APAI2024 - LAB02

PULP_Embedded_Programming

Authors: Davide Nadalini, Lorenzo Lamberti, Alberto Dequino, Manuele,Rusci, Francesco Conti
Contacts: d.nadalini@unibo.it, lorenzo.lamberti@unibo.it, alberto.dequino@unibo.it

Links: GitHub Link (code), GDoc Link (solution template)

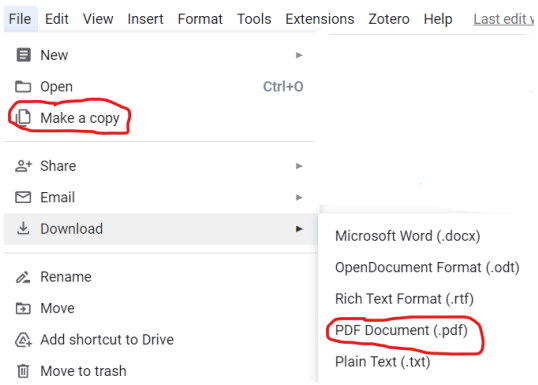
Summary

1. Subject(s):
    - PULP architecture, vector sum, matrix-vector mul, profiling code execution;
2. Programming Language: C;
3. Lab duration: 3h
4. Objective: Embedded programming & profiling
   you will learn basics of embedded programming, the pulp architecture, basic operations (sum & matmul), and how to profile your code execution (MAC, cycles) !
5. Assignment:
    - 2 tasks
    - Time for delivery: 1 week
    - **Submission deadline: Oct 18th 2024**

How to deliver the assignment

You will deliver ONLY THIS TEXT FILE, no code

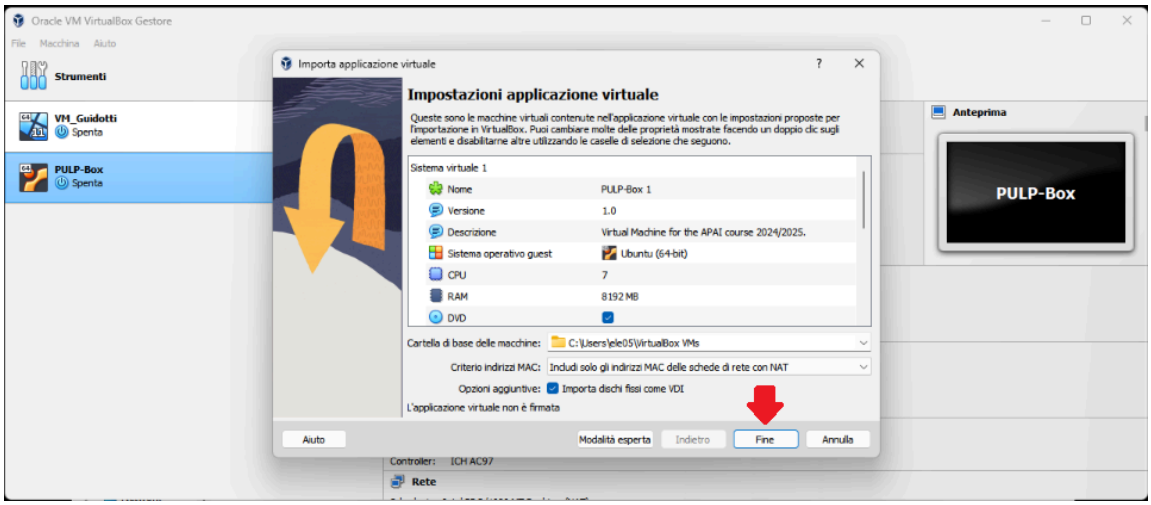
- Copy this google doc to your drive, so that you can modify it. `(File -> make a copy)`
- Fill the tasks on this google doc.
- Export to pdf format.
- Rename the file to: LAB<number_of_the_lesson>_APAI_<your_name>.pdf
- Use Virtuale platform to load ONLY your .pdf file



