Analyzing a complex application

Using the SDVs

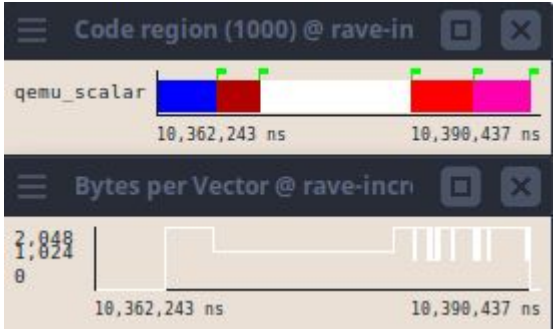
Introduction

- This tutorial exemplifies how to use the SDV environment to vectorize and analyze applications for the EPAC chip.
- We will:
  - Run an application on RISC-V commercial boards
  - Vectorize the application and emulate it with *RAVE*
  - Analyze vectorization traces and optimize the application using common techniques:
    - Increase vectorization, Increase vector-length, guiding the compiler, …
  - Execute it in an FPGA to get real timing measurements and execution traces

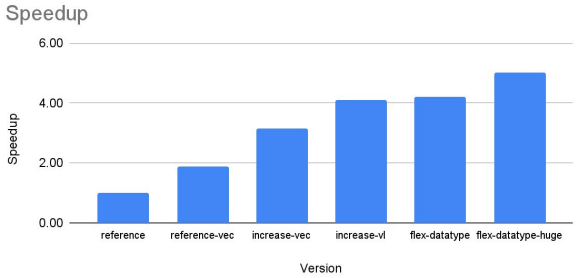
Vectorize

```
for(int i=0; i<N; ++i){
 #pragma clang loop vectorize(enable)
 for(int j=0; j<M; ++j){
   pressures[i*M+j] += (temp[i*M+j]
                     - new_temp[i*M+j]);
  }
}
```

Analyze



Optimize



Before we start…

Key concepts definitions:

- **Traces**: Files that contain information about the execution of a program.
- **Extrae**: BSC's instrumentation and trace generation library
- **Instrument**: Manually add directives in your code to add extra information in the traces
- **Paraver**: BSC's Trace visualization tool
- **RAVE**: BSC's tracer plugin for QEMU vector emulations

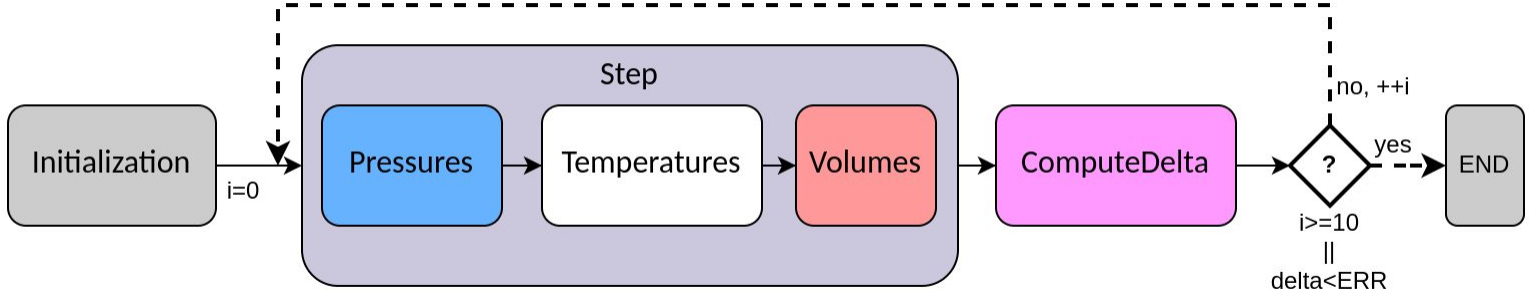
Machines required to follow this tutorial:

- **Laptop**: Where we will run Paraver to analyze traces (*https://tools.bsc.es/paraver*)
- **hca-server**: Login node to the SDV cluster
    - **Arriesgado**: RISC-V Unmatched boards for compilation and native execution
    - **synth-hca:** x86 board for cross-compilation and QEMU emulation
    - **fpga-sdv:** FPGA's implementing the EPAC hardware

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

1.1 Accessing the system

- Log in into HCA and copy the Tutorial sources:

```
your-machine$ ssh user@ssh.hca.bsc.es
hca-server$ cp /home/ictp-mhpc25/wednesday_05/SDV_Tutorial.tar.gz .
```

- Then, uncompress them:

```
hca-server$ tar -xzvf SDV_Tutorial.tar.gz
```

- From HCA, access the shared RISC-V Unmatched board node (Arriesgado-11)

```
hca-server$ ssh riscv
arriesgado-11$ cd SDV_Tutorial
```

1.2 Compiling and running the scalar code

- We support two RISC-V Vector (RVV) specifications:

Spec	Runs in emulation (VEHAVE / RAVE)	Runs in hardware (FPGA)	Compiles C/C++	Compiles Fortran
RVV0.7	✅	✅	✅	❌
RVV1.0	✅	🏗️	✅	✅

- We will use **RVV0.7** since the sources are in C and we will run in the FPGA.
- Load the compiler module:

```
arriesgado-11$ module load llvm/EPI-0.7-development
```

1.2 Compiling and running the scalar code

- Compile the scalar reference code (*SDV_Tutorial/src/reference.c)*

```
arriesgado-11$ make reference.x
```

- And run it:

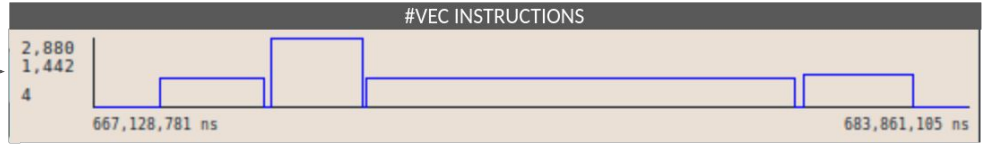
```
arriesgado-11$ ./reference.x
Res: 5620.0032
Res: 1966.3102
Res: 1311.9151
Res: 1039.8516
Res: 881.6927
Res: 785.5846
Res: 715.3618
Res: 666.9991
Res: 627.6045
Res: 598.7502
Microseconds per step: 10947.90
```

