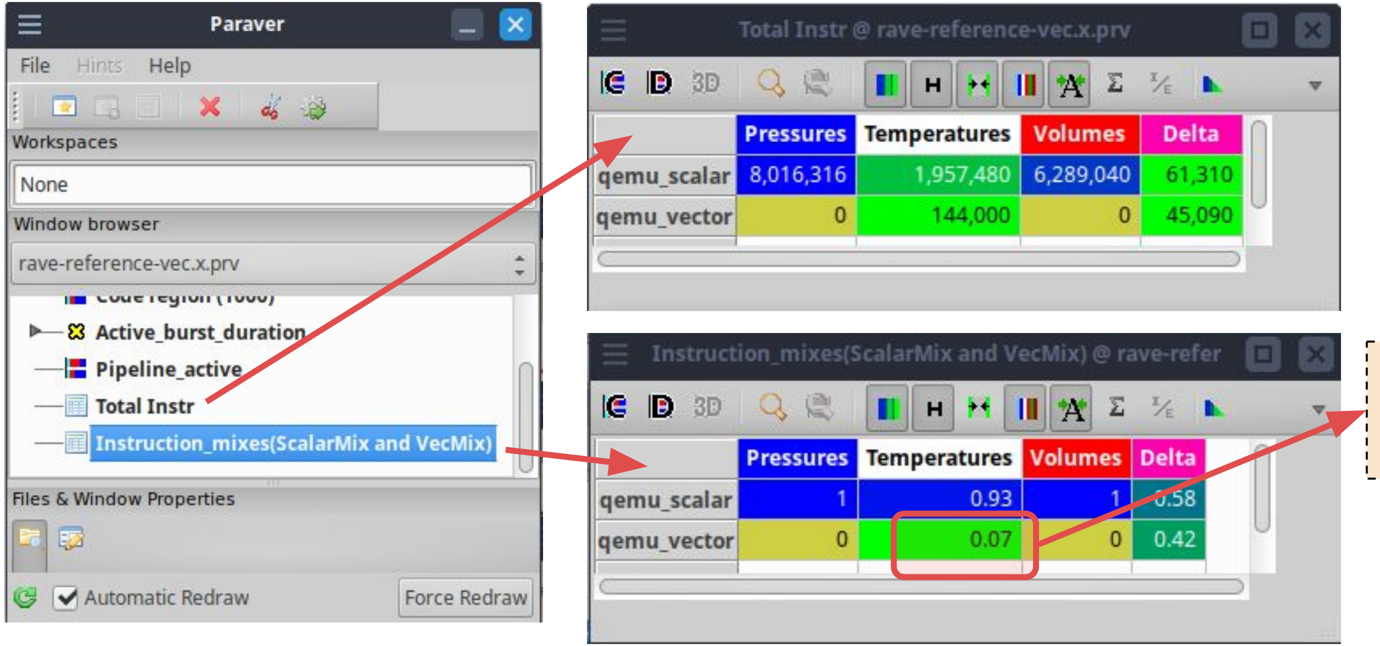


Phases Pressures and Volumes have no vector instructions



2.2 Running and tracing with RAVE

- Load the configuration file: *paraver_cfgs/rave/per_phase_cfgs/tables_vector_mix_per_phase.cfg*



Total Instr @ rave-reference-vec.x.prv

	Pressures	Temperatures	Volumes	Delta
qemu_scalar	8,016,316	1,957,480	6,289,040	61,310
qemu_vector	0	144,000	0	45,090

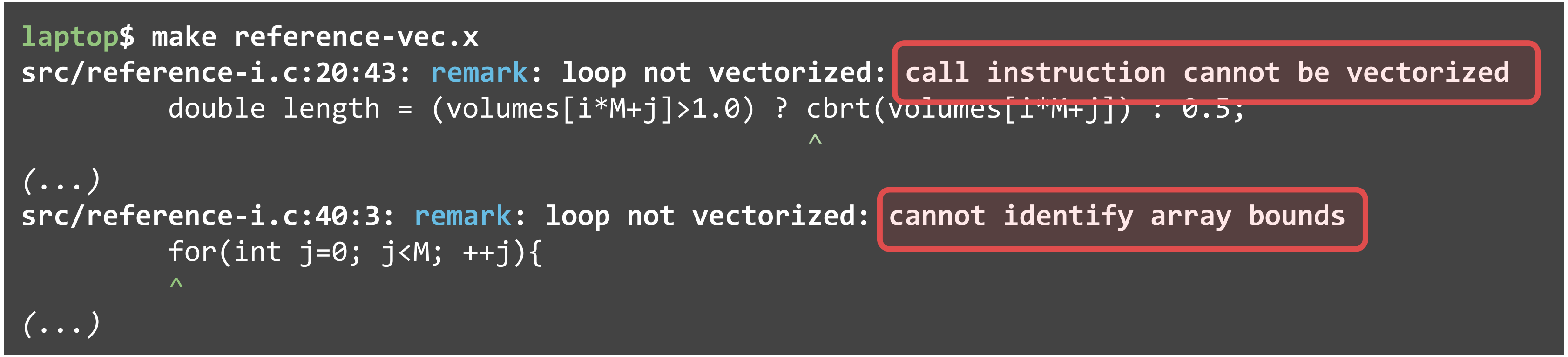
Instruction_mixes(ScalarMix and VecMix) @ rave-refer

	Pressures	Temperatures	Volumes	Delta
qemu_scalar	1	0.93	1	0.58
qemu_vector	0	0.07	0	0.42

Temperatures has a low vector mix (7%)

2.3 Increasing vectorization

- We can look at the compiler’s warnings to find out why some phases are not vectorized:



```
laptop$ make reference-vec.x
src/reference-i.c:20:43: remark: loop not vectorized: call instruction cannot be vectorized
         double length = (volumes[i*M+j]>1.0) ? cbrt(volumes[i*M+j]) : 0.5;
                                                ^
(...)
src/reference-i.c:40:3: remark: loop not vectorized: cannot identify array bounds
         for(int j=0; j<M; ++j){
         ^
(...)
```

- The Pressures phase cannot be vectorized due to the *cbrt()* function call.
- The Volumes phase has a problem with pointers and array bounds.
- For more information on compiler messages, refer to *this FAQ*

2.3 Increasing vectorization (Pressures)

European Processor Initiative

- We can separate vectorizatiable and non-vectorizable work into two loops:

```
trace_event_and_value(1000,1);
for(int i=0; i<N; ++i){
  for(int j=0; j<M; ++j){
    double length = (volumes[i*M+j]>1.0) ? cbrt(volumes[i*M+j]) : 0.5;
    pressures[i*M+j] = length + (temperatures[i*M+j]-new_temperatures[i*M+j]);
  }
}
```

```
trace_event_and_value(1000,1);
for(int i=0; i<N; ++i){
  for(int j=0; j<M; ++j){
    pressures[i*M+j] = (volumes[i*M+j]>1.0) ? cbrt(volumes[i*M+j]) : 0.5;
  }
}
trace_event_and_value(1000,5)
for(int i=0; i<N; ++i){
  for(int j=0; j<M; ++j){
    pressures[i*M+j] += (temperatures[i*M+j]-new_temperatures[i*M+j]);
  }
}
```

