ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

APAI Lab01: PULP Embedded Programming

Davide Nadalini, Lorenzo Lamberti, Alberto Dequino, Luca Bompani, Francesco Conti.

(University of Bologna)

d.nadalini@unibo.it

alberto.dequino@unibo.it

Objective of the Class

Intro: PULP platform and the PULP-SDK

Tasks: some basics of C programming on PULP:

- Hello world
- vector sum,
- Matrix-vector multiplication
- Measuring execution performance

Programming Language: C

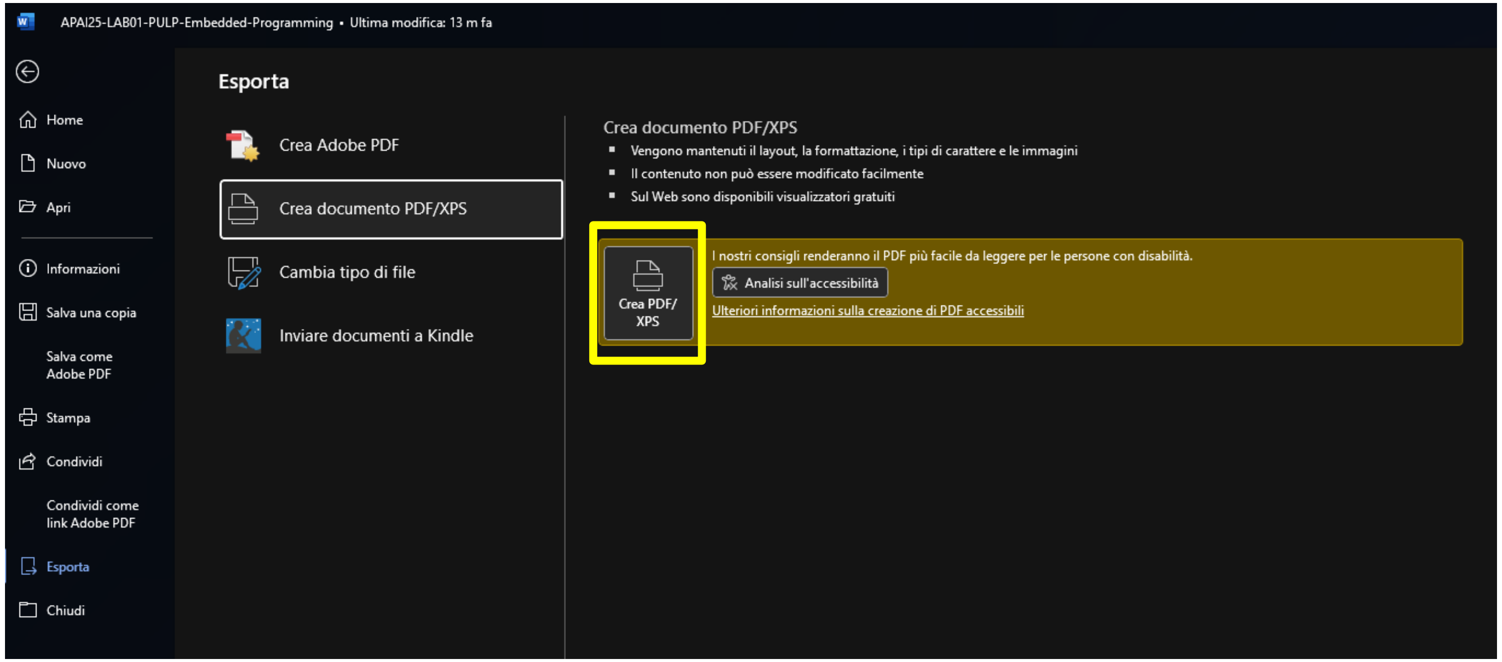
Lab duration: 3h

The class is meant to be interactive: coding together and on your own!

How to deliver the Assignment

You will deliver ONLY the PDF assignment, no code

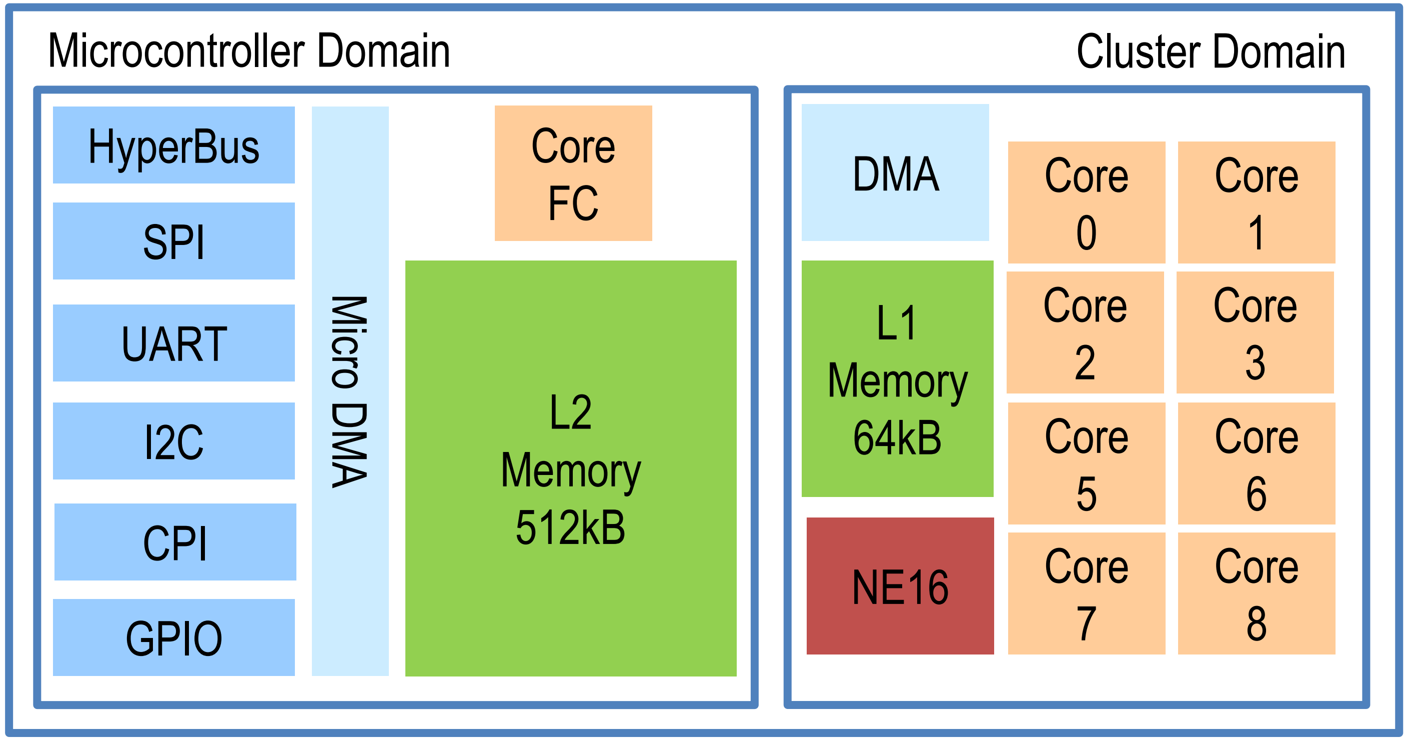
- Download the assignment file from Virtuale.
- Fill the results required by the assignment.
- Export to pdf format.
- Rename the file to: LAB<number_of_the_lesson>_APAI_<your_name>.pdf
- Use Virtuale platform to load ONLY your .pdf file