1.3 Instrumenting the code

- We will trace our code with **RAVE** (emulation) and **EXTRAE** (native execution).
- Both tracers generate tuples of *(event,value)* over time:

Event: Code Region
Value: *"Region1"*, *"Region2"*, …

Event: #Vec Instructions
Value: *0,1,2,…1000,…*



- We provide a library that unifies both APIs (*sdv_tracing*)

- **trace_init()** → To be called at the start of the application
- **trace_name_event_and_values(event, event_name, nvalues, values[ ], values_names[ ])** → Creates an event and names its values
- **trace_event_and_value(x,y)** → Adds at the current timestamp an event=x with value=y
- **trace_enable()**→ Record events and trace after this function call
- **trace_disable()** → Ignore events and disable tracing after this function call

1.3 Instrumenting the code

- We use **event=1000** as a convention to identify "**Code Region**":

```
#include "sdv_tracing.h"
int main(){
  int event = 1000;
  int values[] = {0,1,2};
  const char * v_names[] = {"Other", "reg1","reg2"};
  trace_name_event_and_values(event, "code_region",
                                  3, values, v_names);
  trace_init();
  trace_disable();
  /*...
  non-important work
  ...*/
  trace_enable();
  trace_event_and_value(1000, 1);
  /*...
  1st region of interest
  ...*/
  trace_event_and_value(1000, 0);
  trace_event_and_value(1000, 2);
  /*...
  2nd region of interest
  ...*/
  trace_event_and_value(1000, 0);
```

Disable and *enable* tracing to exclude regions

Begin a traced region with trace_event(1000, *region_id*) and end it with trace_event(1000,0)

File *SDV_Tutorial/src/reference-i.c* applies this instrumentation methodology to the tutorial code.

NOT NEEDED FOR THIS COURSE!

1.4 Running with Extrae

- Compile the instrumented version:

```
arriesgado-11$ module load sdv_trace
arriesgado-11$ make reference-i.x
```

- Trace the binary with Extrae using the tools we provide (if the binary is run directly, no special behavior or overhead will be observed):

```
arriesgado-11$ trace_extrae_arriesgado ./reference-i.x
```

- Folder *SDV_Tutorial/extrae_prv_traces* contains the execution traces.

- You can copy them back to your computer and open them with Paraver:

```
your-machine$ rsync -a user@ssh.hca.bsc.es:~/SDV_Tutorial/extrae_prv_traces .
your-machine$ wxparaver ./extrae_prv_traces/arr-reference-i.x.prv
```

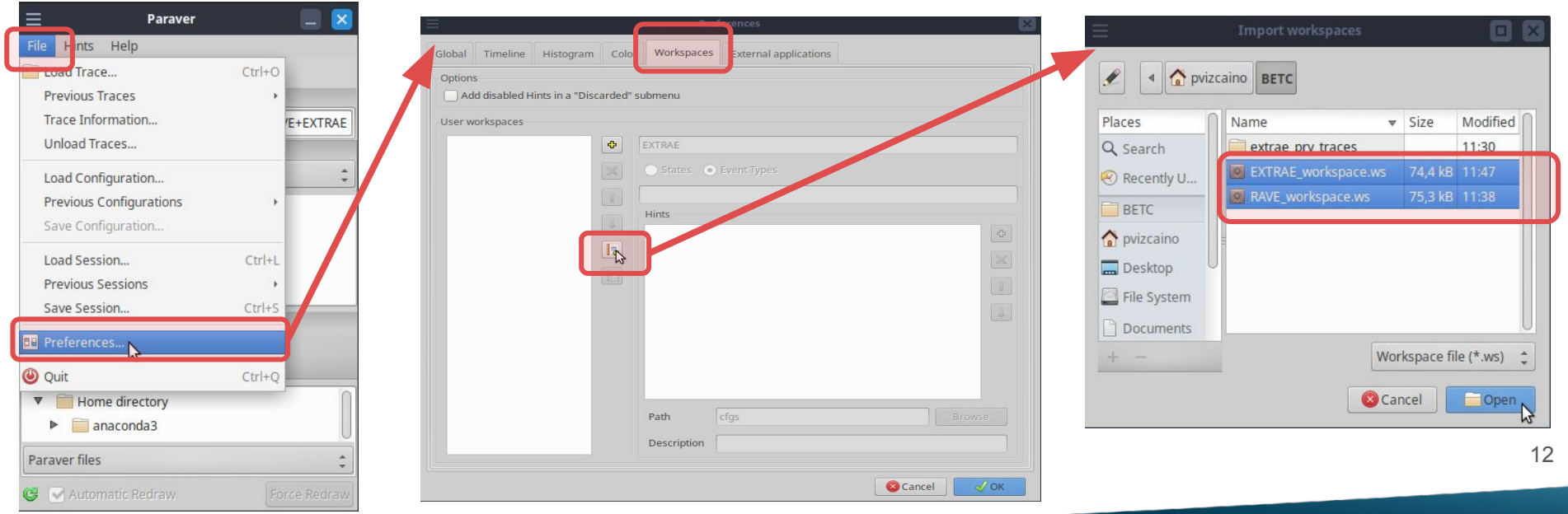
NOT NEEDED FOR THIS COURSE!

1.4 Running with Extrae

- Download the Paraver workspaces we provide:

```
your-machine$ wget https://ssh.hca.bsc.es/epi/ftp/Tutorial/sdv_workspaces.tar.gz
your-machine$ tar -xzvf sdv_workspaces.tar.gz
```