This loop will **not** vectorize

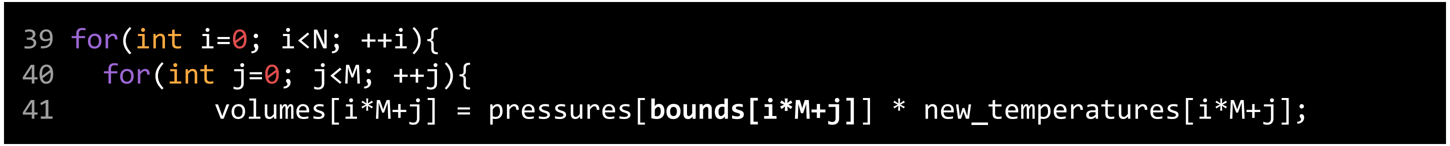
This loop will vectorize

Added a new region!

```
const char * v_names[]={"Other","Pressures_cbrt","Temperatures",
                        "Volumes","Delta","Pressures_vec"};
int values[] = {0,1,2,3,4,5};
trace_name_event_and_values(1000,"code_region",6,values,v_names);
```

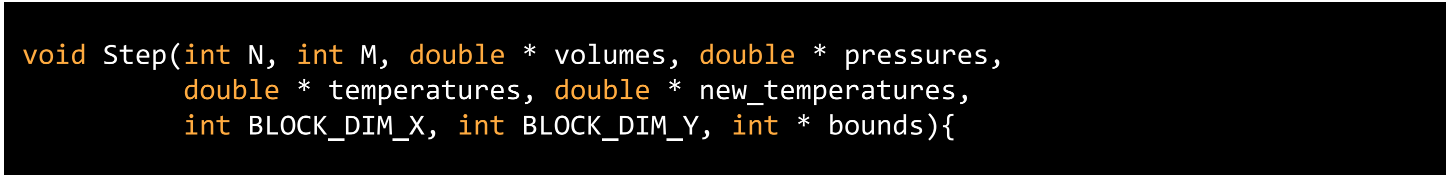
2.3 Increasing vectorization (Volumes)

- The compiler *“cannot identify array bounds”*.
  - This means the compiler cannot assert the aliasing of the arrays/pointers.
  - It normally occurs with indirected accesses

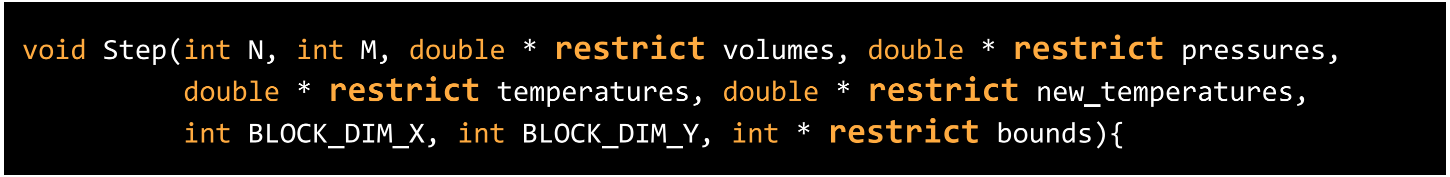


```
for(int i=0; i<N; ++i){
  for(int j=0; j<M; ++j){
        volumes[i*M+j] = pressures[bounds[i*M+j]] * new_temperatures[i*M+j];
```

- It can be solved using a *#pragma* or declaring your array pointers as *restrict*



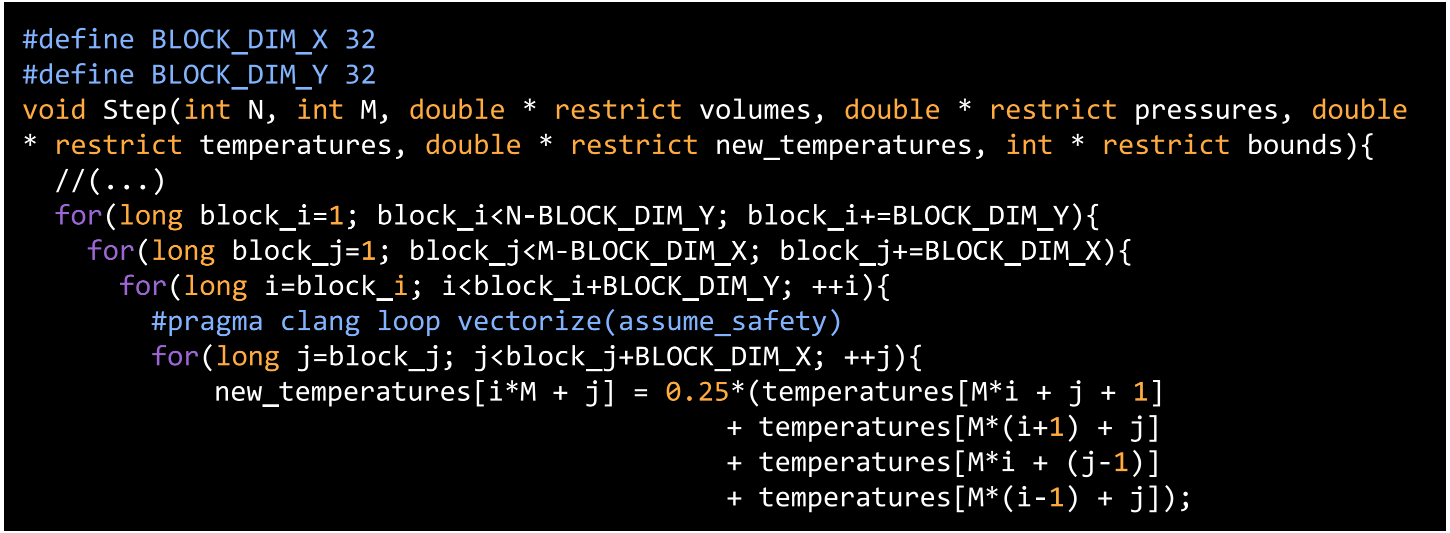
```
void Step(int N, int M, double * volumes, double * pressures,
          double * temperatures, double * new_temperatures,
          int BLOCK_DIM_X, int BLOCK_DIM_Y, int * bounds){
```



```
void Step(int N, int M, double * restrict volumes, double * restrict pressures,
          double * restrict temperatures, double * restrict new_temperatures,
          int BLOCK_DIM_X, int BLOCK_DIM_Y, int * restrict bounds){
```