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

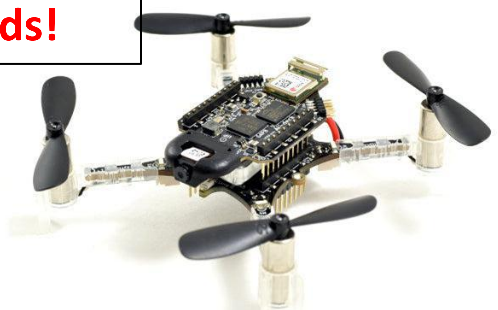
INTRO

Get confident with the PULP Platform



- **Cores**: 1 + 8
- **On-chip Memories**
    - A level 2 Memory, shared among all cores
    - A level 1 Memory, shared by the 8-cores cluster
- **cluster-DMA:** A multi-channel 1D/2D DMA, controlling the transactions between the L2 and L1 memories
- **micro-DMA**: A smart, lightweight and completely autonomous DMA () capable of handling complex I/O scheme
  - **Bus+Peripherals:** HyperBus, I2S, CPI, timers, SPI, GPIOs, etc...

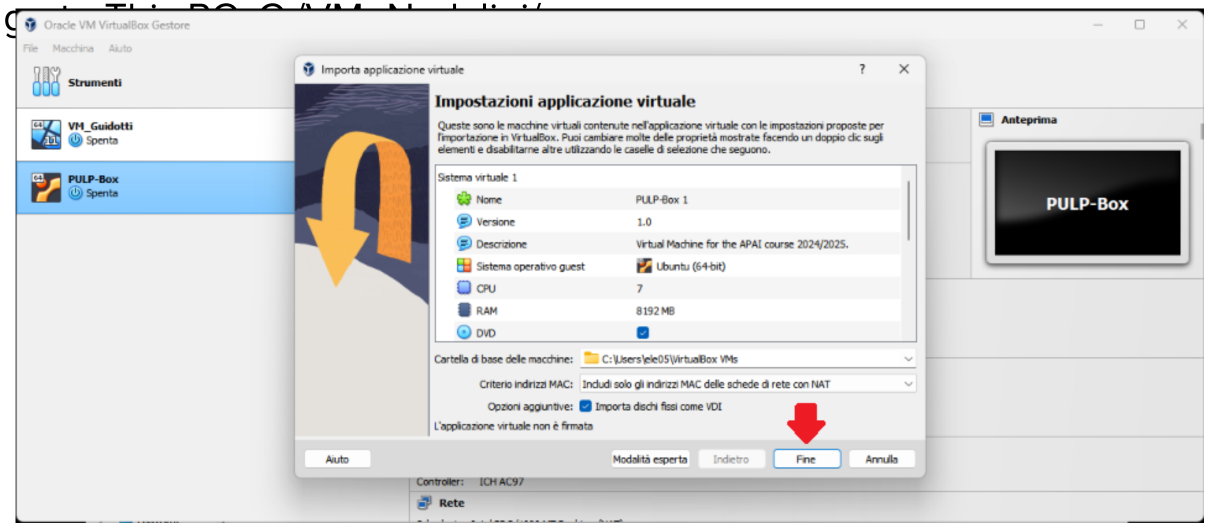
NB: this is the architecture you find on our nano-drones and GAP boards!



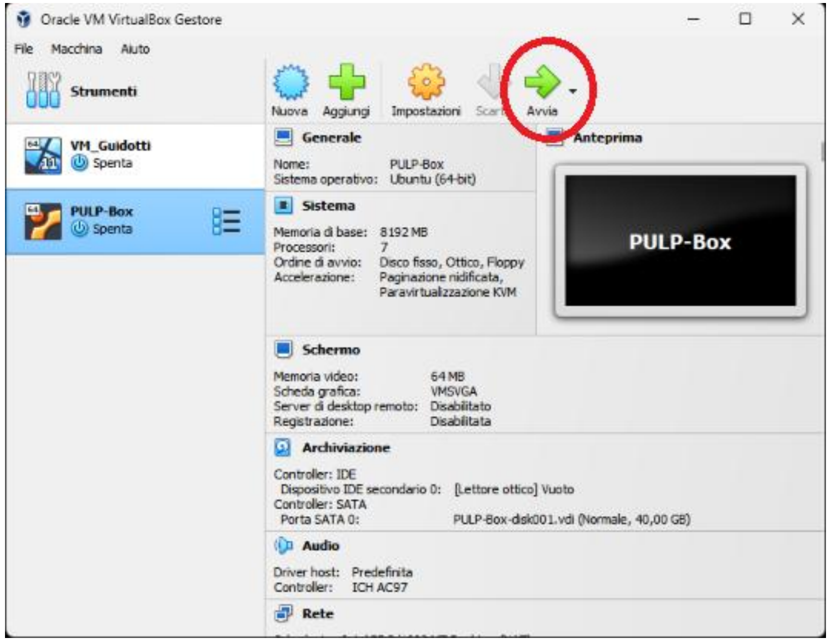
GitHub HW Project: https://github.com/pulp-platform/pulp
HW Documentation: https://raw.githubusercontent.com/pulp-platform/pulp/master/doc/datasheet.pdf

Opening the VM and VSCode

1. On the lab’s PCs, open the file explorer and g



2. Double click on PULP-box.ova
3. VirtualBox opens, just click on “Fine”
4. Wait for the VM to be imported
5. Open the VM with “Avvia”



Password is ‘pulp’

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

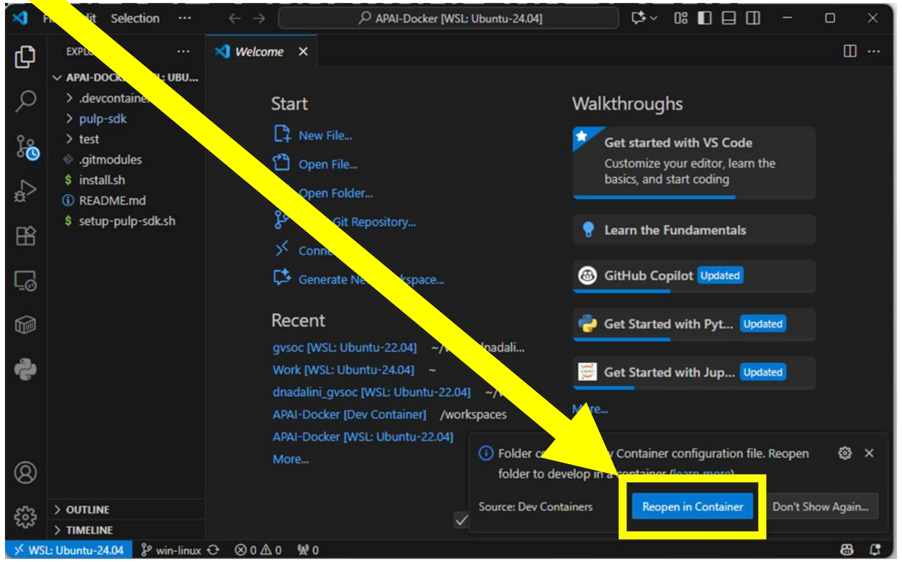
Opening the Docker with VSCode

1. Open a terminal (right click – open a new terminal)
2. From the terminal, open VSCode in the folder of the Docker

```
$ cd APAI-Docker
$ code .
```

