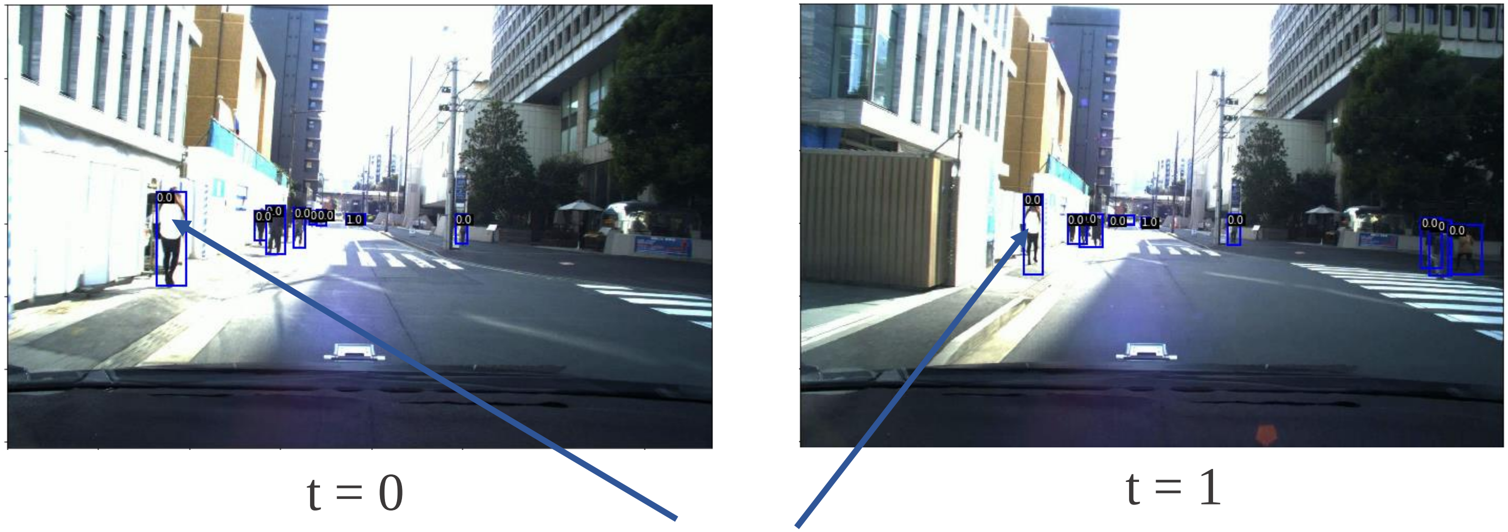
5th AI Edge Contest

Ninnart Fuengfusin (ninfueng)

March 24, 2022

Contest Overview

Tasks: Object Tracking



t = 0

t = 1

…

Same person?

Object Tracking ≈ Object Detection + Tracking.