2.3 Increasing vectorization (Temperatures)

- The compiler does complain, and the aren’t weird accesses or function calls
- We can make the loop more compiler-friendly with these three tricks:
  - Change the induction variables type from **int** to **long**
  - Add the **#pragma clang loop vectorize(assume_safety)** on top of the vectorizable loop (or make pointers **restrict**)
  - Move constant loop bounds known at compile time to **defines** (e.g. Block sizes)



```
#define BLOCK_DIM_X 32
#define BLOCK_DIM_Y 32
void Step(int N, int M, double * restrict volumes, double * restrict pressures, double
* restrict temperatures, double * restrict new_temperatures, int * restrict bounds){
  //(...)
  for(long block_i=1; block_i<N-BLOCK_DIM_Y; block_i+=BLOCK_DIM_Y){
    for(long block_j=1; block_j<M-BLOCK_DIM_X; block_j+=BLOCK_DIM_X){
      for(long i=block_i; i<block_i+BLOCK_DIM_Y; ++i){
        #pragma clang loop vectorize(assume_safety)
        for(long j=block_j; j<block_j+BLOCK_DIM_X; ++j){
            new_temperatures[i*M + j] = 0.25*(temperatures[M*i + j + 1]
                                        + temperatures[M*(i+1) + j]
                                        + temperatures[M*i + (j-1)]
                                        + temperatures[M*(i-1) + j]);
```

2.3 Increasing vectorization

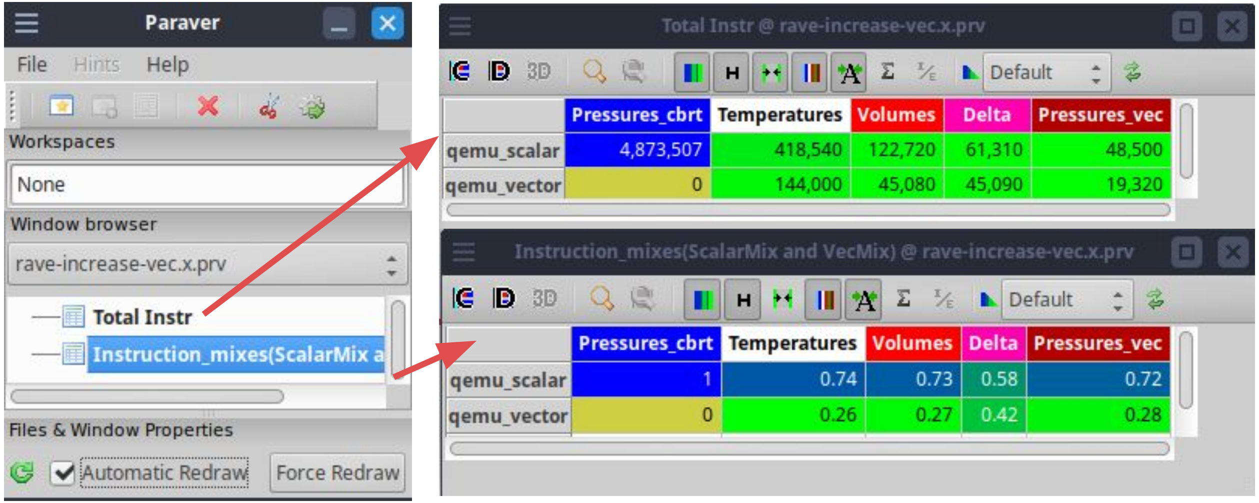
- Compile and trace the improved version *SDV_Tutorial/src/increase-vec.c*

```
laptop$ make increase-vec.x
laptop$ RAVE_PRINT_REPORT=1 RAVE_PRV_NAME=inc-vec rave ./increase-vec.x
```

- Open the traces with Paraver:

```
laptop$ wxparaver inc_vec.prv
```

- Load *paraver_cfgs/rave/per_phase_cfgs/tables_vector_mix_per_phase.cfg*



Total Instr @ rave-increase-vec.x.prv

	Pressures_cbrt	Temperatures	Volumes	Delta	Pressures_vec
qemu_scalar	4,873,507	418,540	122,720	61,310	48,500
qemu_vector	0	144,000	45,080	45,090	19,320

Instruction_mixes(ScalarMix and VecMix) @ rave-increase-vec.x.prv

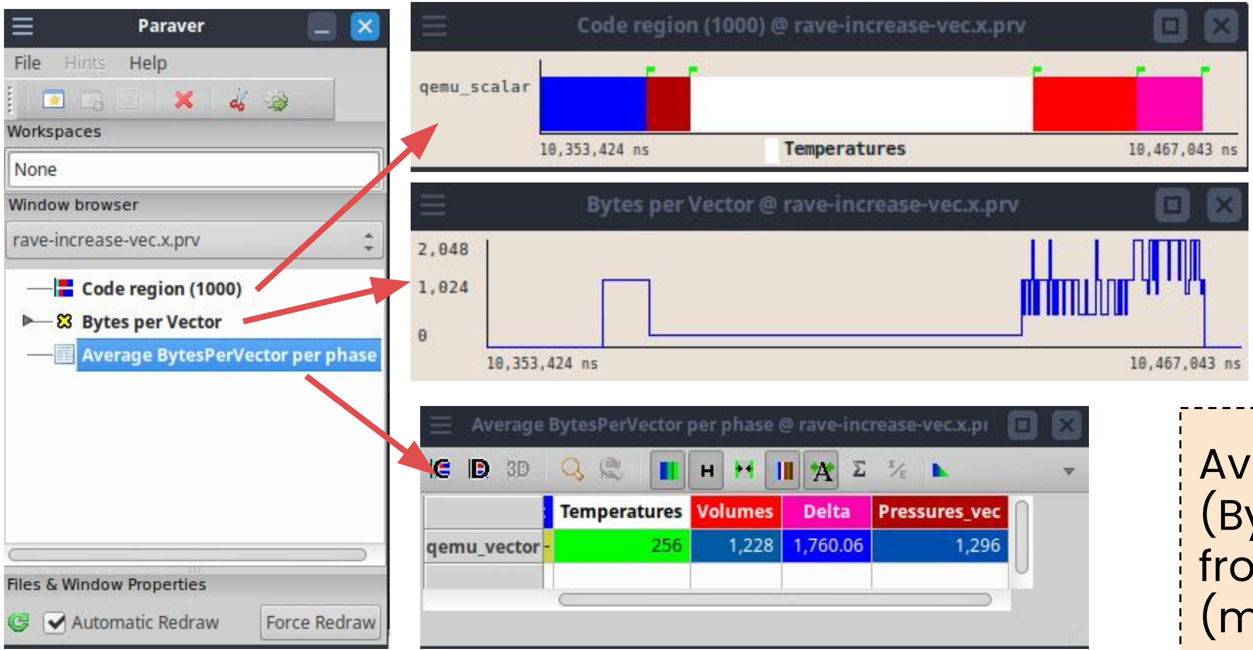
	Pressures_cbrt	Temperatures	Volumes	Delta	Pressures_vec
qemu_scalar	1	0.74	0.73	0.58	0.72
qemu_vector	0	0.26	0.27	0.42	0.28

Only the non-vectorizable Pressures_cbrt phase does not have vector instructions

Pressures_vec reports good vector metrics

2.4 Increasing the Vector Length

- Load *paraver_cfgs/rave/per_phase_cfgs/table_average_vl_per_phase.cfg*



	Temperatures	Volumes	Delta	Pressures_vec
qemu_vector	256	1,228	1,760.06	1,296

Vector Length (Bytes) per instruction varies a lot in Volumes and Delta.