1. Reopen the APAI-Docker folder in VSCode (click on “Reopen in container”)

1. Now you can use the integrated terminal (open with CTRL+J) to run your applications!



IMPORTANT: every time you open a new terminal to work on PULP, launch

```
$ source setup-pulp-sdk.sh
```

Getting Started: *Helloworld*

IMPORTANT: activate the pulp-sdk module file every time a new shell is open.

```
$ source setup-pulp-sdk.sh
```

HOW TO RUN THE CODE:

```
$ git clone https://github.com/EEESlab/APAI25-LAB01-PULP-Embedded-Programming
$ cd APAI25-LAB01-PULP-Embedded-Programming
$ cd pulp-helloworld/
$ make clean all run
```

test.c

- Can you see the **Helloworld from PULP!** ?

```
int main()
{
    printf("Helloworld from PULP!\n");
}
```

Behind the box: Build Automation with Makefiles

Clean previous build processes
(folder *BUILD/*)

Compile the program and produce the ***binary code***

Run ***binary code*** *on the target platform*

```
$ make clean all run
```

Build automation is the process of automating the main steps required to create a software, including *compiling*, *assembling*, *linking* and (possibly) *testing*

Make is one of the most widespread utilities

- ➢ configuration files are called **Makefiles**

A *Makefile* contains rules in the form:

```
target: prerequisites
<TAB> command
```

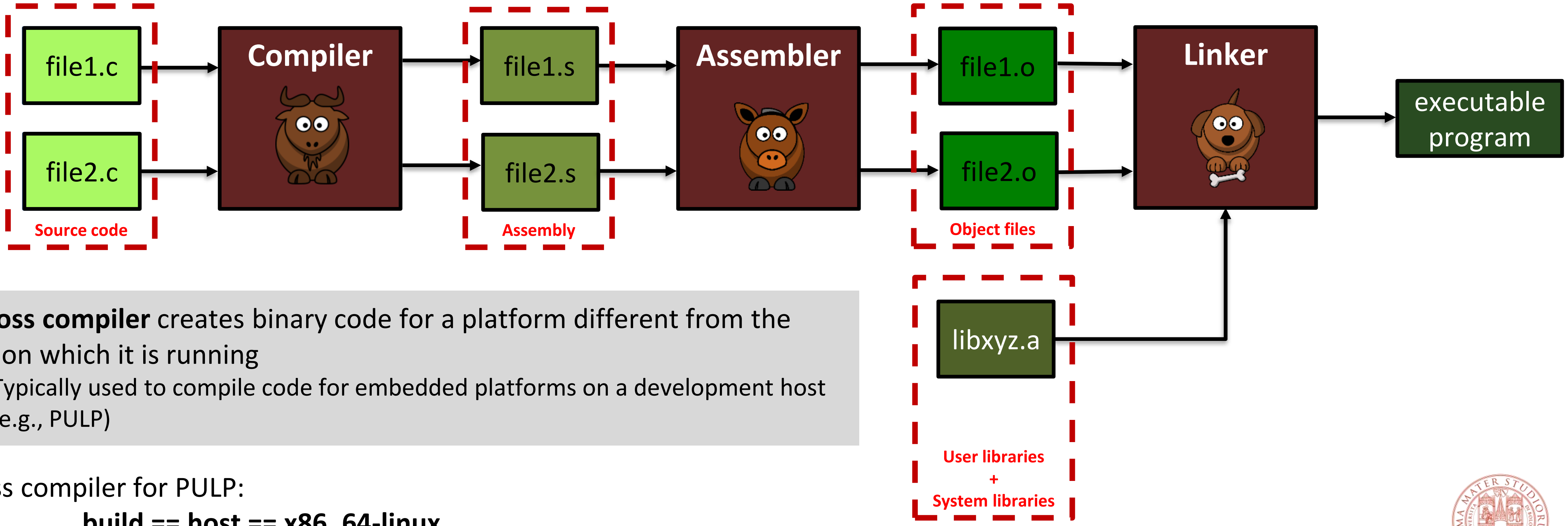
Makefile

```
APP = test

# This is a comment
APP_SRCS    += test.c
APP_CFLAGS  += -O3 -g
APP_LDFLAGS +=

include $(RULES_DIR)/pmsis_rules.mk
```

Compilation toolchain

A compilation toolchain includes several tools to achieve its final goal

- The **COMPILER** translates high-level source code (e.g., C) to a lower-level representation (e.g., assembly)
- The **ASSEMBLER is** program that translates assembly language to machine language
- The **LINKER** combines multiple object files into a single executable



A **cross compiler** creates binary code for a platform different from the one on which it is running

- Typically used to compile code for embedded platforms on a development host (e.g., PULP)

Cross compiler for PULP:

build == host == x86_64-linux

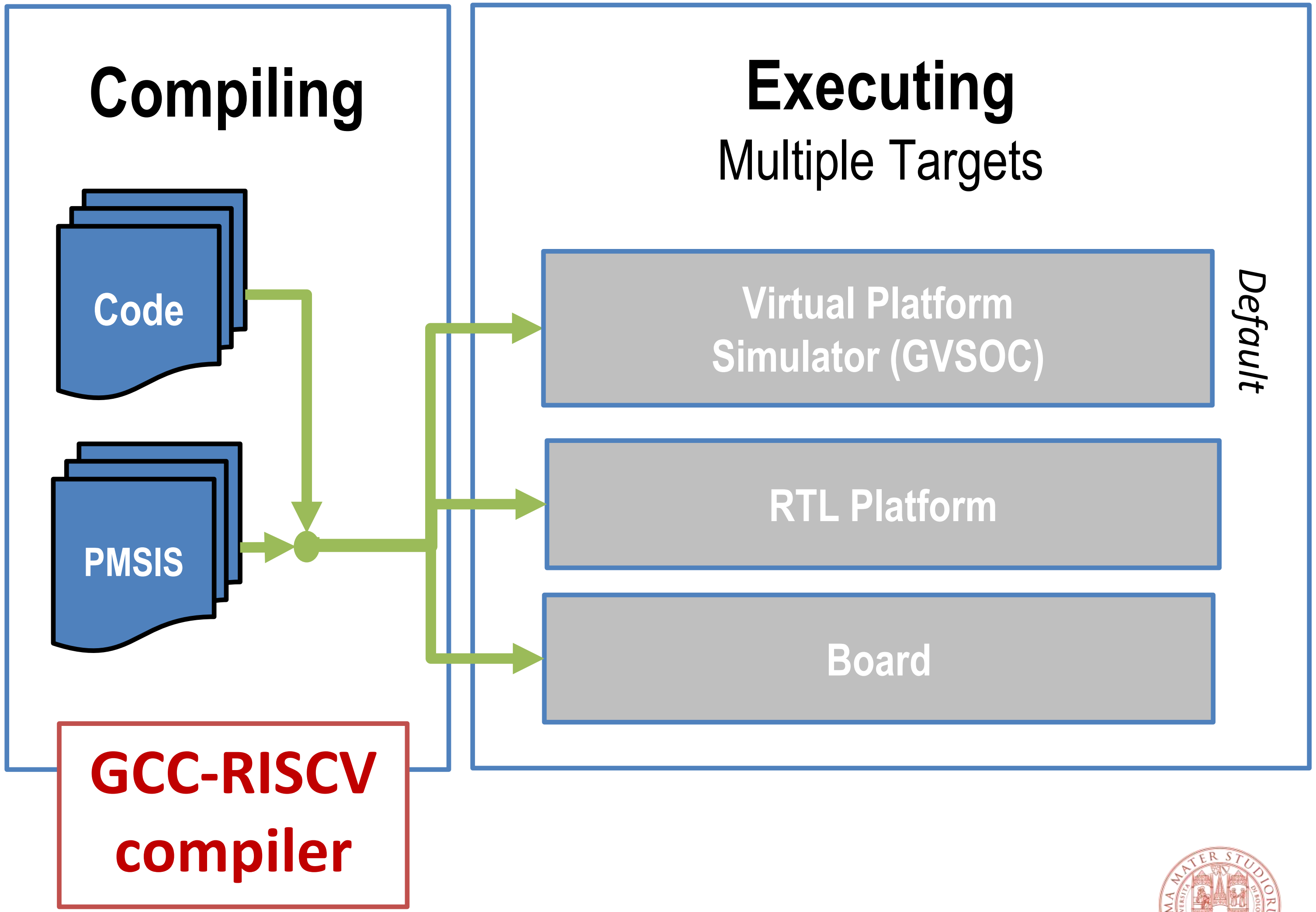
target = riscv-none

PULP Software Environment Workflow

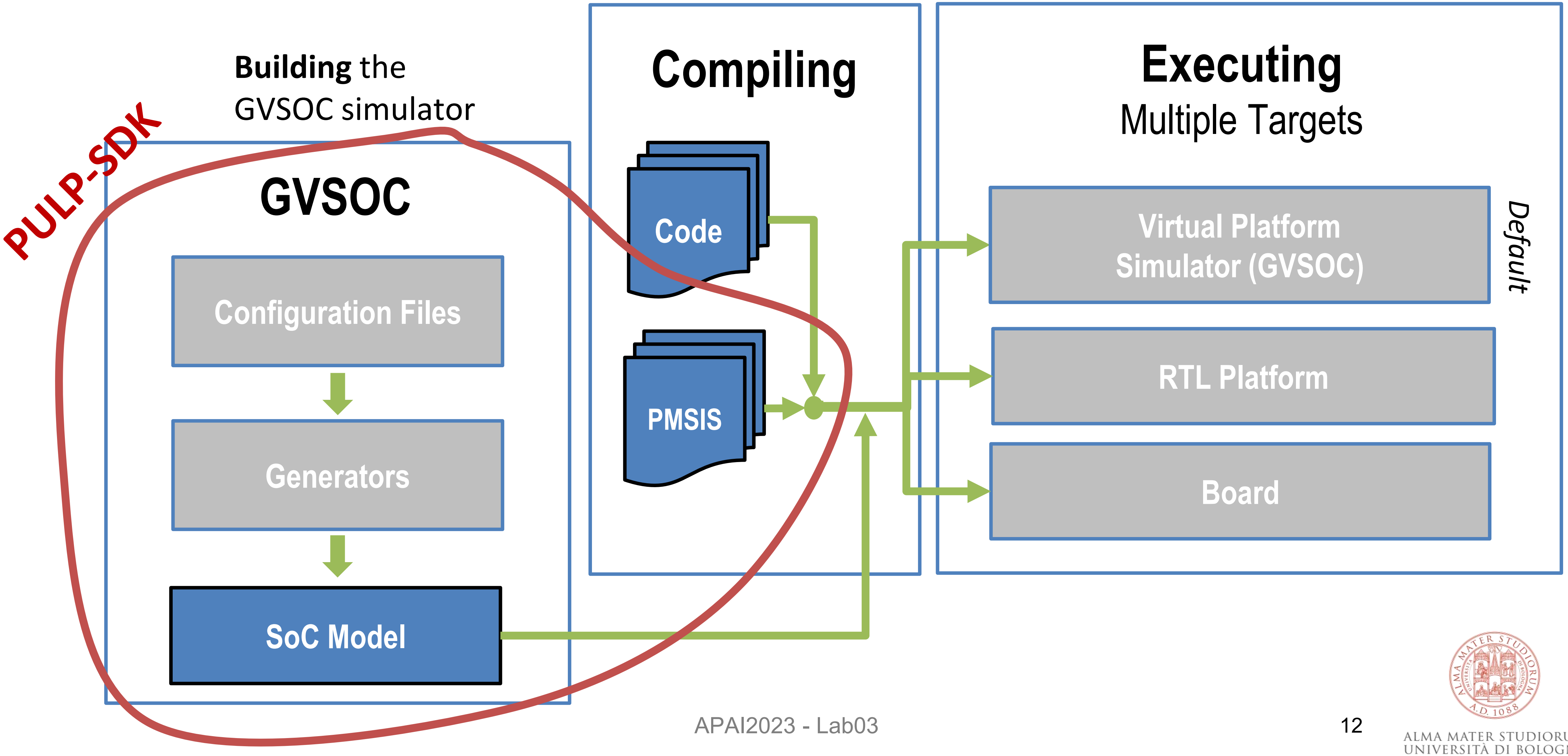
Application code: *main* function, application libraries, e.g. CNN inference function code…

Runtime SW: Peripheral *Drivers,* RTOS, Board support packages (BSP)

- *PMSIS layer*



PULP Software Environment Workflow



PULP SDK (software development kit)

The PULP-SDK (https://github.com/pulp-platform/pulp-sdk) includes a **PULP platform simulator (GVSOC)** and the SW libraries.

Check the */pulp/pulp-sdk* folder:

- `rtos/` runtime code and software stack
- `tools/` configuration files, python generators, pulp runner and the gvsoc components
- `tests/` sample code to test the platform's features
- `applications/` full-application codes

```
$ cd /pulp/pulp-sdk
```

```
pulp-user@pulp-box /pulp/pulp-sdk $ ll
total 60
drwxrwxr-x  9 pulp-user pulp-user  4096 Feb  3  2021 ./
drwxrwxr-x  6 pulp-user pulp-user  4096 Feb  3  2021 ../
drwxrwxr-x  6 pulp-user pulp-user  4096 Feb  3  2021 build/
drwxrwxr-x  2 pulp-user pulp-user  4096 Feb  3  2021 configs/
drwxrwxr-x  8 pulp-user pulp-user  4096 Feb  3  2021 .git/
-rw-rw-r--  1 pulp-user pulp-user    80 Feb  3  2021 .gitignore
drwxrwxr-x  4 pulp-user pulp-user  4096 Feb  3  2021 install/
-rw-rw-r--  1 pulp-user pulp-user 11357 Feb  3  2021 LICENSE
-rw-rw-r--  1 pulp-user pulp-user   877 Feb  3  2021 Makefile
-rw-rw-r--  1 pulp-user pulp-user  2455 Feb  3  2021 README.md
drwxrwxr-x  4 pulp-user pulp-user  4096 Feb  3  2021 rtos/
drwxrwxr-x  2 pulp-user pulp-user  4096 Feb  3  2021 rules/
drwxrwxr-x 10 pulp-user pulp-user  4096 Feb  3  2021 tools/
pulp-user@pulp-box /pulp/pulp-sdk $
```

You could get the latest update from the open-source community and rebuild the GVSOC simulator

```
$ git pull origin main
$ source configs/pulp-open.sh
$ make build
```

Do NOT do it

GVSoC – Features

Virtual platform features:

- *C++* for fast native simulation
- *Python* for instantiation + configuration
- Complete set of traces to see what happen

Timing model:

- Fully-event based, instances can generate events at specific time
- Includes timing models for interconnects, DMACs, memories…
- Performance counters for information from the execution