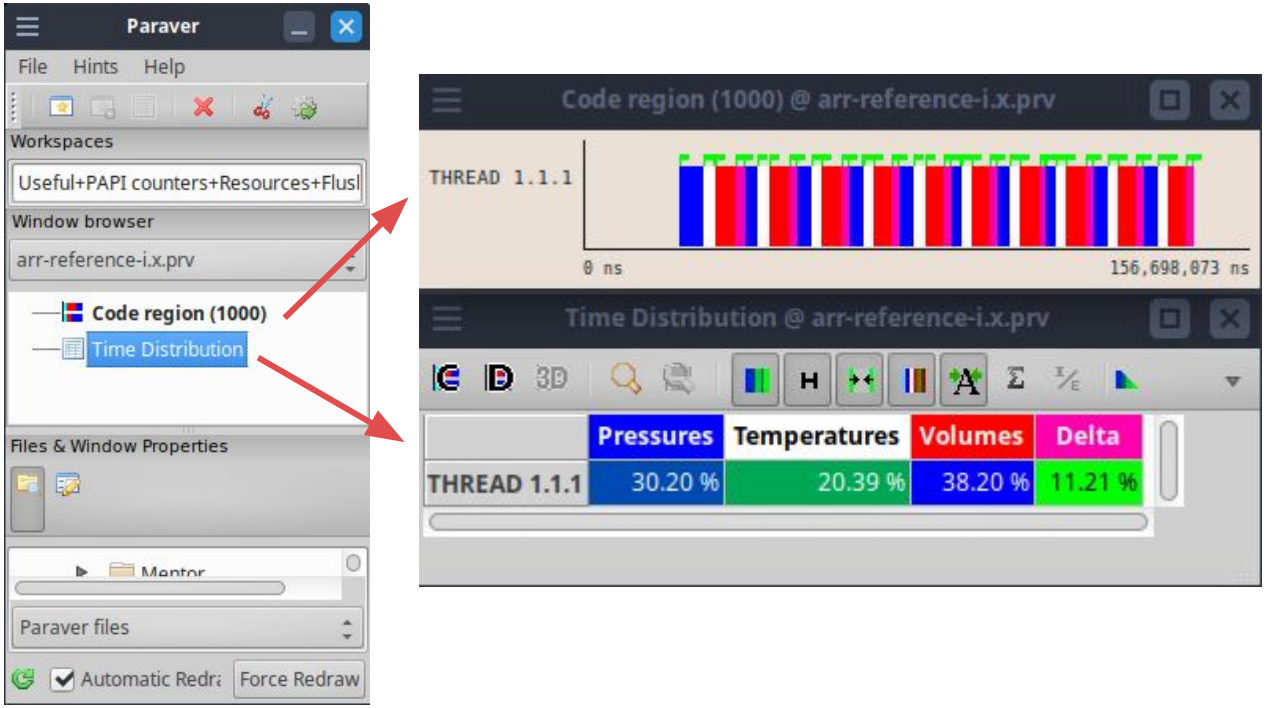
- And load them into Paraver



NOT NEEDED FOR THIS COURSE!

1.4 Running with Extrae

- Load the Hint **Extrae→Phase: Time distribution**



Iterative flow of the code

Pressures and Volumes entail the most amount of time

1.5 Running with PAPI

- You can “avoid” getting Paraver traces, and generate PAPI reports:

```
arriesgado-11$ trace_papi_arriesgado ./reference-i.x
Pressures(1)     PAPI_TOT_CYC     5451970
Pressures(1)     PAPI_TOT_INS     797249
Temperatures(2)  PAPI_TOT_CYC     3307563
Temperatures(2)  PAPI_TOT_INS     1062611
Volumes(3)       PAPI_TOT_CYC     6096505
Volumes(3)       PAPI_TOT_INS     638417
Delta(4)         PAPI_TOT_CYC     1790694
Delta(4)         PAPI_TOT_INS     428781
5620.0032
```

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

2.1 Compiler autovectorization

- BSC's LLVM compiler can autovectorize loops
- We recommend using these flags (scalar and vector):

```
-O3 -ffast-math -mepi -mllvm -combiner-store-merging=0 -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize
-mcpu=avispado -mllvm -vectorizer-use-vp-strided-load-store -mllvm -enable-mem-access-versioning=0 -mllvm
-disable-loop-idiom-memcpy -fno-slp-vectorize
```

- Compile the autovectorized code:

```
arriesgado-11$ make reference-vec.x
```

- If you run this binary natively on Arriesgado, the program will crash (as Unmatched does not support vector instructions):

```
arriesgado-11$ ./reference-vec.x
Illegal instruction (core dumped)
```

2.2 Running and tracing with RAVE

- You can emulate the vectorized binary with RAVE on the synth-hca server

```
hca-server$ ssh synth-hca
synth-hca$ module load rave/EPI-0.7
synth-hca$ rave ./reference-vec.x
```

- You can also generate instrumented RAVE traces of the emulation:

```
synth-hca$ module load sdv_trace
synth-hca$ trace_rave_0_7 ./reference-vec.x
```

- This will leave traces in the folder: *SDV_Tutorial/rave_prv_traces*

```
synth-hca$ ls rave_prv_traces/
rave-reference-vec.x.pcf
rave-reference-vec.x.prv
rave-reference-vec.x.row
```

- And generate a report:

```
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 %)
```

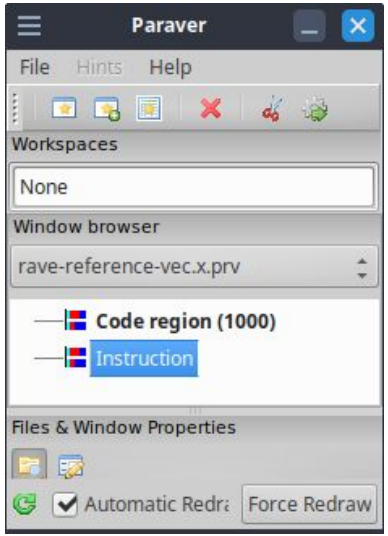
NOT NEEDED FOR THIS COURSE!

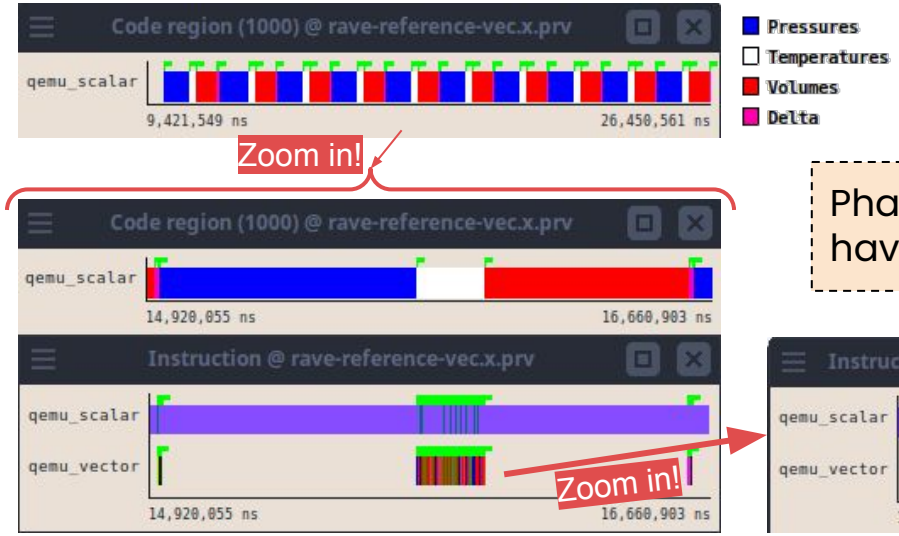
2.2 Running and tracing with RAVE

- Copy the traces back to your machine and open them with Paraver:

```
your-machine$ rsync -a user@ssh.hca.bsc.es:~/SDV_Tutorial/rave_prv_traces .
your-machine$ wxparaver ./rave_prv_traces/rave-reference-vec.x.prv
```