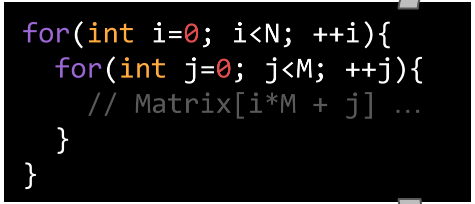
Average Vector Length (Bytes) in all phases far from **2048** Bytes/Vector (maximum in EPAC)

2.4 Increasing the Vector Length

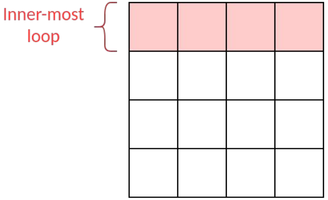
- The vector length is limited by the bounds of the inner-most loops.
- In this code, inner-most loops go from *j=0* to *j=M-1*, with M=162
- With double-precision data (64 bits), EPAC’s vectors support up to 256 elems.
  - The efficiency of the vector instructions grow with the vector length

- **Solution**: Increase inner-most loops bounds (collapsing loops)

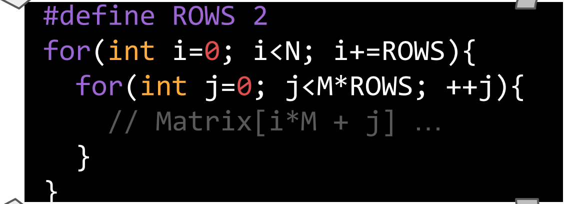
One row at a time



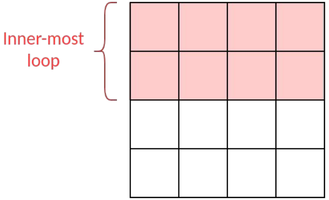
```
for(int i=0; i<N; ++i){
  for(int j=0; j<M; ++j){
    // Matrix[i*M + j] …
  }
}
```



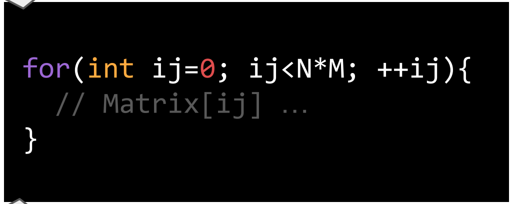
X (e.g., 2) rows at a time



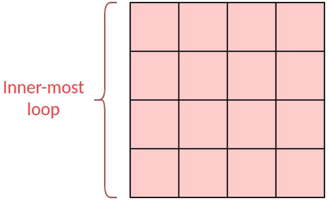
```
#define ROWS 2
for(int i=0; i<N; i+=ROWS){
  for(int j=0; j<M*ROWS; ++j){
    // Matrix[i*M + j] …
  }
}
```



All rows at a time (collapsing)



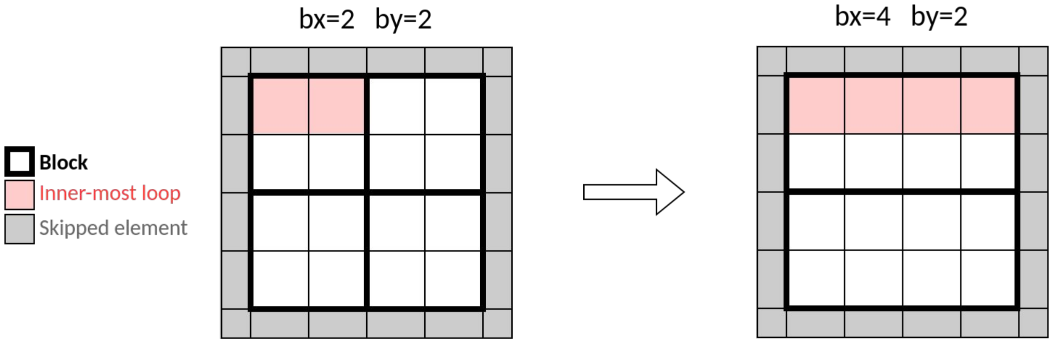
```
for(int ij=0; ij<N*M; ++ij){
  // Matrix[ij] …
}
```



2.4 Increasing the Vector Length

- We apply collapsing to phases Pressures_vec, Volumes, Delta.

- Phase Temperatures presents a blocking structure.
  - Normally intended to improve cache usage or vectorization on smaller extensions.
- Cannot collapse loops because edge elements should not be accessed in the stencil.
- We can increase the inner-most block size (bx) to match the columns' width to increase the vector length:



2.4 Increasing the Vector Length

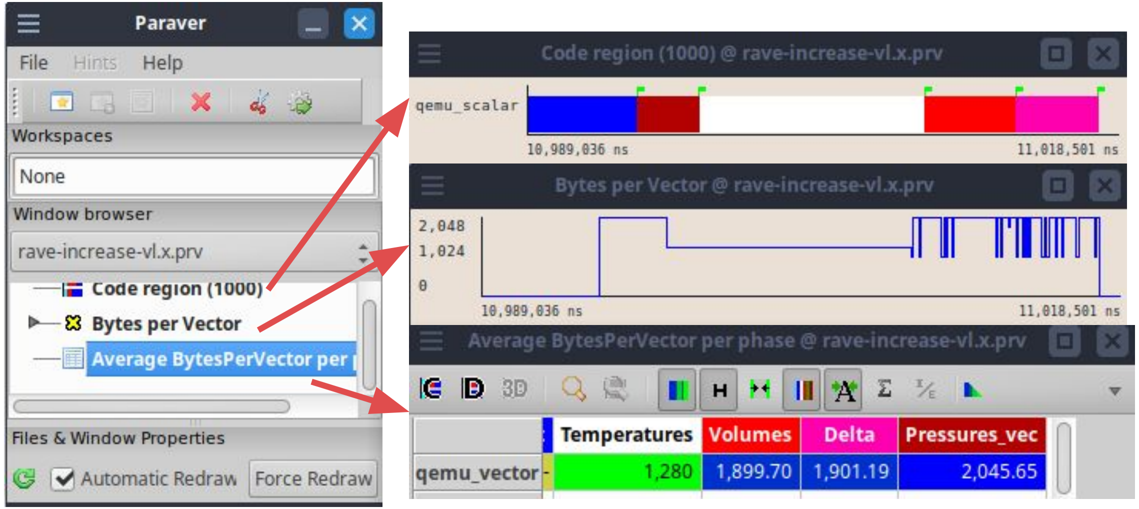
- Compile and trace the improved version *SDV_Tutorial/src/increase-vl.c*

```
laptop$ make increase-vl.x
laptop$ RAVE_PRINT_REPORT=1 RAVE_PRV_NAME=inc-vl rave ./increase-vl.x
```