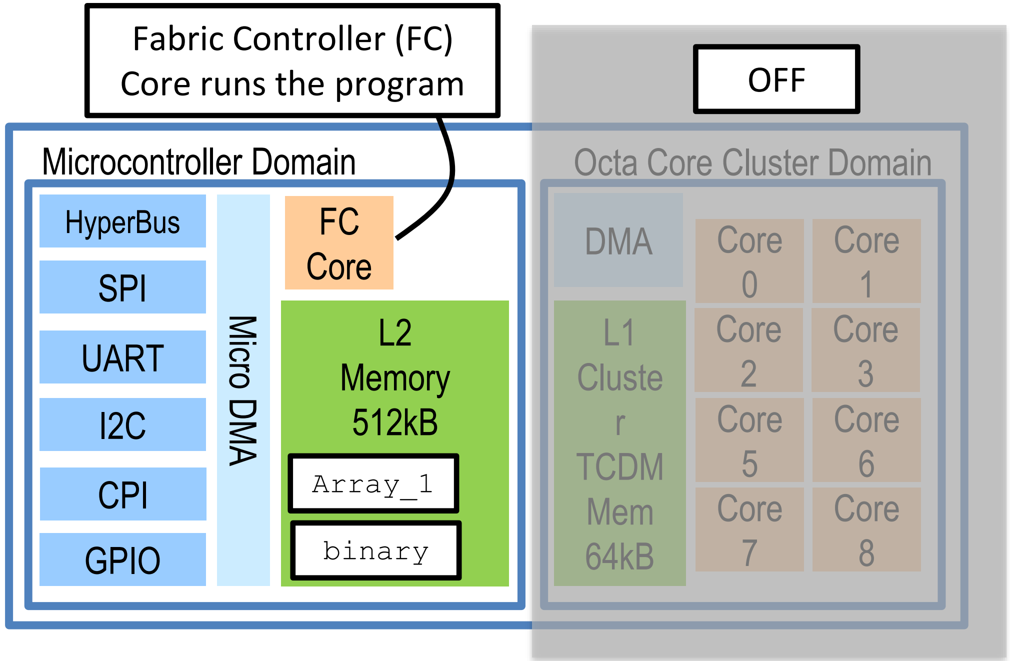
Simulation performance:

- Around 1MIPS simulation speed
- Functionally aligned and calibrated with HW
- Timing accuracy is within 10-20% of target HW

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

TASK1

TASK 1: vector sum example



Preparation

1. Clone repository for the lab, if you didn't already do it

```
git clone https://github.com/EEESlab/APAI25-LAB01-PULP-Embedded-Programming
```

2. Go into folder for task1

```
cd APAI25-LAB01-PULP_Embedded_Programming/
cd vector_sum/
make clean all run
```

Tasks

Read your assignment!

The `vector_sum()` function returns the element-wise add of the values in `array_1[N]`.

Note: the `main` include the function testbench : $S = \sum_{i=1}^{N} i = \frac{N \cdot (N-1)}{2}$

Good practice for test!

I. Array initialization (*a[i] = i*)
II. Function Call
III. Check Result

Task 1.2

Take now a look to the code.
What happen if you change:
(*line 13*) `#define N 350`
➢ **How would you solve it?**

SOLUTION: Casting Variables

(line 13) `#define N 350`

Output: Result is **not** correct. Got 37105 instead of 61425

If printing the `array_1` values after initialization:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 [....] 247 248 249 250 251 252 253 254 255 0 1 2 3 4 5 6 7 8 [...] 88 89 90 91 92 93 94
```

- `unsigned char` datatype can represent numbers from 0 to 255!! If >255, assigned values get truncated (value % 256)!
- **Solution:** cast array_1 to `int` or `short int` (and the function's arguments!)

(optional) Passing parameters via `Makefile`

(OPTIONAL) After applying the fix

We can pass parameters via Makefile: comment line 13, change the *Makefile* and launch:

```
$ make clean all run N=350
```

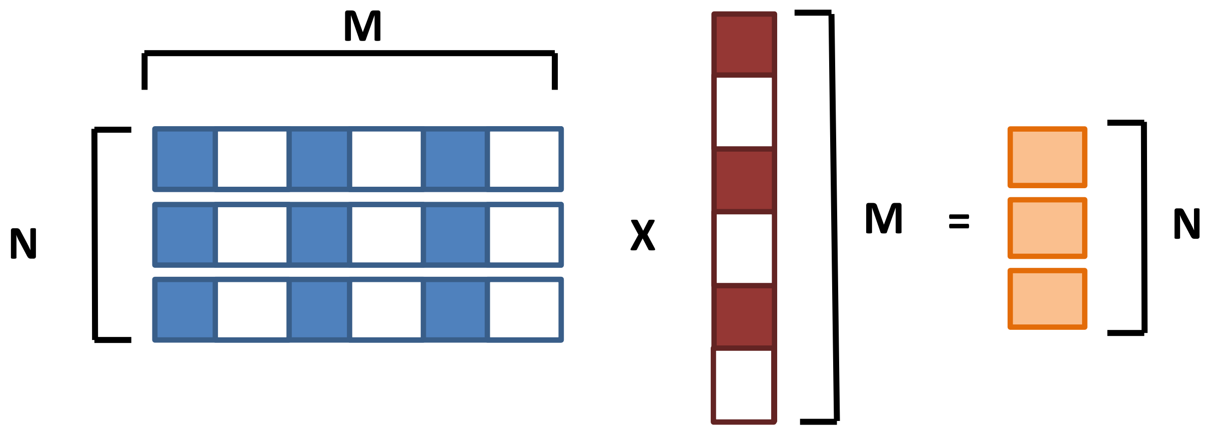
Makefile

```
APP = test
N?=50

APP_SRCS    += test.c
APP_CFLAGS  += -O3 -g
APP_CFLAGS  += -DN=$(N)
APP_LDFLAGS +=
include $(RULES_DIR)/pmsis_rules.mk
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Intro: TASK2

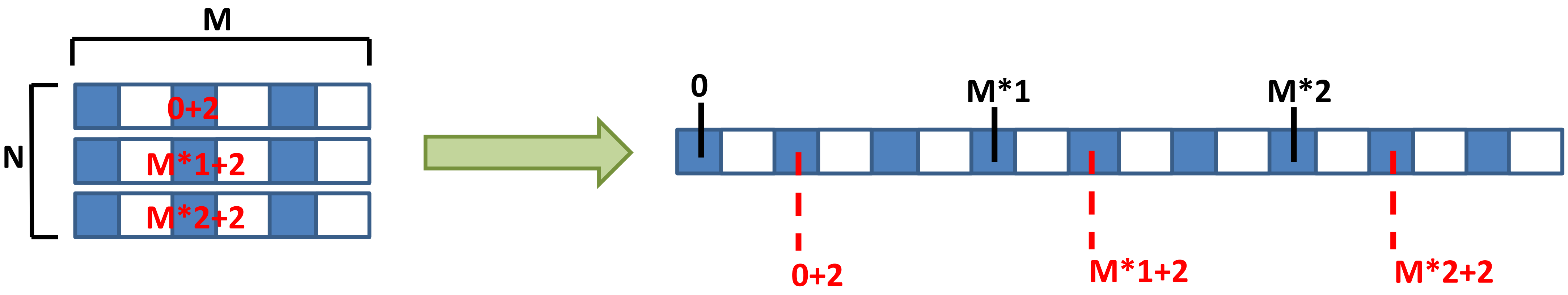
TASK2: Matrix-Vector Product




```
$ cd matrix_vector/
$ make clean all run
```