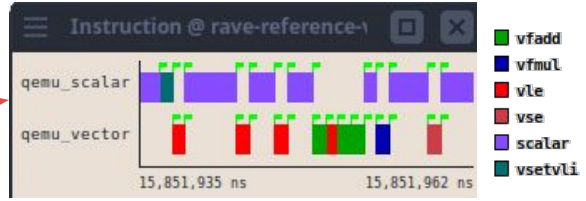
- Load the Hints **RAVE→Phase** and **RAVE→Phase: Time distribution**:





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

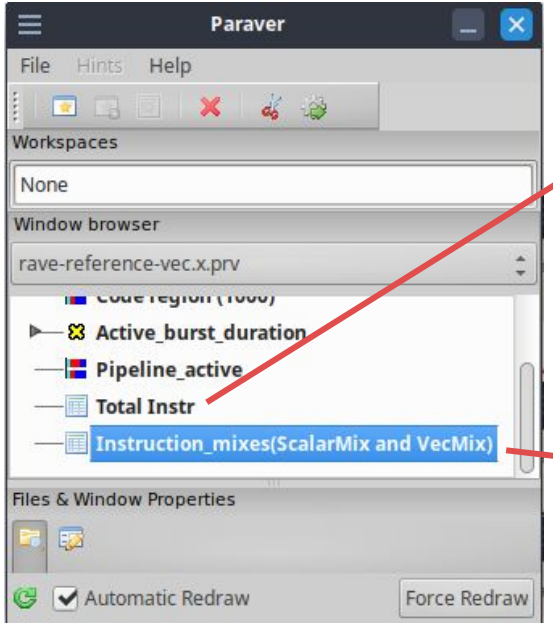
Phases Pressures and Volumes have no vector instructions

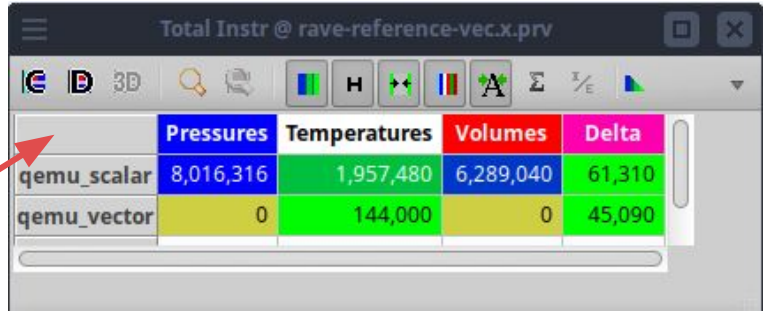


NOT NEEDED FOR THIS COURSE!

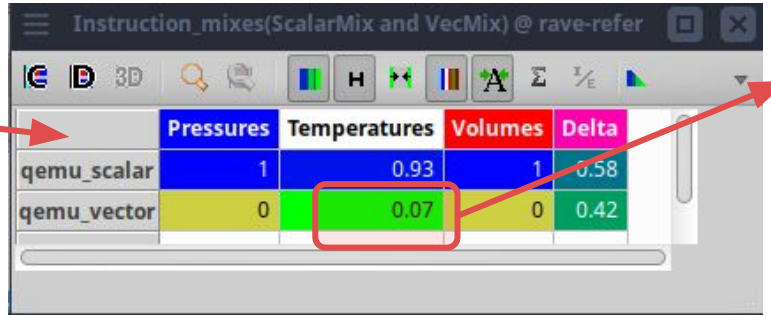
2.2 Running and tracing with RAVE

- Load the Hint **RAVE→Phase: Vector Mix per phase** :





	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



	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:

```
synth-hca$ module load llvm/cross/EPI-0.7-development
synth-hca$ make reference-vec.x
src/reference-i.c:26:43: 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:49:3: 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)

- 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

```
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]);
```

NOT NEEDED FOR THIS COURSE!

2.3 Increasing vectorization

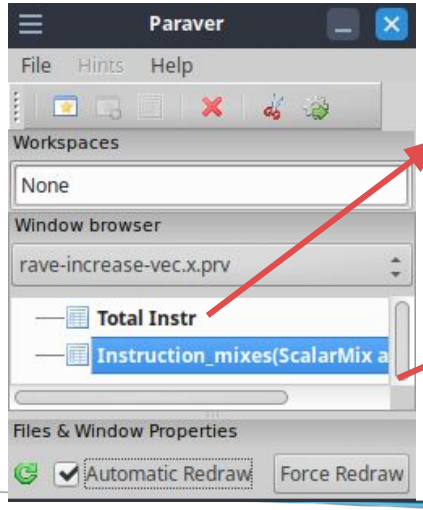
- Compile and trace the improved version *SDV_Tutorial/src/increase-vec.c*

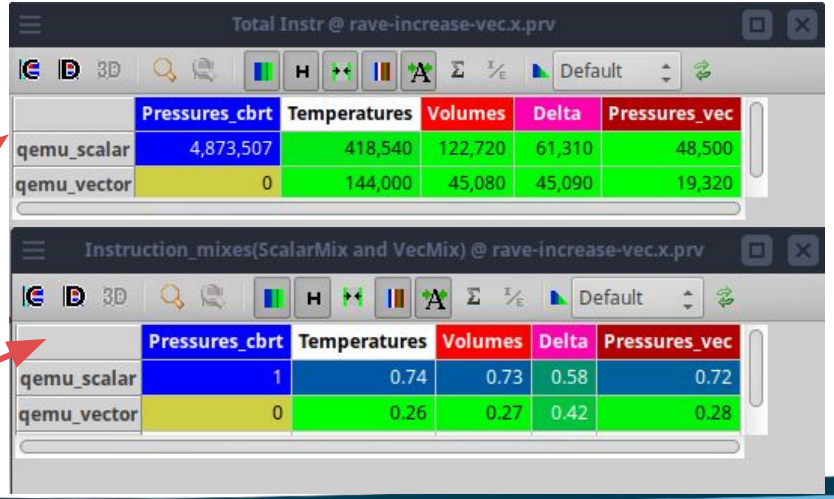
```
synth-hca$ make increase-vec.x
synth-hca$ trace_rave_0_7 ./increase-vec.x
```

- Copy the traces back to your machine and open them with Paraver:

```
your-machine$ rsync -a user@ssh.hca.bsc.es:~/SDV_Tutorial/rave_prv_traces .
your-machine$ wxparaver ./rave_prv_traces/rave-increase-vec.x.prv
```

- Load the Hint **RAVE→Phase: Vector Mix per phase** :





	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

	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

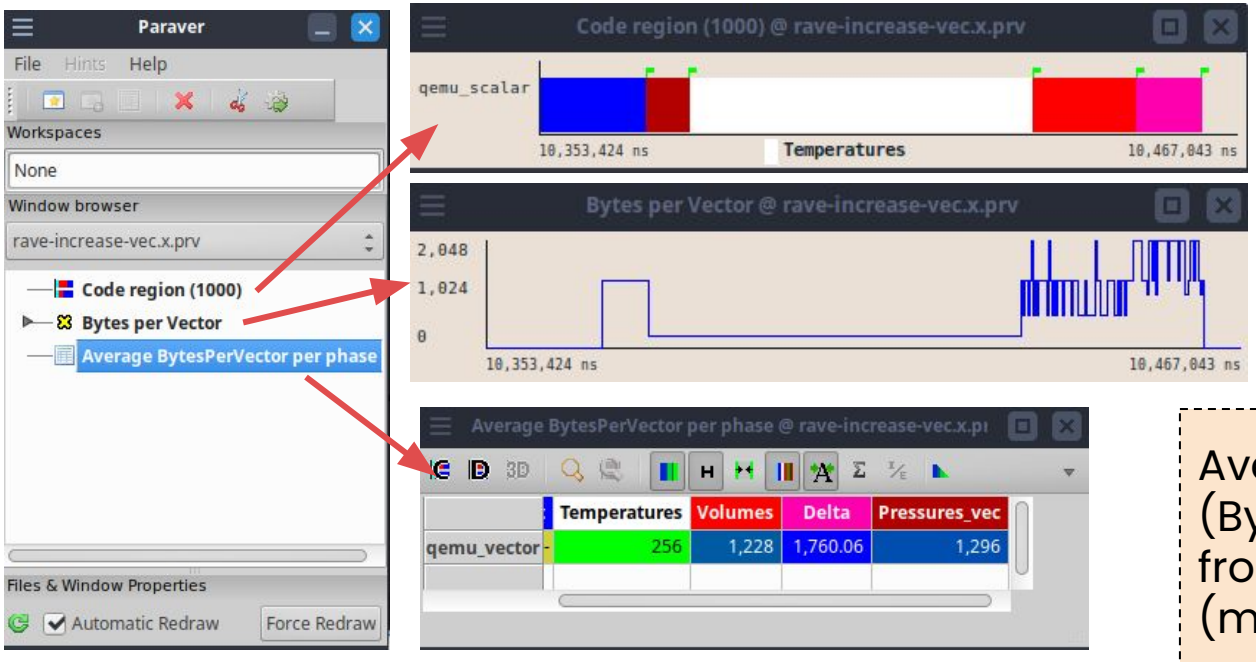
Only the non-vectorizable Pressures_cbrt phase does not have vector instructions

Pressures_vec reports good vector metrics

NOT NEEDED FOR THIS COURSE!

2.4 Increasing the Vector Length

- Load the Hint **RAVE→Phase: Avg VL per phase** :



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)

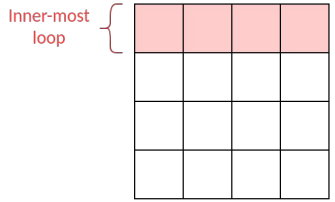
2.4 Increasing the Vector Length

- 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)

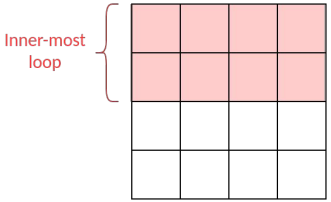
One row at a time

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



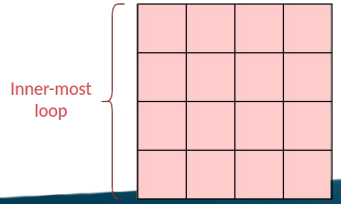
X (e.g., 2) rows at a time