- Open the traces with Paraver:

```
laptop$ wxparaver inc-vl.prv
```

- Load *paraver_cfgs/rave/per_phase_cfgs/table_average_vl_per_phase.cfg*



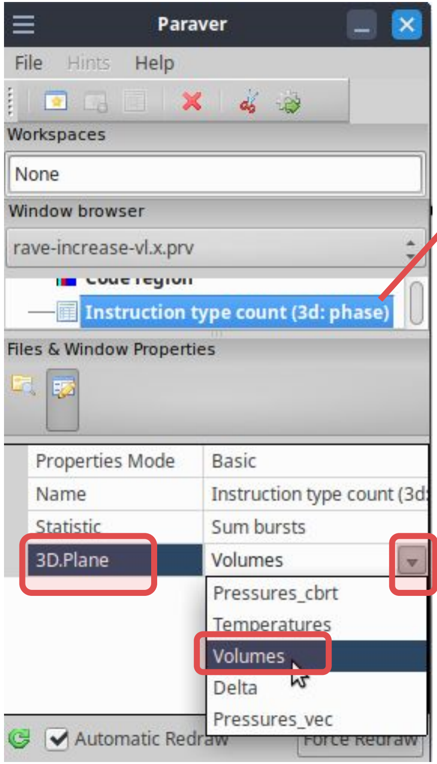
	Temperatures	Volumes	Delta	Pressures_vec
qemu_vector	1,280	1,899.70	1,901.19	2,045.65

Average Vector Length closer to **2048** Bytes/Vector (maximum in EPAC)

Temperatures increased from **256** to **1280**

2.5 Avoid mixing datatypes

- Load *paraver_cfgs/rave/per_phase_cfgs/table_instruction_type_count_per_phase.cfg*



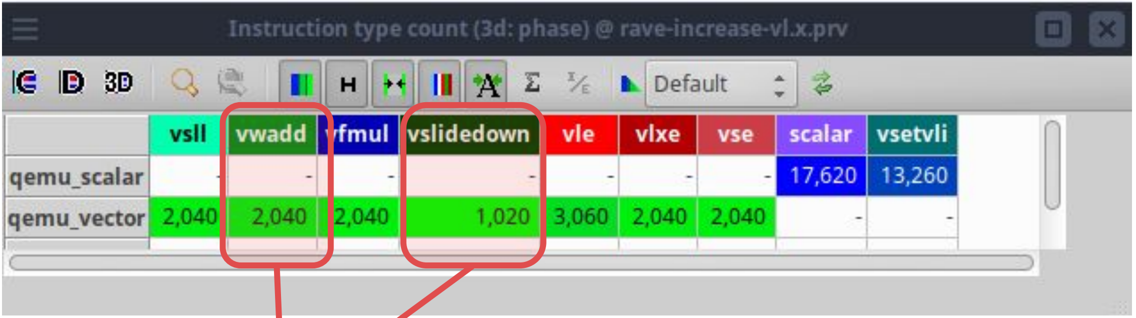
Workspaces: None

Window browser: rave-increase-vl.x.prv

Instruction type count (3d: phase)

Properties Mode	Basic
Name	Instruction type count (3d
Statistic	Sum bursts
3D.Plane	Volumes

- Pressures_cbrt
- Temperatures
- Volumes
- Delta
- Pressures_vec



Instruction type count (3d: phase) @ rave-increase-vl.x.prv

	vsll	vwadd	vfmul	vslidedown	vle	vlxe	vse	scalar	vsetvli
qemu_scalar	-	-	-	-	-	-	-	17,620	13,260
qemu_vector	2,040	2,040	2,040	1,020	3,060	2,040	2,040	-	-

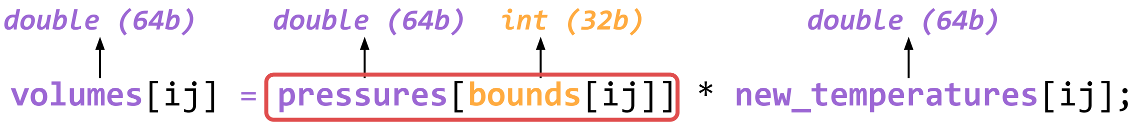
Two high-latency [1] instructions detected in Volumes

slide and ***vw**** instructions often appear when mixing datatypes.

[1] *Refer to the last slide (Annex) for latency estimates of vector instructions*

2.5 Avoid mixing datatypes

- Phase 3 uses an array of integers to index an array of doubles:



```
volumes[ij] = pressures[bounds[ij]] * new_temperatures[ij];
```

- We recommend parameterizing the datatypes, to experiment with different sizes:

```
typedef double T_FP; //64b
typedef long long T_INT; //64b
```

- Solution in *SDV_Tutorial/src/flex-datatype.c*. Compile it and trace it:

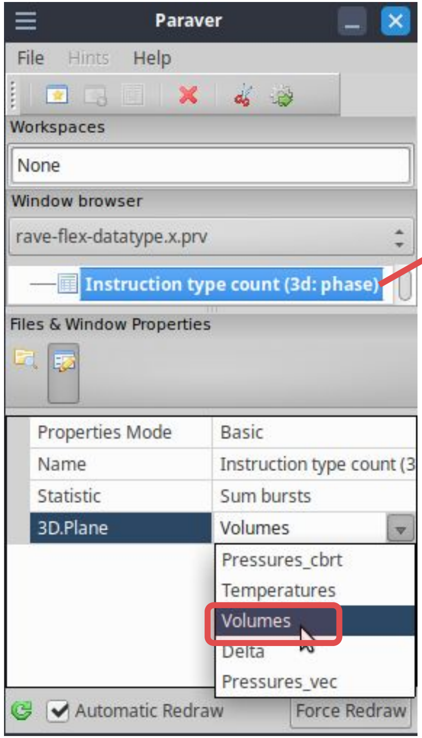
```
laptop$ make flex-datatype-i-vehave.x
laptop$ RAVE_PRINT_REPORT=1 RAVE_PRV_NAME=flex rave ./flex-datatype-i-vehave.x
```

- Copy the traces back to your computer and open them in Paraver:

```
laptop$ wxparaver flex.prv
```

2.5 Avoid mixing datatypes

- Load *paraver_cfgs/rave/per_phase_cfgs/table_instruction_type_count_per_phase.cfg*



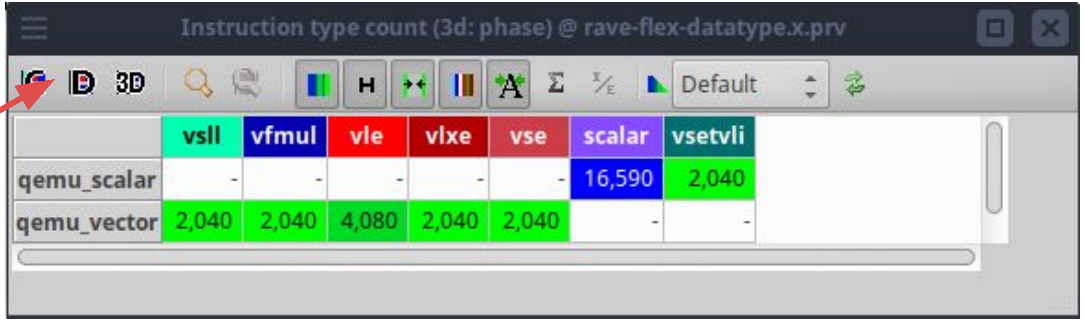
Workspaces: None

Window browser: rave-flex-datatype.x.prv

Instruction type count (3d: phase)

Properties Mode	Basic
Name	Instruction type count (3
Statistic	Sum bursts
3D.Plane	Volumes