Example II: Matrix-Vector Product

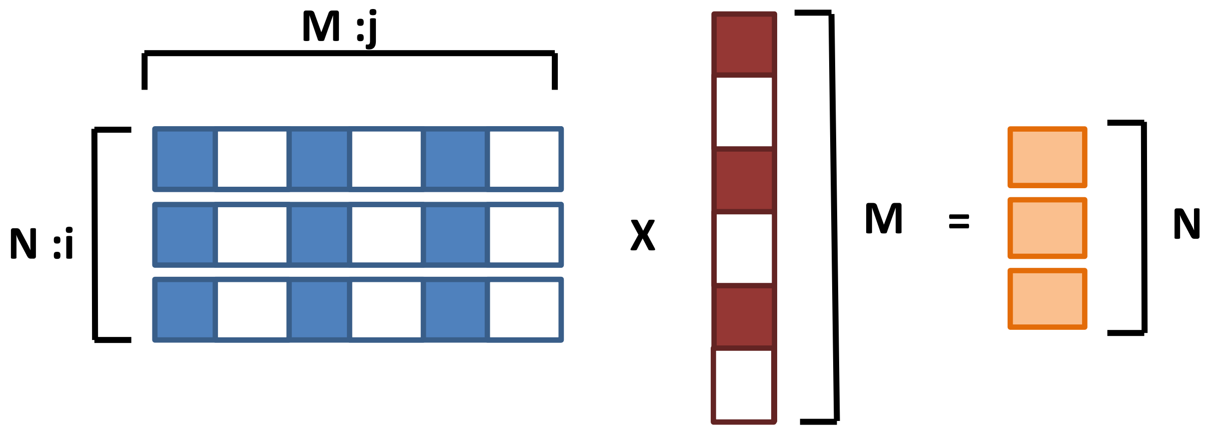
A multi-dimensional array, including a matrix, is efficiently represented as a memory-contiguous array (**NOT** Mat[M][N]).



In general: **Mat[i][j]** -> **M[i*M+j]**

```
$ cd matrix_vector/
$ make clean all run
```

Example II: Matrix-Vector Product




```
// generic matrix-vector multiplication
int gemv(int N, int M, float * mat, float *vec, float * output_vec){

    for (int i=0; i<N; i++){
      for (int j=0; j<M; j++){
        vec_o[i] += mat_i[i*size_M+j] * vec_i[j];
      }
    }

}
```

Assembly Code

A disassembler is a computer program that translates machine language into assembly language → the inverse operation to that of an assembler.

Disassembly, the output of a disassembler, is often formatted for human-readability rather than suitability for input to an assembler, making it principally a reverse-engineering tool. *[Wikipedia]*

To obtain the assembly code use the command:

```
$ make dis > dis.txt
```

Check the RI5CY User manual:

https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf

Assembly Code

```
1c008706 <gemv>:
1c008706:  02a05d63      blez   a0,1c008740 <gemv+0x3a>
1c00870a:  00259f13      slli   t5,a1,0x2
1c00870e:  8fb6          mv     t6,a3
1c008710:  01e68e33      add    t3,a3,t5
1c008714:  4e81          li     t4,0
1c008716:  a005          j      1c008736 <gemv+0x30>
1c008718:  0048230b      p.lw   t1,4(a6!)
1c00871c:  0046a88b      p.lw   a7,4(a3!)
1c008720:  431c          lw     a5,0(a4)
1c008722:  431307b3      p.mac  a5,t1,a7
1c008726:  c31c          sw     a5,0(a4)
1c008728:  ffc698e3      bne    a3,t3,1c008718 <gemv+0x12>
1c00872c:  0e85          addi   t4,t4,1
1c00872e:  967a          add    a2,a2,t5
1c008730:  0711          addi   a4,a4,4
1c008732:  01d50763      beq    a0,t4,1c008740 <gemv+0x3a>
1c008736:  86fe          mv     a3,t6
1c008738:  8832          mv     a6,a2
1c00873a:  fcb04fe3      bgtz   a1,1c008718 <gemv+0x12>
1c00873e:  b7fd          j      1c00872c <gemv+0x26>
```

1c008738:	8832	mv a6,a2
PC	opcode	Instruction

Program counter

Inner loop

```
// generic matrix-vector multiplication
void gemv(int size_N, int size_M,
          float * mat_I, float *vec_i, float * vec_o)
{
    for (int i=0; i<size_N; i++){
      for (int j=0; j<size_M; j++){
        // multiply accumulate operation
        vec_o[i] += mat_i[i*size_M+j] * vec_i[j];
      }
    }
}
```

In the **Makefile**:

1. Change from -01 to -03. What has changed in the assembly?
2. Remove –mnohwloop. What has changed in the assembly?

<u>NB</u>. After changing the compiler flags, compile the code again and generate and disassembly using `make dis`

Assembly Code: O3 optimization

```
APP = matrix-vector

APP_SRCS += test.c
APP_CFLAGS += -O3 -g -mnohwloops
APP_LDFLAGS +=

include $(RULES_DIR)/pmsis_rules.mk
```

```
1c008706 <gemv>:
1c008706:   02a05c63    blez  a0,1c00873e <gemv+0x38>
1c00870a:   02b05a63    blez  a1,1c00873e <gemv+0x38>
1c00870e:   00259e93    slli  t4,a1,0x2
1c008712:   050a        slli  a0,a0,0x2
1c008714:   00a70e33    add   t3,a4,a0
1c008718:   01d68333    add   t1,a3,t4
1c00871c:   0047258b    p.lw  a1,4(a4!) # 10004 <__l1_heap_size+0x20>
1c008720:   87b6        mv    a5,a3
1c008722:   8532        mv    a0,a2
1c008724:   0045288b    p.lw  a7,4(a0!)
1c008728:   0047a80b    p.lw  a6,4(a5!)
1c00872c:   430885b3    p.mac a1,a7,a6
1c008730:   feb72e23    sw    a1,-4(a4)
1c008734:   fef318e3    bne   t1,a5,1c008724 <gemv+0x1e>
1c008738:   9676        add   a2,a2,t4
1c00873a:   feee11e3    bne   t3,a4,1c00871c <gemv+0x16>
1c00873e:   8082        ret
```

The -03 optimized code get rid of a *lw* instruction because the accumulator is kept in the register file!