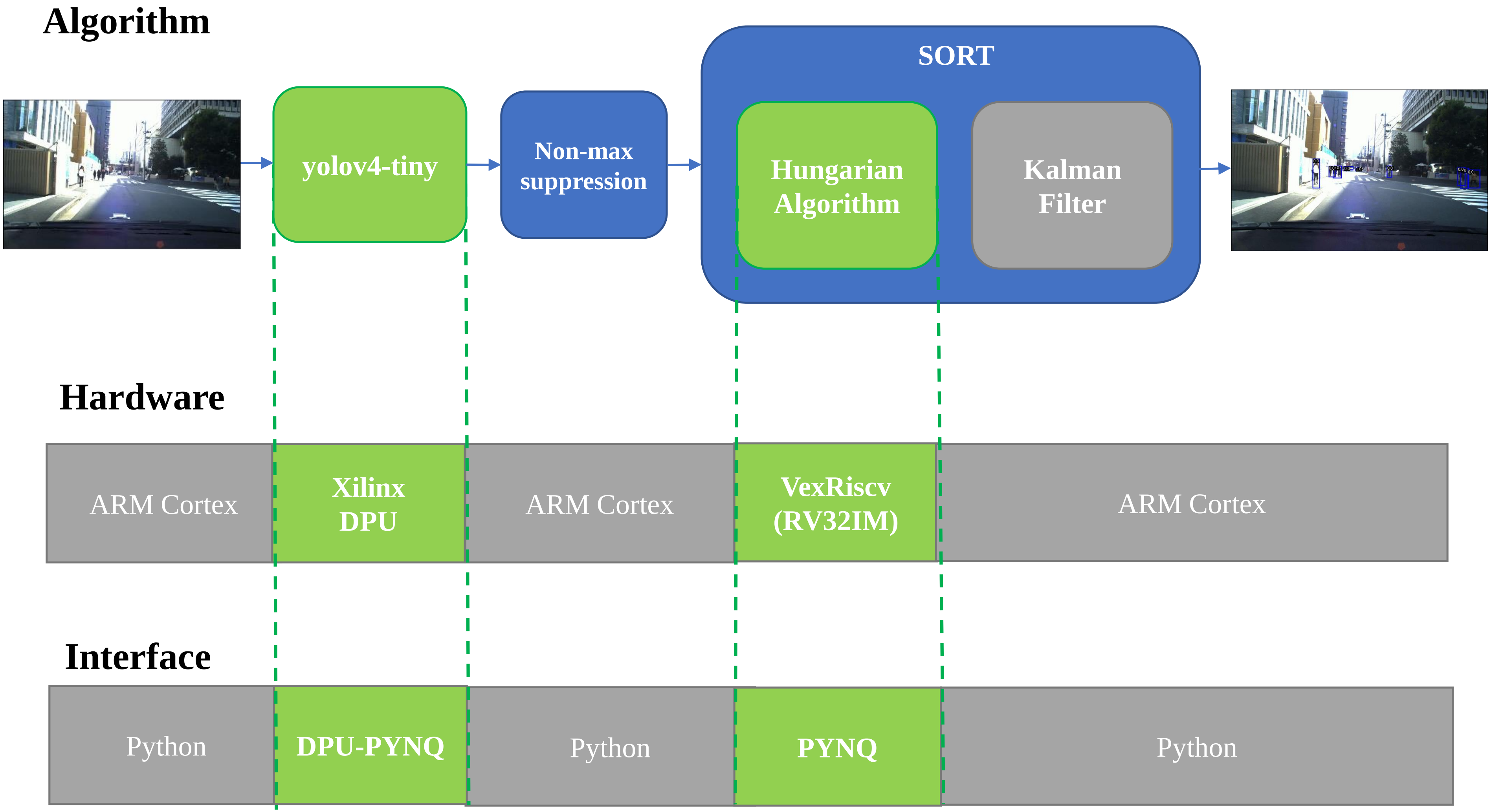
Chosen methods

- **yolov4-tiny**
- yolov4
- CenterNet
- SSD300
- …

- **SORT**
- DeepSORT
- ByteTrack
- …

Requirements: One of operations must done in RISCV (or control by RISCV)

Implementation Overview



Algorithm

yolov4-tiny → Non-max suppression → SORT (Hungarian Algorithm, Kalman Filter) →

Hardware

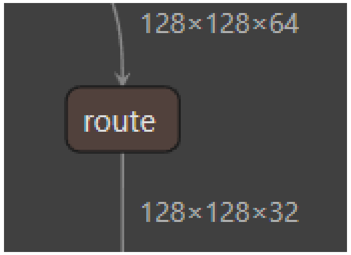
ARM Cortex	**Xilinx DPU**	ARM Cortex	**VexRiscv (RV32IM)**	ARM Cortex

Interface

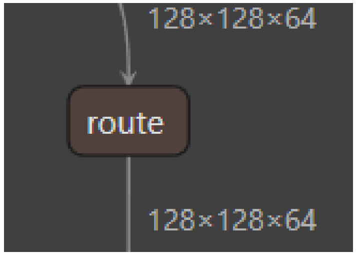
Python	**DPU-PYNQ**	Python	**PYNQ**	Python

Yolov4-tiny

- Utilized yolov4-tiny from AlexeyAB DarkNet[1].
- Input image size: 512x512.
- **Make yolov4-tiny operate-able with Vitis AI.**
  - New route (split and route) in yolo4-tiny -> route (same as identify function?).



Before



After

- **Anchor box optimization:**
  - Original anchors =
    - [10,14], [23,27], [37,58], [81,82], [135,169], [344,319]
  - Optimized anchors =
    - [5, 14], [9, 30], [27, 29], [16, 64], [49, 68], [100,117]

Smaller anchors: Better?

- **Quantization and compile with Vitis AI Tensorflow V1 1.4 flow.**
- **Control:** DPU-PYNQ.

[1]: https://github.com/AlexeyAB/darknet

SORT

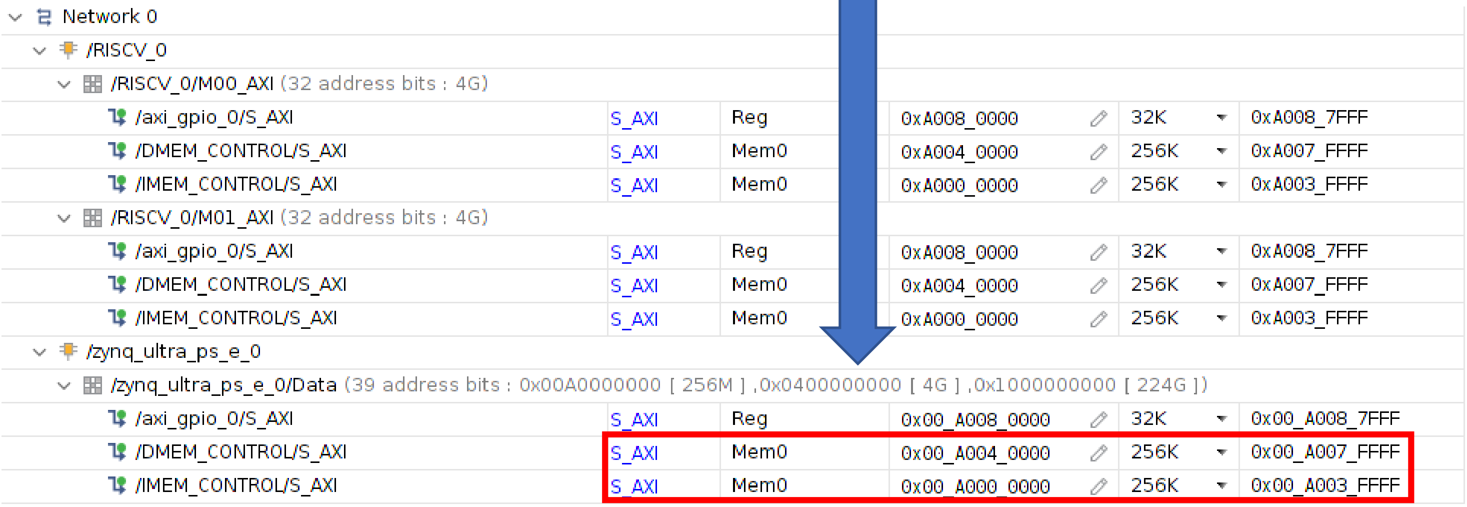
- **Consists: Hungarian Algorithm** and Kalman Filter
- **Hungarian Algorithm:** checks objects from the current frame are the same with the objects from previous frames or not.

- **Use Hungarian Algorithm with RISCV.**
  - **Using Hungarian algorithm with RISCV. Why?**
    - Small size of input matrix.
    - Possible to convert to integer only operations.
      - $cost_{new} = round(1 - cost) \times 1000$
      - Convert input cost matrix to **integer** and inverse the optimization direction (**min** -> **max**).
  - **Using C implementation[2].**
    - Minimum implementation of Hungarian algorithm (No standard libraries.)
    - Modify to access inputs from RISCV DMEM and produce outputs to RISCV DMEM.
    - All integer type.

[2]: https://github.com/mohammadusman/Hungarian-Algorithm-in-C-Language

RISCV Core

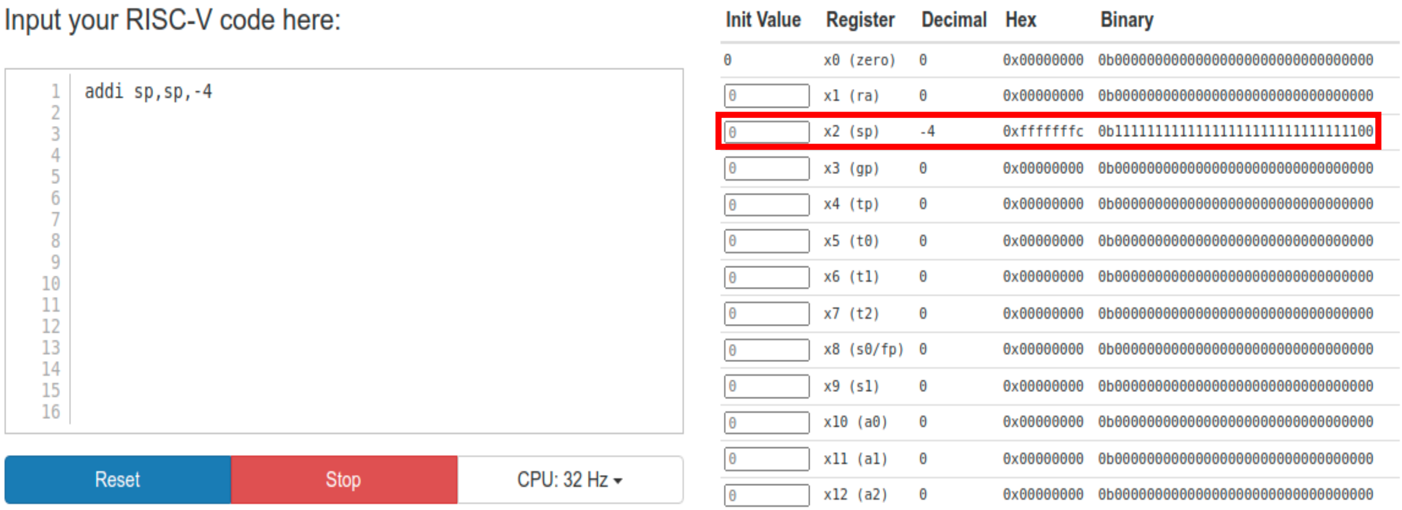
- **Almost same to SIGNATE RISCV core.**
  - Based on VexRiscv. Built with Xilinx Vivado 2020.2.
  - **RV32IM** (32-bit supports integer operations and multiplication).
  - Increase IMEM (instruction memory) and DMEM (data memory) size. (To able to input more instructions.)
- Hungarian C code -> assembly code, load assembly code to IMEM.
- Provides and produces inputs and outputs with DMEM.
- **Address of RISCV memory:** [A0000000, A007FFFF].
- **Control:** PYNQ



Network 0					
/RISCV_0					
/RISCV_0/M00_AXI (32 address bits : 4G)					
/axi_gpio_0/S_AXI	S_AXI	Reg	0xA008_0000	32K	0xA008_7FFF
/DMEM_CONTROL/S_AXI	S_AXI	Mem0	0xA004_0000	256K	0xA007_FFFF
/IMEM_CONTROL/S_AXI	S_AXI	Mem0	0xA000_0000	256K	0xA003_FFFF
/RISCV_0/M01_AXI (32 address bits : 4G)					
/axi_gpio_0/S_AXI	S_AXI	Reg	0xA008_0000	32K	0xA008_7FFF
/DMEM_CONTROL/S_AXI	S_AXI	Mem0	0xA004_0000	256K	0xA007_FFFF
/IMEM_CONTROL/S_AXI	S_AXI	Mem0	0xA000_0000	256K	0xA003_FFFF
/zynq_ultra_ps_e_0					
/zynq_ultra_ps_e_0/Data (39 address bits : 0x00A0000000 [256M] ,0x0400000000 [4G] ,0x1000000000 [224G])					
/axi_gpio_0/S_AXI	S_AXI	Reg	0x00_A008_0000	32K	0x00_A008_7FFF
/DMEM_CONTROL/S_AXI	S_AXI	Mem0	0x00_A004_0000	256K	0x00_A007_FFFF
/IMEM_CONTROL/S_AXI	S_AXI	Mem0	0x00_A000_0000	256K	0x00_A003_FFFF

Limitations

- **One of Limitations:** FPGA environment differs from PC environment.
- FPGA valid address: [A0000000, A007FFFF].
- **Allocate 32-bit into the stack pointer (***addi sp, sp, -4***)**



Input your RISC-V code here:

```
addi sp,sp,-4
```

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	0	0x00000000	0b00000000000000000000000000000000
0	x2 (sp)	-4	0xfffffffc	0b11111111111111111111111111111100
0	x3 (gp)	0	0x00000000	0b00000000000000000000000000000000
0	x4 (tp)	0	0x00000000	0b00000000000000000000000000000000
0	x5 (t0)	0	0x00000000	0b00000000000000000000000000000000
0	x6 (t1)	0	0x00000000	0b00000000000000000000000000000000
0	x7 (t2)	0	0x00000000	0b00000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0b00000000000000000000000000000000
0	x12 (a2)	0	0x00000000	0b00000000000000000000000000000000

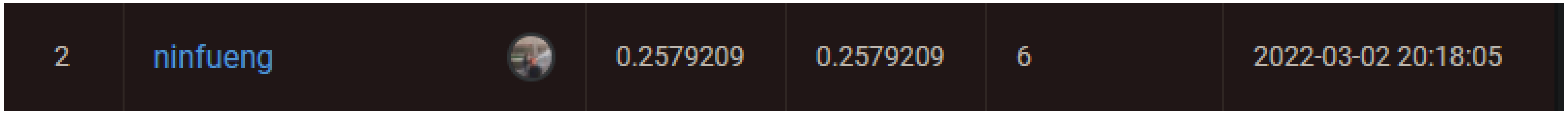
From: https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/

- **Solve:** Insert *lui (load upper immediate) sp, sp, A0030* to initialize the stack pointer.

- **Another Limitation:** somehow this RISCV Hungarian assembly code **operates only with 2x2 cost matrix**.
- **Cannot solve in time:** using RISCV only inputs 2x2 cost matrix, otherwise using ARM.

Results

- **Test video:** 74 videos with size of 1216, 1936.
- **Total:** 11,100 frames (each of test video: 150 frames).
- **Total runtime:** 31 minutes 38 seconds
- **Frames per second: 5.85**, **MOTA: 0.2579**



2	ninfueng	0.2579209	0.2579209	6	2022-03-02 20:18:05

- With a limitation from RISCV assembly, use RISCV to process only 123 frames (with 2x2 cost matrix).
- Did not see the difference in term of MOTA from with or without RISC-V Hungarian algorithm.

Thank you for your attentions

Any questions or comments?