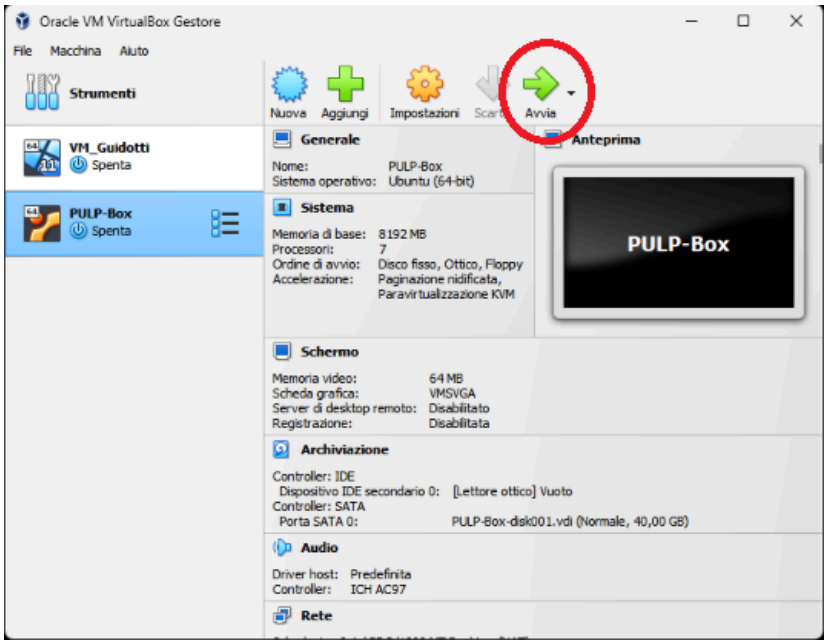
LAB STARTS HERE

0. Access to the local VM and setup pulp-sdk

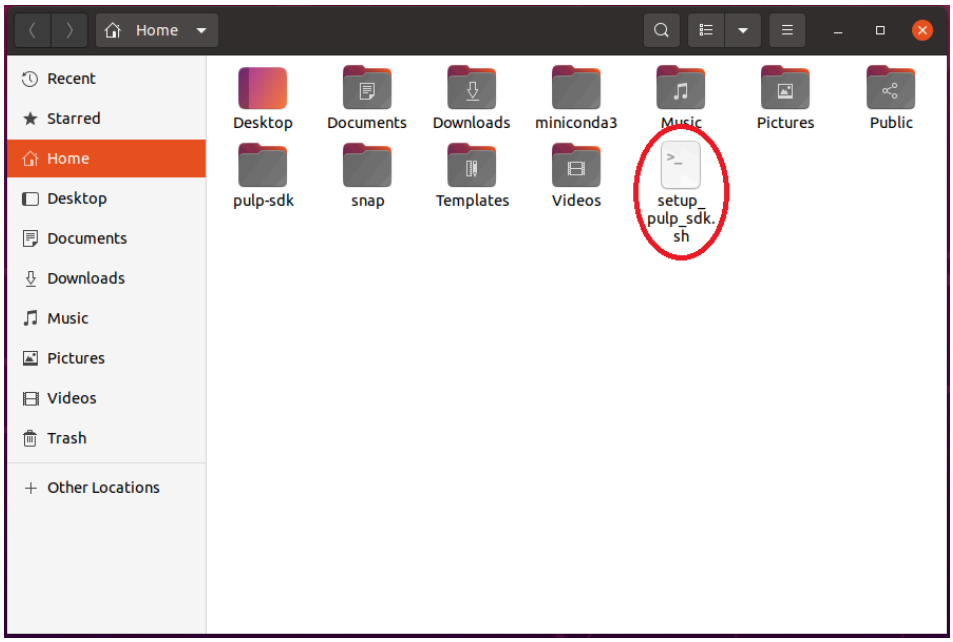
- On the lab's PCs, open the file explorer and go to This PC, C:/VM_Nadalini
- Double click on PULP-box.ova
- VirtualBox opens, just click on "Fine"



- Wait for the VM to be imported
- Open the VM with "Avvia"



- Password is ‘pulp’
- Open a terminal (right click – open a new terminal)
- Setup the PULP-SDK: `source setup_pulp_sdk.sh`



- Clone GitHub repository of today's lab: `git clone https://github.com/EEESlab/APAI24-LAB02-PULP-Embedded-Programming`
- `cd APAI24-LAB02-PULP-Embedded-Programming`
- `cd pulp-helloworld`
- `make clean all run`

If the output is a "Hello World!", then everything is correctly set up!

Task 1: vector sum

0. Setup:

- Switch to the main folder: `cd`
- Open VSCode: `code .`
- Go to "vector_sum/" folder
- Every time you want to run the code, **SAVE your file** and write in the terminal "**make clean all run**"

1.1. Define N to 50 (N= vector size) and run the code:

	Question	Anwer
1	What's the result of the vector_sum() function?	
2	Is the checksum correct ?	
3	Print all the elements of "array_1". What's the output?	

Tips:

- ***Question 3:*** *to print the element of the array, uncomment the function call "print_array(array_1, N)" inside the main()*

1.2. Define N to 350 (N= vector size) and run the code:

	Question	Anwer
1	What's the result of the vector_sum() function?	
2	Is the checksum correct ?	
3	Print all the elements of "array_1". What's the output?	
4	Array_1 should be filled with increasing values from 1 to N. Why isn't the case here?	

Tips:

- ***Question 3:*** *to print the element of the array, uncomment the function call "print_array(array_1, N)" inside the main()*
- ***Question4:*** *what range of values a "char" data type can represent?*

1.3. Fix the issue: cast "array_1" and the function's arguments to int (or short int)

Write down your solution

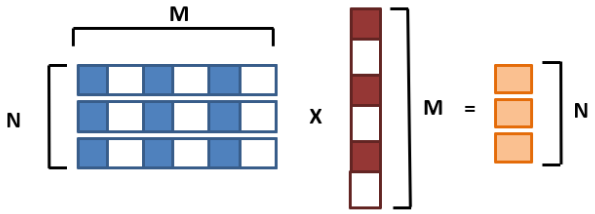
before	After your fix
`unsigned char array_1[N];`	
`int init_array(unsigned char * A_ar, int size)`	
`void print_array(unsigned char * A_ar, int size)`	

`int vector_sum(unsigned char * A_ar, int size)`	

After the fix, answer again

	Question	Anwer
1	What's the result of the vector_sum() function?	
2	Is the checksum correct ?	
3	Print all the elements of "array_1". What's the output?	

Task 2: matrix-vector product & profiling



0. Setup:

- Go to "matrix_vector_product/" folder
- Every time you want to run the code, **SAVE your file** and write in the terminal "**make clean all run**"

2.1. Implement missing code

GOAL: We want to count how many Multiply-ACcumulate operations we perform for a GEMV

TASK: Increment a counter every time we perform a MAC inside the GEMV inner loop. I prepared a variable called mac_counter. This is the reference code

```
int mac_counter=0;
/* generic matrix-vector multiplication */
int __attribute__((noinline)) gemv(int size_N, int size_M, int *mat_i, int *vec_i, int *vec_o)
{
    for (int i = 0; i < size_N; i++) // outer looop of the gemv
    {
        for (int j = 0; j < size_M; j++) // inner loop of the gemv
        {
            // multiply accumulate operation (MAC)
            vec_o[i] += mat_i[i * size_M + j] * vec_i[j];
            //*(vec_o+i) += *(mat_i+i*M+j)*(*(vec_i+j)); // try to uncomment this and comment the above line
        }
    }
}
```

Put your solution below (code)

[HERE]

2.2. Implement missing code part II

Add performance counters to profile the gemv. The gemv is calculated with this line of code in the `main()`:

```
gemv(N, M, matrix, vector, output_vec);
```

To profile it, exploit these two functions I prepared for you. You should start the profiling right before and stop it right after

```
start_perf_counter();
stop_perf_counter();
```

Put your solution below (code)

[HERE]

2.3. Implement missing code part III

Enable the performance counters of our interest. We want to profile:

- Execution cycles (total)
- N° instructions executed

Here's the full list of the performance counters

```
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;
```

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

You can enable them in the `start_perf_counter()` function.
Complete the code where you find /* YOUR CODE HERE */ with the right performance counters (see previous figure)

```
void start_perf_counter()
{
    // enable the perf counter of interest
    pi_perf_conf(   1 << PI_PERF_/* YOUR CODE HERE*/ |  // count cycles
                    1 << PI_PERF_/* YOUR CODE HERE*/ ); // count instructions
    // reset the perf counters
    pi_perf_reset();
    //  start the perf counter
    pi_perf_start();
}
```

Put your solution below (code)

[HERE]

2.4. Implement missing code part IV

GOAL: we now are able to measure Cycles & instructions (thanks to performance counters), and MAC operations (your implementation in the code)

TASK: calculate and print the following metrics in the code (fill in where you read /*YOUR CODE HERE*/)

- CPI = cycles/instructions
- MAC/cycles
- Instructions/Cycles
- Instructions/MACs

```
/*
    TASK 2.2
    Measure:
        - How many MAC operations are needed for the gemv
    Calculate:
        - CPI
        - MACs/Cycles
        - Instructions/Cycles
        - Instructions/ MACs
*/

// N° Multiply Accumulate Operations (MACs)
/* already done with task 2.1. use the mac_counter global variable */
// CPI = cycles / n°instructions_executed
float cpi = /* YOUR CODE HERE*/
// MAC/Cycles
float mac_on_cycles = /* YOUR CODE HERE*/
// Instructions/Cycles
float instructions_on_cycles = /* YOUR CODE HERE*/
// Instructions/MAC
float instructions_on_mac = /* YOUR CODE HERE*/
```

Put your solution below (code)

[HERE]

2. The size of the matrix is NxM=50x50 and the vector is M=50. Run the code and fill the table:

Note:

- Profile the matrix-vector product with different compiler optimizations: -O1, -O3, -O3 with HW Loops (default is -O1)

	-01	**-03 -mnohwloops**	**-03**
Cycles			
N°Instructions			
MACs			
CPI			
Instructions/Cycles			
Instructions/MACs			