Inner loop

```
// generic matrix-vector multiplication
void gemv(int size_N, int size_M,
float * mat_I, float *vec_i, float * vec_o)
{
    for (int i=0; i<size_N; i++){
        for (int j=0; j<size_M; j++){
            // multitply accumulate operation
            vec_o[i] += mat_i[i*size_M+j] * vec_i[j];
        }
    }
}
```

Assembly Code: HW loops

```
APP = matrix-vector

APP_SRCS += test.c
APP_CFLAGS += -O3 -g -mnohwloops
APP_LDFLAGS +=

include $(RULES_DIR)/pmsis_rules.mk
```

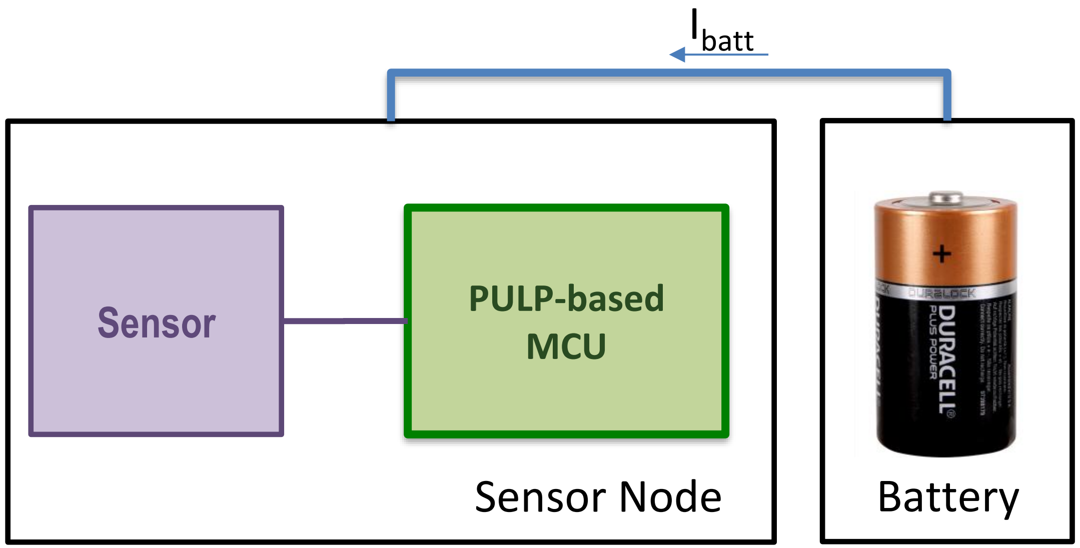
Hardware Loops Instructions have been placed!

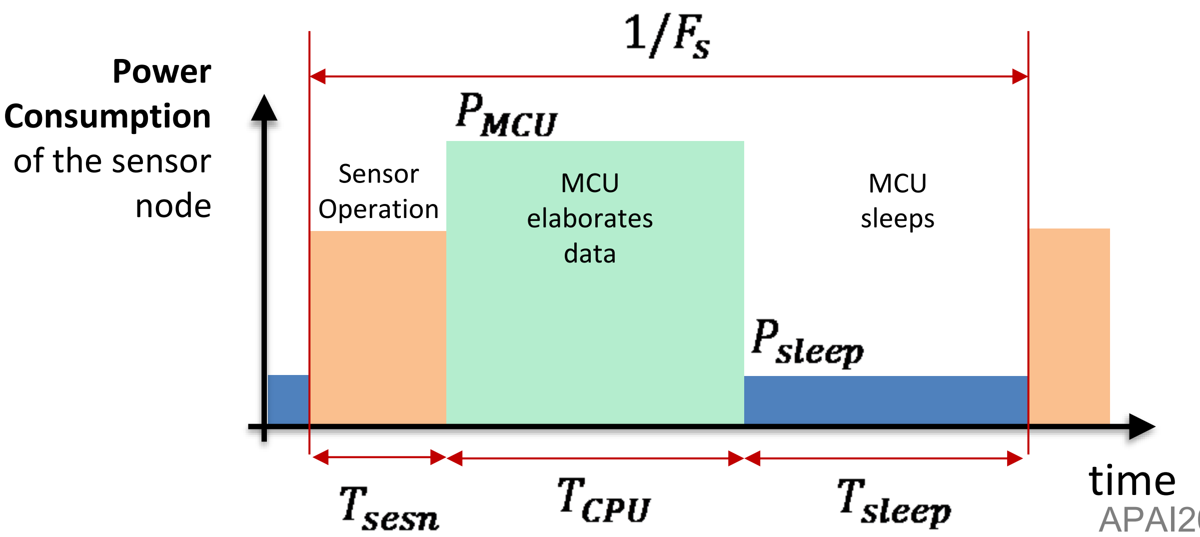
```
1c008706 <gemv>:
1c008706:    04a05463    blez       a0,1c00874e <ge
1c00870a:    04b05263    blez       a1,1c00874e <ge
1c00870e:    00251e13    slli       t3,a0,0x2
1c008712:    1e71        addi       t3,t3,-4
1c008714:    00259e93    slli       t4,a1,0x2
1c008718:    002e5e13    srli       t3,t3,0x2
1c00871c:    01d68f33    add        t5,a3,t4
1c008720:    0e05        addi       t3,t3,1
1c008722:    015e407b    lp.setup   x0,t3,1c00874c <gemv+0x46>
1c008726:    40df07b3    sub        a5,t5,a3
1c00872a:    17f1        addi       a5,a5,-4
1c00872c:    0047258b    p.lw       a1,4(a4!) # 10004 <__l1_heap_size+0x20>
1c008730:    8389        srli       a5,a5,0x2
1c008732:    8836        mv         a6,a3
1c008734:    8532        mv         a0,a2
1c008736:    0785        addi       a5,a5,1
1c008738:    0087c0fb    lp.setup   x1,a5,1c008748 <gemv+0x42>
1c00873c:    0045230b    p.lw       t1,4(a0!)
1c008740:    0048288b    p.lw       a7,4(a6!)
1c008744:    431305b3    p.mac      a1,t1,a7
1c008748:    feb72e23    sw         a1,-4(a4)
1c00874c:    9676        add        a2,a2,t4
1c00874e:    8082        ret
```

```
// generic matrix-vector multiplication
void gemv(int size_N, int size_M,
float * mat_I, float *vec_i, float * vec_o)
{
    for (int i=0; i<size_N; i++){
     for (int j=0; j<size_M; j++){
        // multiply accumulate operation
        vec_o[i] += mat_i[i*size_M+j] * vec_i[j];
     }
    }
}
```

Inner loop

Assessing the MCU performance





Goal

- Extending the battery lifetime
  - Minimize the sensor node energy consumption
- Real time processing of sensor data

$$\min E_s + E_{MCU} \quad \text{s.t. } T_{CPU} < 1/F_s$$

Assuming:

- a fixed sample rate F_s a negligible sensor energy cost $E_s \ll E_{MCU}$(e.g., $T_{sesn} \ll 1/F_s$)
- a constant power envelope of the MCU for active (P_{MCU}) and sleep ($P_{sleep} \ll P_{MCU}$) modes

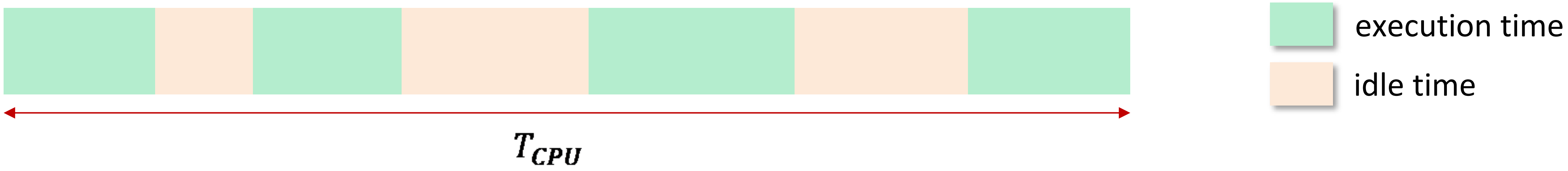
$$\min T_{CPU}\, P_{MCU} + T_{sleep} P_{sleep} \qquad T_{CPU} + T_{sleep} = 1/F_s$$

Aim: minimize the processing time T_{CPU}

What can we optimize? -> the execution time

The processing time T_{CPU} composes of:

- the **execution time**, where the CPU
- the **idle time**, where the CPU waits for events or IRQ (the core may be clock gated to save power)