Pressures_cbrt
Temperatures
Volumes
Delta
Pressures_vec

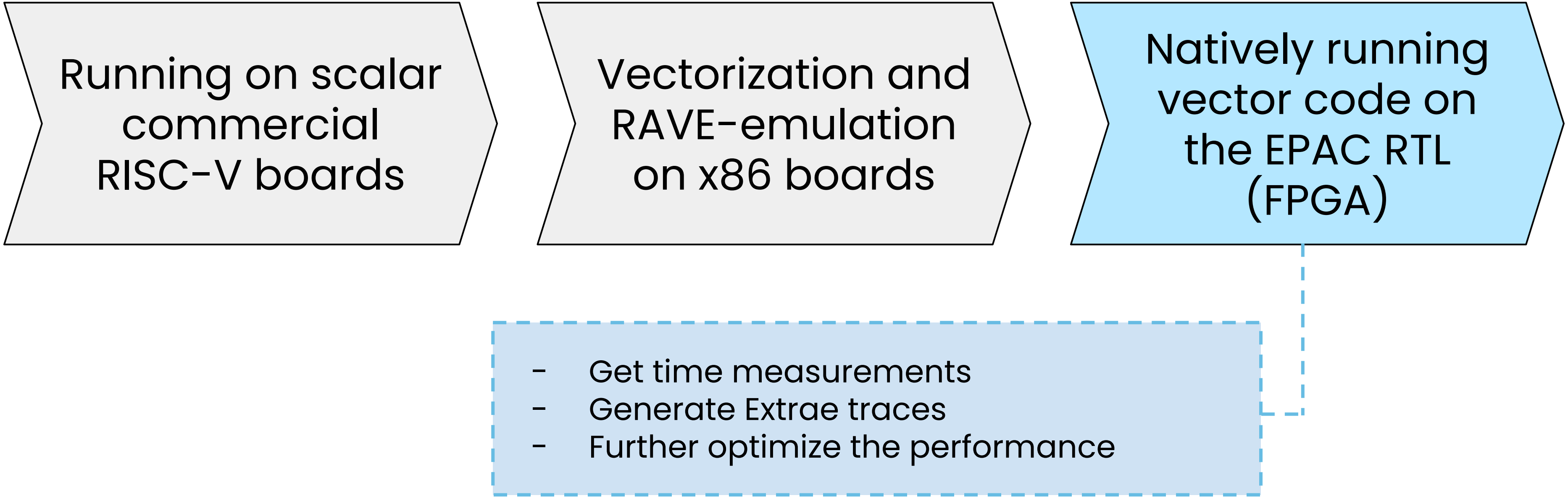


Instruction type count (3d: phase) @ rave-flex-datatype.x.prv

	vsll	vfmul	vle	vlxe	vse	scalar	vsetvli
qemu_scalar	-	-	-	-	-	16,590	2,040
qemu_vector	2,040	2,040	4,080	2,040	2,040	-	-

slide and ***vw**** instructions no longer present on Volumes

vsetvli instructions reduced from **13k** to **2k**



Running on scalar commercial RISC-V boards

Vectorization and RAVE-emulation on x86 boards

Natively running vector code on the EPAC RTL (FPGA)

- Get time measurements
- Generate Extrae traces
- Further optimize the performance

3.1 Sending jobs to the FPGA nodes

- You can send binaries to the FPGA using the SLURM queue manager and the *fpga_job* jobscript.
- Use the *run_all.sh* script to run all versions and parse their outputs:

```
synth-hca$ sbatch fpga_job ./run_all.sh
Submitted batch job 189294
```

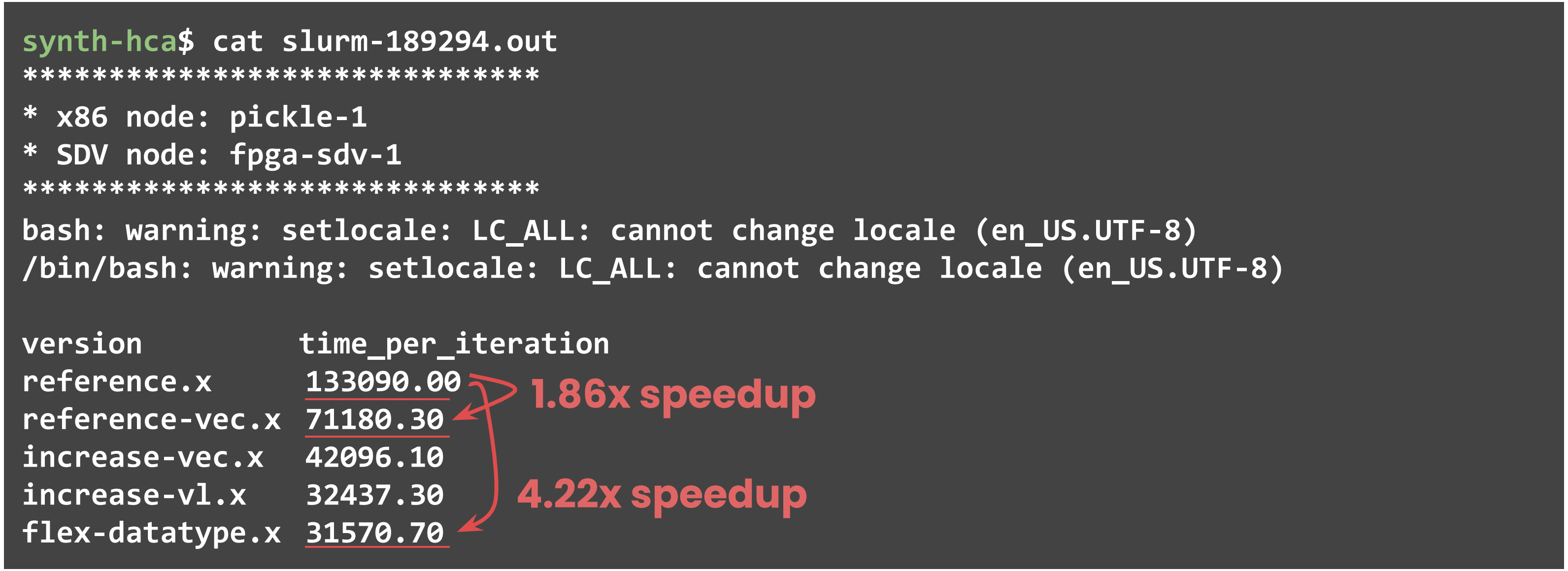
- You can query the state of an FPGA job using the **squeue** command:

```
synth-hca$ squeue
  JOBID   PARTITION    NAME       USER  ST  TIME  NODES  NODELIST(REASON)
  189294  fpga-sdv     fpga_job   user  R   0:21  1     pickle-1
```

Job is running

3.1 Sending jobs to the FPGA nodes

- When the job finishes, you can read its output file:



```
synth-hca$ cat slurm-189294.out
******************************
* x86 node: pickle-1
* SDV node: fpga-sdv-1
******************************
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
/bin/bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)

version         time_per_iteration
reference.x      133090.00
reference-vec.x  71180.30
increase-vec.x   42096.10
increase-vl.x    32437.30
flex-datatype.x  31570.70
```

1.86x speedup

4.22x speedup

3.2 Getting Extrae traces in the FPGA nodes

- We recommend using Extrae to trace the binaries running in the FPGA, but there are other methods (like PAPI), described in this guide.

- Send an Extrae job to the FPGA using the *trace_extrae_fpga script*:

```
synth-hca$ sbatch fpga_job trace_extrae_fpga ./flex-datatype.x
```

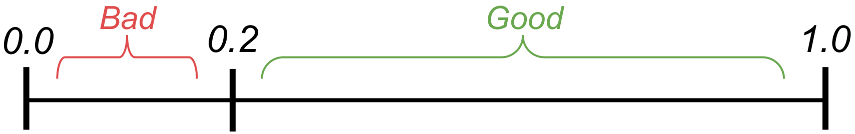
- When the job finishes, copy the traces back to your machine and open them in Paraver

```
your-machine$ rsync -a user@ssh.hca.bsc.es:~/SDV_Tutorial/extrae_prv_traces .
your-machine$ wxparaver ./extrae_prv_traces/fpga-flex-datatype.prv
```

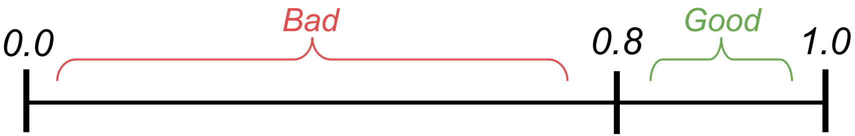
3.2 Getting Extrae traces in the FPGA nodes

- The configuration file *paraver_cfgs/extrae/PerfMetrics.cfg computes these vector metrics*:

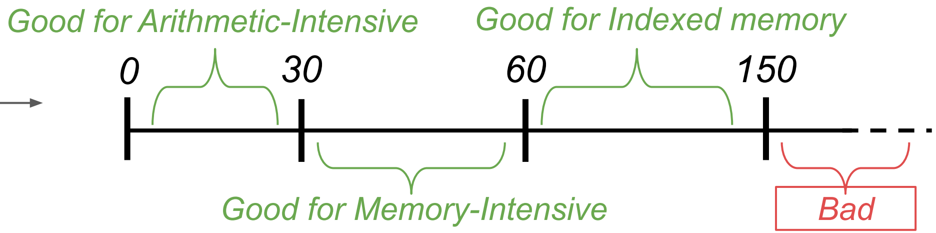
1. ***Vector Mix:*** $\frac{\#\text{Vec. Instr}}{\#\text{Tot. Instr}} \longrightarrow$



2. ***Vector Activity:*** $\frac{\#\text{Vec. Cyc}}{\#\text{Tot. Cyc}} \longrightarrow$

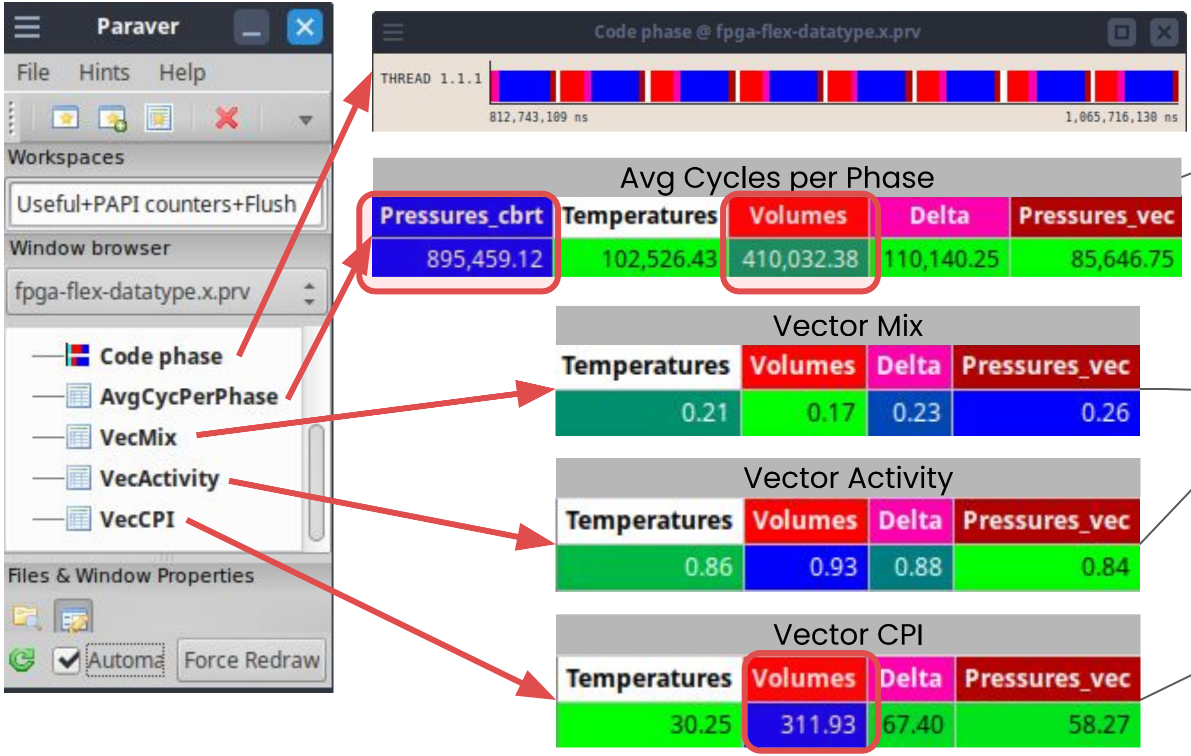


3. ***Vector CPI:*** $\frac{\#\text{Vec. Cyc}}{\#\text{Vec. Instr}} \longrightarrow$



3.2 Getting Extrae traces in the FPGA nodes

- Load the configuration file *paraver_cfgs/extrae/PerfMetrics.cfg*



Avg Cycles per Phase

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

Vector Mix

Temperatures	Volumes	Delta	Pressures_vec
0.21	0.17	0.23	0.26

Vector Activity

Temperatures	Volumes	Delta	Pressures_vec
0.86	0.93	0.88	0.84

Vector CPI

Temperatures	Volumes	Delta	Pressures_vec
30.25	311.93	67.40	58.27

Most of the time is spent in Pressures_cbrt and Volumes regions

All phases have a good **VecMix** and **VecActivity**

Volumes has a high **VecCPI**

3.3 Using huge pages

- When a region with indexed/indirect access (like Volumes) has a large **VecCPI** it might be a TLB issue:
  - The vector elements might be accessing more pages than there are entries in the TLB.

- The solution is to let the OS use **huge pages (2MB)** instead of 4KB pages.
  - You can use the script in *huge_pages* to execute your binary with huge pages.

- You can send an Extrae job using huge pages like this:

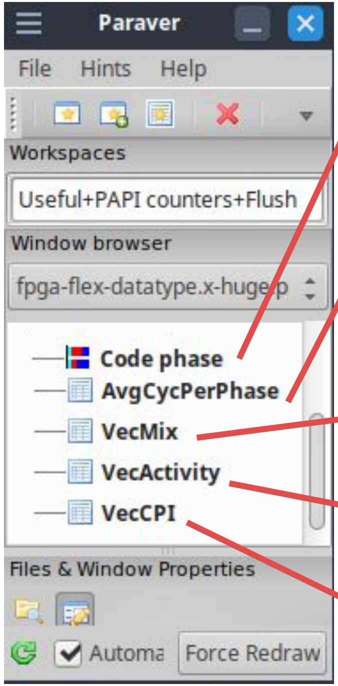
```
synth-hca$ sbatch fpga_job huge_pages run_extrae_fpga ./flex-datatype.x
```

- When the job finishes, copy the traces back to your machine and open them

```
your-machine$ rsync -a user@ssh.hca.bsc.es:~/SDV_Tutorial/extrae_prv_traces .
your-machine$ wxparaver ./extrae_prv_traces/flex-datatype.x-huge.prv
```

3.3 Using huge pages

- Load the configuration file *paraver_cfgs/extrae/PerfMetrics.cfg*



Workspaces

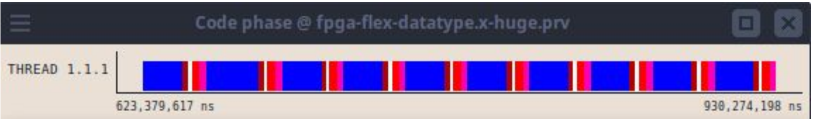
Useful+PAPI counters+Flush

Window browser

fpga-flex-datatype.x-huge.p

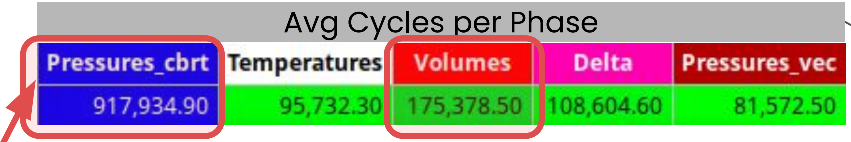
- Code phase
- AvgCycPerPhase
- VecMix
- VecActivity
- VecCPI

Files & Window Properties



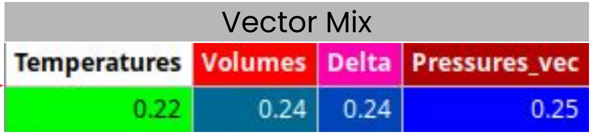
Code phase @ fpga-flex-datatype.x-huge.prv

```
THREAD 1.1.1
623,379,617 ns                                        930,274,198 ns
```



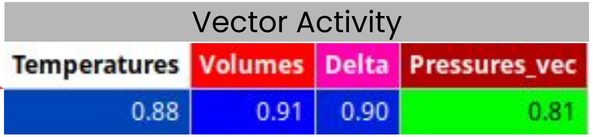
Avg Cycles per Phase

Pressures_cbrt	Temperatures	Volumes	Delta	Pressures_vec
917,934.90	95,732.30	175,378.50	108,604.60	81,572.50



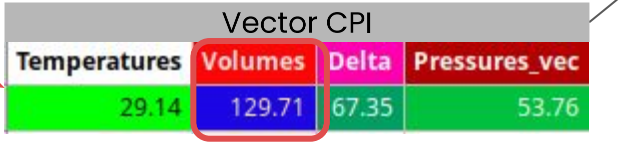
Vector Mix

Temperatures	Volumes	Delta	Pressures_vec
0.22	0.24	0.24	0.25



Vector Activity

Temperatures	Volumes	Delta	Pressures_vec
0.88	0.91	0.90	0.81



Vector CPI

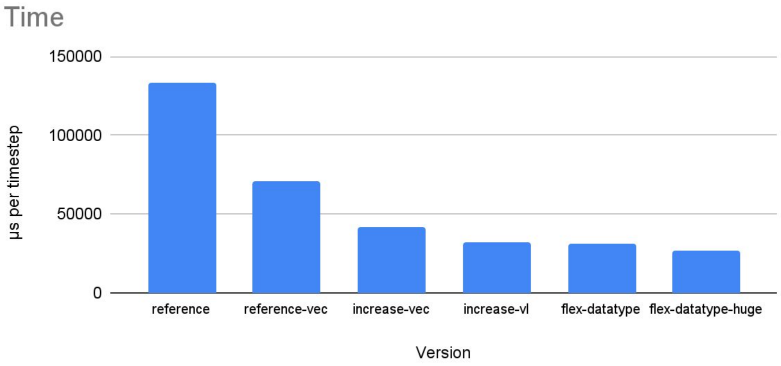
Temperatures	Volumes	Delta	Pressures_vec
29.14	129.71	67.35	53.76

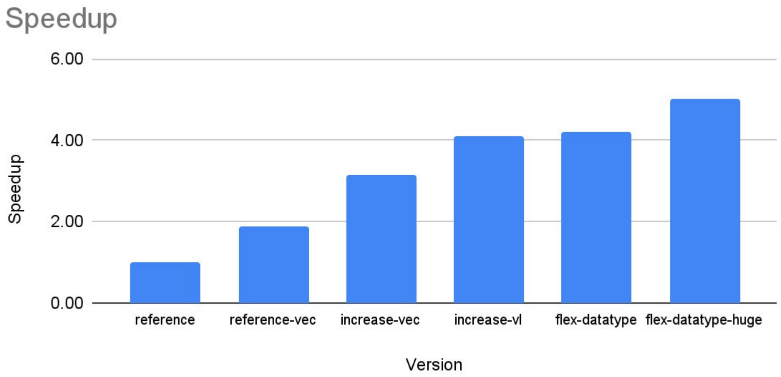
Volumes reduced its average duration from **410k** cycles to **175k**

Volumes reduced its **VecCPI** from **>300** to **130**

Conclusions

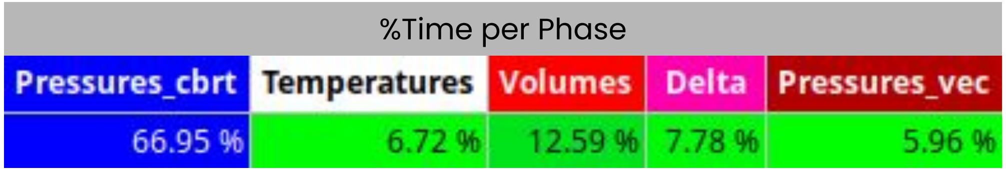
- Improvement across versions:





- Using the SDV methodology we achieved a 5x speedup on the application

- Most of the time is spent on non-vectorized code



%Time per Phase				
Pressures_cbrt	Temperatures	Volumes	Delta	Pressures_vec
66.95 %	6.72 %	12.59 %	7.78 %	5.96 %

Future improvements should focus on vectorizing Pressures_cbrt

Acknowledgment

This research has received funding from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 (European Processor Initiative) and Specific Grant Agreement No 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland. The EPI-SGA2 project, PCI2022-132935 is also co-funded by MCIN/AEI /10.13039/501100011033 and by the UE NextGenerationEU/PRTR.

Don't hesitate to contact me at pablo.vizcaino@bsc.es !