```
#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] …
}
```

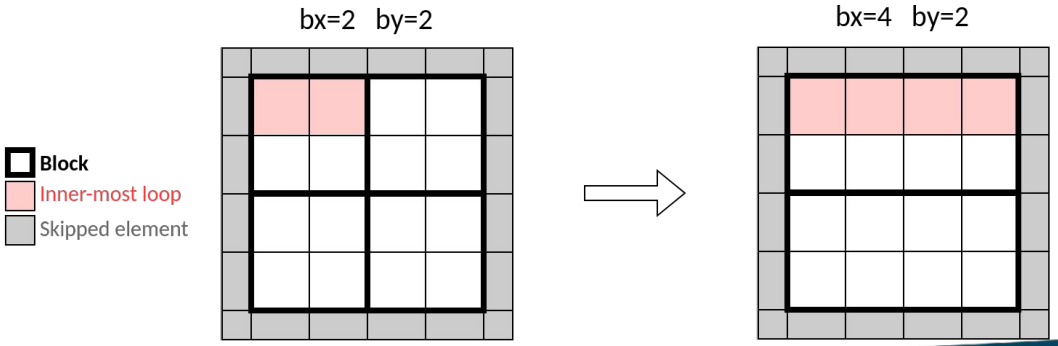


2.4 Increasing the Vector Length

- 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:



NOT NEEDED FOR THIS COURSE!

2.4 Increasing the Vector Length

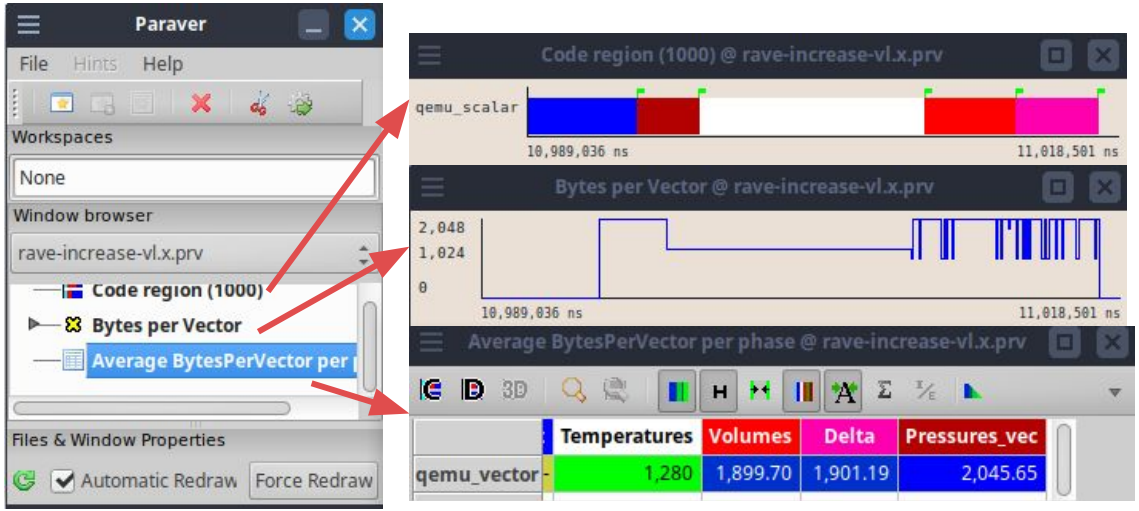
- Compile and trace the improved version *SDV_Tutorial/src/increase-vl.c*

```
synth-hca$ make increase-vl.x
synth-hca$ trace_rave_0_7 ./increase-vl.x
```

- Copy the traces back to your machine and open them with Paraver:

```
your-machine$ rsync -a user@ssh.hca.bsc.es:~/SDV_Tutorial/rave_prv_traces .
your-machine$ wxparaver ./rave_prv_traces/rave-increase-vl.x.prv
```

- Load the Hint **RAVE→Phase: Avg VL per phase** :



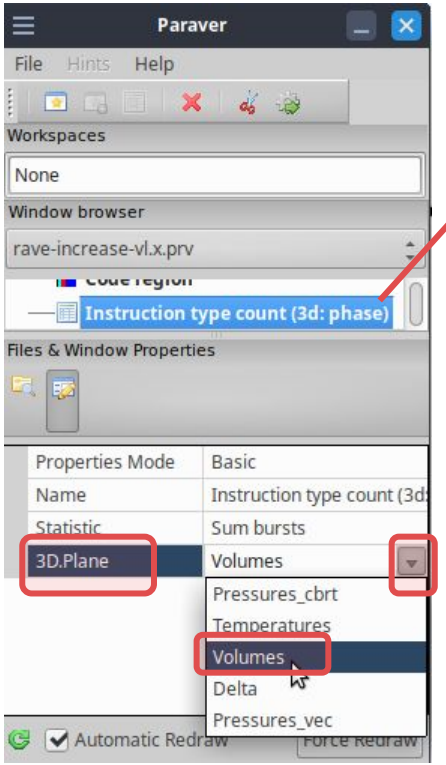
Average Vector Length closer to **2048** Bytes/Vector (maximum in EPAC)

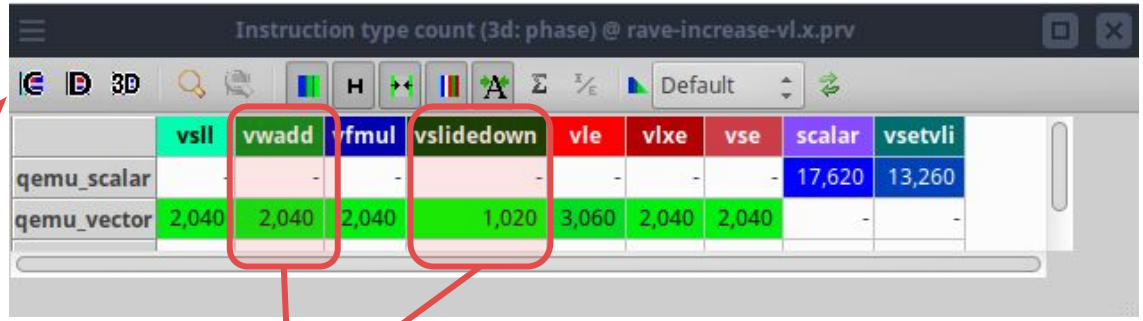
Temperatures increased from **256** to **1280**

NOT NEEDED FOR THIS COURSE!

2.5 Avoid mixing datatypes

- Load **RAVE→Phase: Instruction type count per phase**





	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:

```
double (64b)     double (64b)  int (32b)          double (64b)
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:

```
synth-hca$ make flex-datatype-i-vehave.x
synth-hca$ trace_vehave ./flex-datatype-i-vehave.x
```

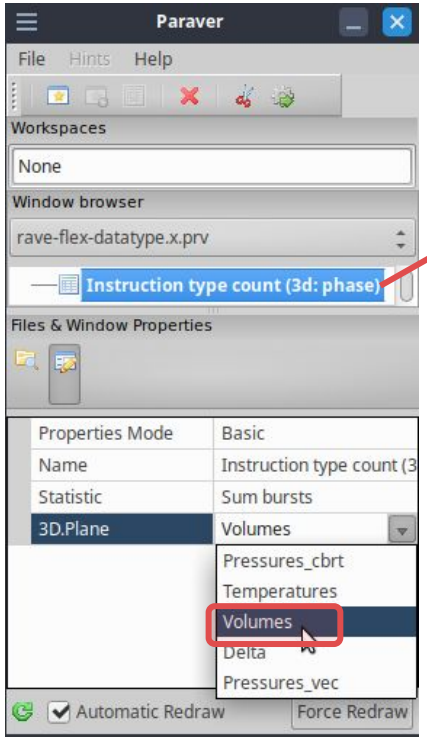
- Copy the traces back to your computer and open them in Paraver:

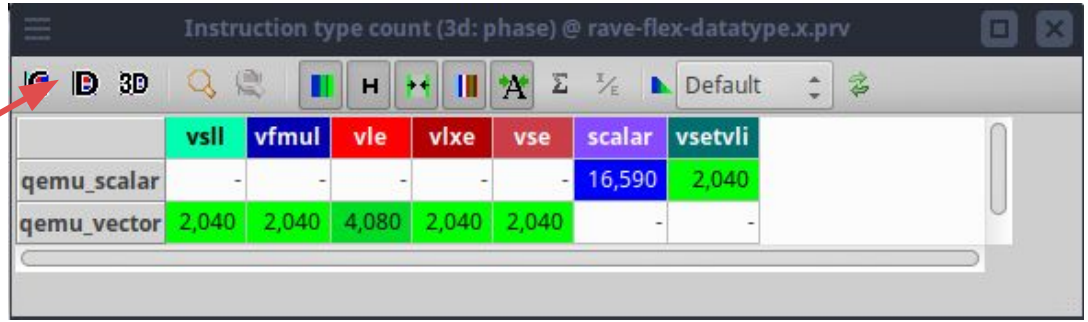
```
your-machine$ rsync -a user@ssh.hca.bsc.es:~/SDV_Tutorial/rave_prv_traces .
your-machine$ wxparaver ./rave_prv_traces/rave-flex-datatype.x.prv
```

NOT NEEDED FOR THIS COURSE!

2.5 Avoid mixing datatypes

- Load the Hint **RAVE→Phase: Instruction type count per phase**





	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 FPGAs 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$ module load sdv_trace #If not already loaded
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

NOT NEEDED FOR THIS COURSE!

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
```

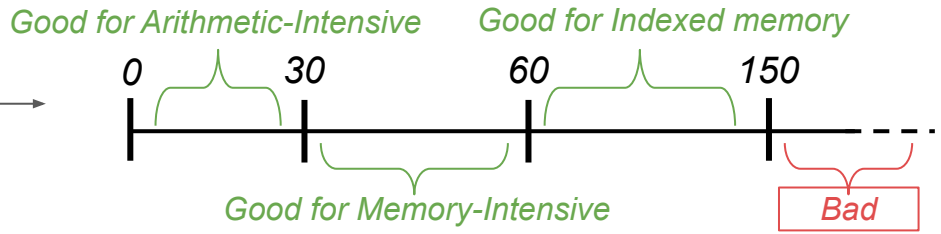
3.2 Getting Extrae traces in the FPGA nodes

- The Paraver Hint **EXTRAE→General Vector Metrics** *computes these vector metrics:*

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

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

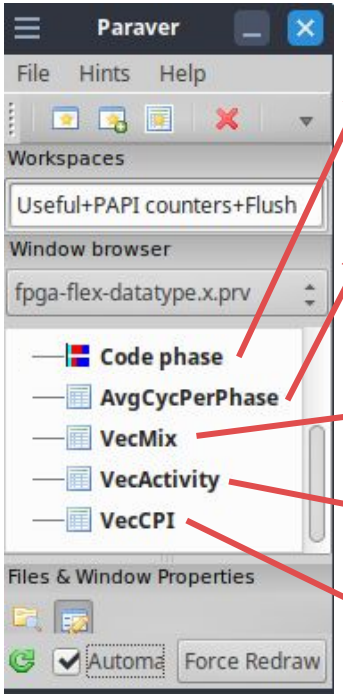
3. ***Vector CPI:*** $\frac{\text{\#Vec. Cyc}}{\text{\#Vec. Instr}}$ →

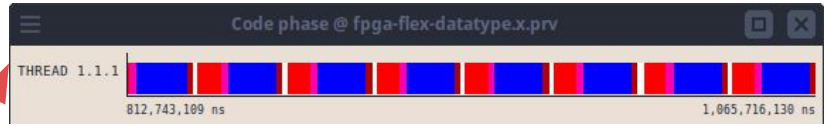


NOT NEEDED FOR THIS COURSE!

3.2 Getting Extrae traces in the FPGA nodes

- Load the Hint **EXTRAE→General Vector Metrics**





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 Getting PAPI reports in the FPGA nodes

- You can “avoid” paraver traces by getting PAPI reports in the FPGAs too:

```
arriesgado-11$ sbatch fpga_job trace_papi_fpga ./flex-datatype.x
```

- And then read the SLURM job output:

```
arriesgado-11$ cat slurm-221457.out
Pressures_cbrt(1)   PAPI_TOT_CYC 4105231
Pressures_cbrt(1)   PAPI_TOT_INS 470316
Pressures_cbrt(1)   VPU_ACTIVE   0
Pressures_cbrt(1)   VPU_COMPLETED_INST  0
Pressures_vec(5)    PAPI_TOT_CYC 477821
Pressures_vec(5)    PAPI_TOT_INS 3181
Pressures_vec(5)    VPU_ACTIVE   470165
Pressures_vec(5)    VPU_COMPLETED_INST  1224
Temperatures(2)     PAPI_TOT_CYC 369542
Temperatures(2)     PAPI_TOT_INS 11400
Temperatures(2)     VPU_ACTIVE   366538
Temperatures(2)     VPU_COMPLETED_INST  2880
Volumes(3)   PAPI_TOT_CYC 1482269
Volumes(3)   PAPI_TOT_INS 6123
(...)
```

3.4 Bonus: Running on Pioneer / Bananapi

- You can run in other machines without allocating them using **srun**:

```
arriesgado-11$ module load llvm/EPI-0.7-development
arriesgado-11$ make flex-datatype.x
arriesgado-11$ srun -p pioneer -t 00:05:00 ./flex-datatype.x
```

```
arriesgado-11$ module load llvm/EPI-development
arriesgado-11$ rm flex-datatype.x ; make flex-datatype.x
arriesgado-11$ srun -p bananaf3 -t 00:05:00 ./flex-datatype.x
```

- Or you can allocate them, prepare jobscripts, …

3.5 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
```

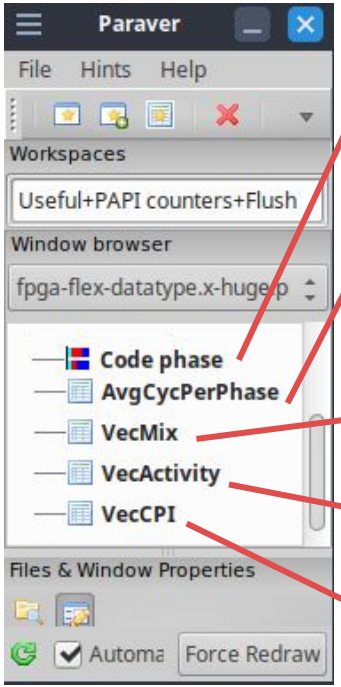
- When the job finishes, copy the traces back to your machine and open them

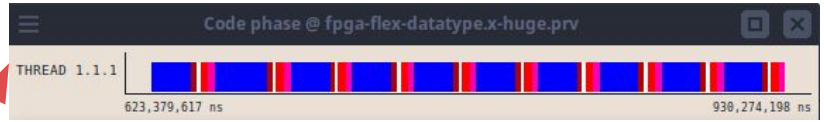
```
your-machine$ rsync -a user@ssh.hca.bsc.es:~/SDV_Tutorial/extrae_prv_traces .
your-machine$ wxparaver ./extrae_prv_traces/flex-datatype.x-huge.prv
```

NOT NEEDED FOR THIS COURSE!

3.5 Using huge pages

- Load the Hint **EXTRAE→General Vector Metrics**





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

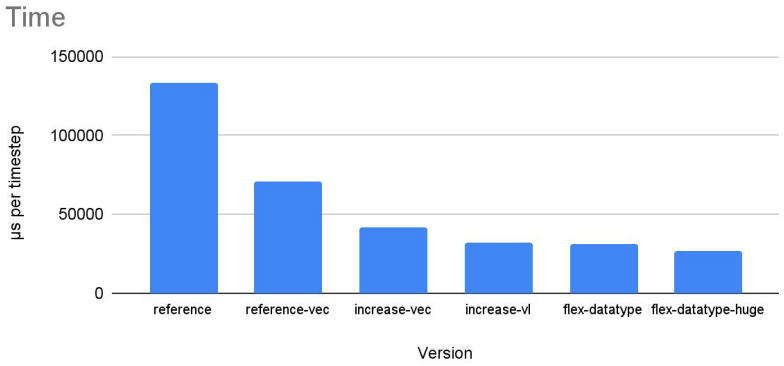
Temperatures	Volumes	Delta	Pressures_vec
29.14	129.71	67.35	53.76

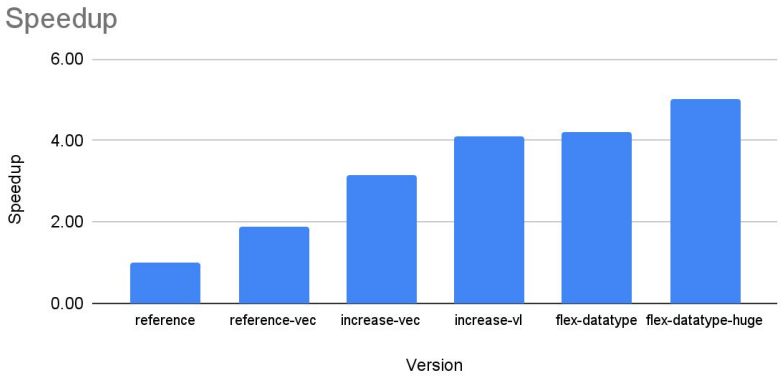
Volumes reduced its average duration from **410k** cycles to **175k**

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

Now it's your turn!

- We give you a "real HPC" code to study: The **Conjugate Gradient**
- You can find the source files in this path:

  `/home/ictp-mhpc25/wednesday_05/conjgrad/`
- The conjugate gradient resolves a system of equations:
  - $Ax = b$ $\longrightarrow$ Where A is a square matrix, and X and B vectors.
- It's an **iterative algorithm** that converges (ends when error < Tolerance).
- It's composed of a few algebra operations:
  - Dot product
  - Scale
  - Axpy
  - GeMV (general matrix-vector product)
- You can run it with three optional arguments: `./cg.x -m M -t T -q`
  - M is the square size of the matrix (MxM), T the tolerance, and Q removes prints

Now it's your turn!

Example: /home/ictp-mhpc25/wednesday_05/SDV_Tutorial/src/reference-i.c

- You need to:
  1) Instrument the code (using "*sdv_tracing.h*" header!)
  2) Evaluate the vectorization (using RAVE)
  3) Modify the code:
     a) Can you improve the vectorization?
     b) Does it make sense to fuse functions together (e.g. Axpy and Scale)
     c) Can you improve it with blocking? Intrinsics?
     d) Is it faster to use "floats" or "doubles" ?
  4) Evaluate it on various nodes:
     a) Arriesgado (without vectors)
     b) Pioneer (with short 0.7.1 vectors) → module load llvm/EPI-0.7-development
     c) Bananapi (with short 1.0 vectors) → module load llvm/EPI-development
     d) FPGA-SDV (with long 0.7.1 vectors) → module load llvm/EPI-0.7-development

- You will prepare a presentation for tomorrow with your findings!

Annex: Instruction latencies

- Rough estimates of vector instruction latencies at 256 DP elements per vector:

Function	Assembly	Latency (256 DP elem)
Division, Sqrt	*vfdiv, vfsqrt*	+2000 cycles
Gather/Scatter	*vlxe, vsxe*	[256 : 2000] cycles
Strided memory	*vlse,vsse*	[128 : 512] cycles
Unit-strided memory	*vle, vse*	[32 : 128] cycles
Widening/Narrowing	*vnsrl, vwadd, vfwcvt, ...*	[128 : 300] cycles
Slides	*vslidedown, vslideup, ...*	[100: 200] cycles
FP reductions	*vfredsum, vfredmin, ...*	160 cycles
Integer reductions	*vredsum, vredand, ...*	100 cycles
Arithmetic	*vfadd,vsub,vfmadd,...*	35 cycles
Moves	*vmv.v.x, vmv.v.v, ...*	50 cycles