Given a CPU clock frequency, the performance of a task (e.g. a C function) can be measured by accounting:

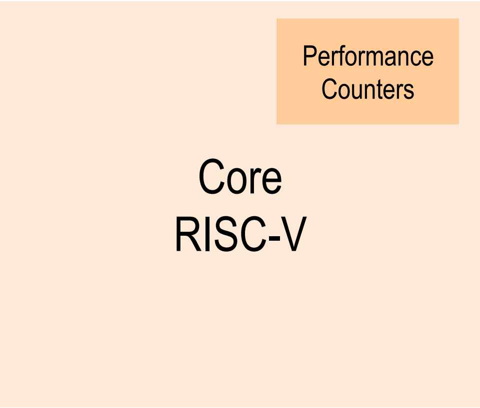
- the number of **elapsed clock cycles** (N_{clk})
- the **number of instructions** (N_{instr}) executed within the task depends on the code optimization!
- **CPI (Clock Cycles Per Instruction)**: different instructions may take a different number of **clock cycles** (depending on the CPU microarchitecture)

$$T_{CPU} = N_{clk} \times T_{clk} = CPI_{avg} \times N_{instr} \times T_{clk}$$

$$CPI_{avg} = \frac{N_{clk}}{N_{instr}}$$

$\boldsymbol{N_{clk}}$ and $\boldsymbol{N_{instr}}$ **greatly** affects CPU performance!

Performance Counters: Measuring the CPU Time



Each RI5CY cores of the PULP platform provide a **performance counter**. These 32-bit counters can be configured to count the:

- Total number of cycles (also includes the cycles where the core is sleeping)
- Number of cycles the core was active (not sleeping)
- Number of instructions executed
- Number of load data hazards
- Number of jump register data hazards
- Number of cycles waiting for instruction fetches, i.e. number of instructions wasted due to non-ideal caching
- Number of data memory loads executed. Misaligned accesses are counted twice
- Number of data memory stores executed. Misaligned accesses are counted twice
- Number of unconditional jumps (j, jal, jr, jalr)
- Number of both taken and not taken branches
- Number of taken branches
- Number of compressed instructions executed

The performance counters of the cluster cores can also account:

- Number of memory loads to EXT executed. Misaligned accesses are counted twice. Every non-L1 access is considered external
- Number of memory stores to EXT executed. Misaligned accesses are counted twice. Every non-L1 access is considered external
- Number of cycles used for memory loads to EXT. Every non-L1 access is considered external
- Number of cycles used for memory stores to EXT. Every non-L1 access is considered external
- Number of cycles wasted due to L1/log-interconnect contention

Important: while on the PULP Virtual Platform multiple perf counters can be enabled concurrently, only one counter is available on a real HW device!

Using the performance Counters on PULP

Using the PMSIS library APIs:

```c
// enable the perf counters of interest
pi_perf_conf(    1 << PI_PERF_CYCLES |
                 1 << PI_PERF_INSTR );

// reset the performance counters
pi_perf_reset();
// start the performance counters
pi_perf_start();

// task to profile
foo();

// stop the performance counters
pi_perf_stop();

// collect and print statistics
uint32_t instr_cnt = pi_perf_read(PI_PERF_INSTR);
uint32_t cycles_cnt = pi_perf_read(PI_PERF_CYCLES);
```

Configure which performance counters to enable!

```c
typedef enum {
  PI_PERF_CYCLES        = 17, /*!< Total number of cycles (also includes the
    cycles where the core is sleeping). Be careful that this event is using a
    timer shared within the cluster, so resetting, starting or stopping it on
    one core will impact other cores of the same cluster. */
  PI_PERF_ACTIVE_CYCLES = 0,  /*!< Counts the number of cycles the core was
    active (not sleeping). */
  PI_PERF_INSTR         = 1,  /*!< Counts the number of instructions executed.
  */
  PI_PERF_LD_STALL      = 2,  /*!< Number of load data hazards. */
  PI_PERF_JR_STALL      = 3,  /*!< Number of jump register data hazards. */
  PI_PERF_IMISS         = 4,  /*!< Cycles waiting for instruction fetches, i.e.
    number of instructions wasted due to non-ideal caching. */
  PI_PERF_LD            = 5,  /*!< Number of data memory loads executed.
    Misaligned accesses are counted twice. */
  PI_PERF_ST            = 6,  /*!< Number of data memory stores executed.
    Misaligned accesses are counted twice. */
  PI_PERF_JUMP          = 7,  /*!< Number of unconditional jumps (j, jal, jr,
    jalr). */
  PI_PERF_BRANCH        = 8,  /*!< Number of branches. Counts both taken and
    not taken branches. */
  PI_PERF_BTAKEN        = 9, /*!< Number of taken branches. */
  PI_PERF_RVC           = 10, /*!< Number of compressed instructions
    executed. */
  PI_PERF_LD_EXT        = 12, /*!< Number of memory loads to EXT executed.
    Misaligned accesses are counted twice. Every non-TCDM access is considered
    external (cluster only). */
  PI_PERF_ST_EXT        = 13, /*!< Number of memory stores to EXT executed.
    Misaligned accesses are counted twice. Every non-TCDM access is considered
    external (cluster only). */
  PI_PERF_LD_EXT_CYC    = 14, /*!< Cycles used for memory loads to EXT.
  Every non-TCDM access is considered external (cluster only). */
  PI_PERF_ST_EXT_CYC    = 15, /*!< Cycles used for memory stores to EXT.
  Every non-TCDM access is considered external (cluster only). */
  PI_PERF_TCDM_CONT     = 16, /*!< Cycles wasted due to TCDM/log-interconnect
  contention (cluster only). */
} pi_perf_event_e;
```

/rtos/pmsis/pmsis_api/include/pmsis/chips/default.h

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

TASK2

Profile the *gemv*

Make a new project by copying the *matrix-vector gemv* example into a new folder

```
$ cd matrix_vector/
make clean all run
```

For any of the previous compiler optimization, **measure the performance** using the PMSIS performance counters and report:

- number of clock cycles.
- number of instructions.
- Number of Multiply-Add operations
- the CPI.
- compute the number of clock cycles and instructions per *elementary operation*.
  - Define 1 elementary operation == 1 Multiply-and-Accumulate

$$a \leftarrow a + (b \times c)$$

#MAC = N x M = 50 x 50 = 2500

	-01	-03	-03 HWLoops
Clock Cycles			
Instr.			
MAC			
CPI			
Intr/Cycles			
Instr / MAC			

SOLUTION: Profile the *gemv*

Make a new project by copying the *matrix-vector gemv* example into a new folder

```
$ cd matrix_vector/
make clean all run
```

Solution:

```
  pi_perf_conf(1<<PI_PERF_CYCLES | 1<<PI_PERF_INSTR);
  pi_perf_reset();

  pi_perf_start();
  // call the matrix-vector fucntion
  gemv(N, M, matrix, vector, output_vec);
  pi_perf_stop();

  uint32_t instr_cnt = pi_perf_read(PI_PERF_INSTR);
  uint32_t cycles_cnt = pi_perf_read(PI_PERF_CYCLES);
  printf("Num.Istr: %d Num.Cycles: %d \n", instr_cnt,
cycles_cnt);
```

#MAC = N x M = 50 x 50 = 2500

	-01	-03	-03 HWLoops
Clock Cycles			
Instr.			
MAC			
CPI			
Intr/Cycles			
Instr / MAC			

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA