Posture Recognition Based on Deep Learning

1. Description

To help machines learn what we human beings are doing via a camera is important. Once it comes true, machines can make different responses to all kinds of human's postures. But the process is very difficult as well, because usually it is very slow and power-consuming, and requires a very large memory space. Here we focus on real-time posture recognition, and try to make the machine "know" what posture we make. The posture recognition system is consisted of DE10-Nano SoC FPGA Kit, a camera, and an HDMI monitor. SoC FPGA captures video streams from the camera, recognizes human postures with a CNN model, and finally shows the original video and classification result (standing, walking, waving, etc.) via HDMI interface.

2. High-level Project Description

2.1 Introduction

It is very important to teach the machines to recognize human postures, using computer vision technologies, because it is much more convenient compared with those using extra devices (gravity accelerators, gyroscope and so on). So, which algorithm or model can we choose to realize posture recognition?

Convolution Neural Networks (CNNs) are used in many fields related to images and videos, such as image recognition, object classification and target tracking. However, the computations of CNNs can be so complicated that CNNs on ordinary CPUs are very slow, power-consuming and requires too much memory space. And it becomes difficult to realize CNNs on embedded devices.

Usually CNNs are realized on GPUs, TPUs, FPGAs and so on. And with the development of High-Level-Synthesis (HLS) and OpenCL, FPGAs are more and more used to accelerate the computing of CNNs.

2.2 Proposal of the Design

Here we propose a posture recognition system based on DE10-Nano SoC FPGA Kit, with both FPGA part and HPS part on it. The FPGA part captures video streams from the camera, and computes ***L.K. optical flow*** at the same time. Meanwhile, we use ***HOG + SVM*** to detect pedestrians in the scene. All video, optical flow and pedestrian detection results are stored into DDR3, through AXI-Bridge or FPGA-to-SDRAM interface. Moreover, we design and realize an ***ISA-NPU***, which can be used to realize 2-D convolution, 2-D pooling, matrix addition and multiplication, and non-linear functions, e.g. sigmoid, tanh, relu, etc. We realize the process of ***recognizing posture based on CNNs***. And the result, again, is stored into DDR3 SDRAM. Finally, we can see the original video and classification result (standing, walking, waving, etc.), and how the system performs via HDMI video.

2.3 Application Scope

In consideration of camera's performance, the proposed posture recognition system requires a bright environment. Once the machines learn how to recognize users' postures, the classification results can be used in many applications, such as remotely controlling a model car, motion sensing

games, etc.

2.4 Targeted Users

The proposed posture recognition system is targeted to those who are wild about interesting technologies, motion sensing games and so on.

3. Block Diagram

3.1 Introduction

Posture recognition is a difficult task, since the machine should ***consider several continuous frames in the video***, and capture the object or the human correctly, and recognize whether the object is static or dynamic.

Generally, action recognition consists of two main problems. Firstly, the machine should find out where human is; and secondly, what action or posture the human make.

In the "foreground extraction" field, there is a so-called "optical flow" method, which can be used to measure the speed of the object, works well in finding out a moving human, but not enough to find out a person in the static picture; And in the pedestrian detection field, HOG + SVM together turn out to be a strong algorithm, but the accuracy is not good.

And in last years, where are several important methods to recognize human action in video, such as iDT (improved dense trajectory), 3-D convolution network, two-stream convolution network, CNN & RNN / LSTM, and so on.

So, here we plan to recognize the posture in the following way, as shown in Fig. 1:

1. Detect pedestrian in a static picture using ***HOG +SVM***, and we get "a static window";
2. Measure the speed of the body using ***LK optical flow method***, and we get "a dynamic window";
3. Combine the pedestrian detection and optical flow detection results, to find out the zone where human is, which can be called ***merging static and dynamic windows***;
4. Use the ***two-stream CNN model*** to recognize the posture with the static picture & optical flow information.

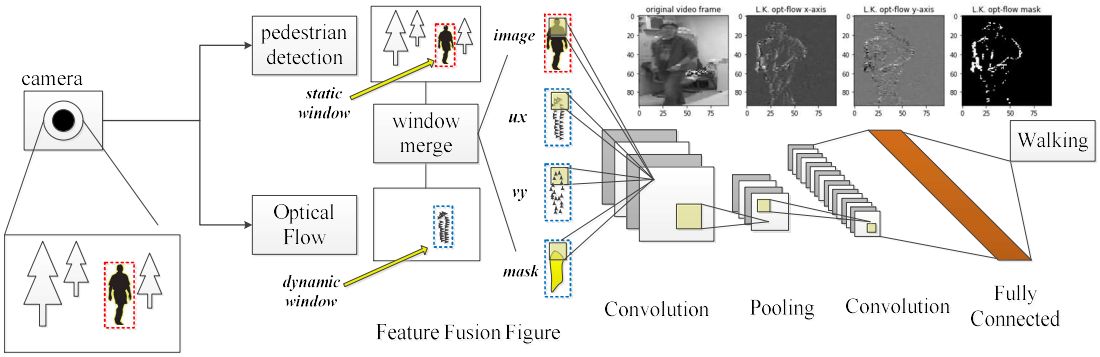


Fig. 1. The total process of posture recognition in our project

3.2 Framework

We train the CNN model in the offline way, and it means that we should sample many labeled data. After training, we can make real-time inference. The posture recognition system is consisted of a camera (MT9D111 in our design), the DE10-Nano SoC FPGA Kit, and an HDMI monitor, as shown in Fig. 2.

Firstly, the video stream is transferred into posture recognition module, where there is an LK optical flow module and a pedestrian detection module. All the video and computation results are stored into DDR3 on the HPS side.

Secondly, HPS reads optical flow and pedestrian detection results from DDR3, using mmap() function in Linux OS. After merging the dynamic and static windows, the program on the HPS finds out the zone where human is. Moreover, HPS should generate the NPU instructions, and store weights and biases into DDR, so that a CNN can be realized on NPU. And then HPS commands the ISA-NPU on FPGA-logic to recognition the posture.

Finally, the posture label and the original video are mixed in the image fusion module, and are shown together via the HDMI interface.

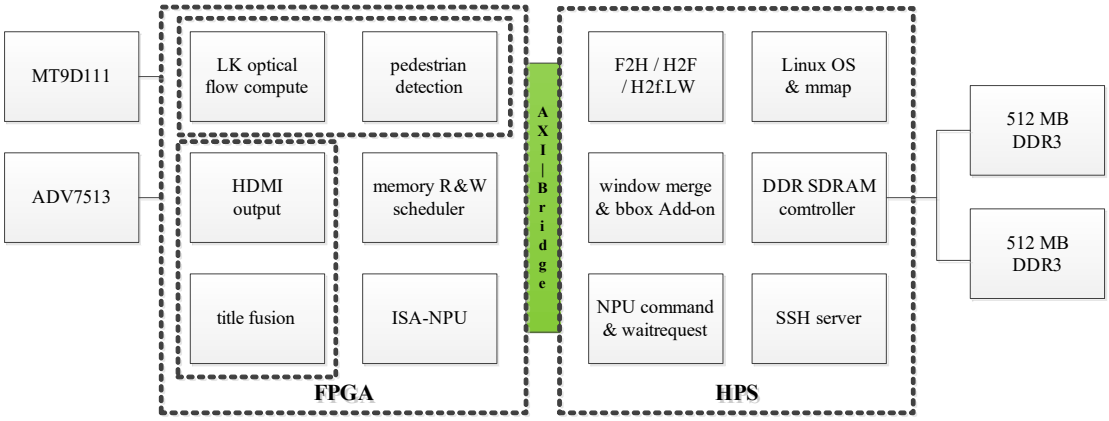


Fig. 2. The framework of our posture recognition system

3.3 Connect Camera and FPGA Kit

Since we focus on posture recognition algorithm and its realization on SoC FPGAs, we choose an MT9D111 camera. We choose DE10-Nano FPGA Kit as our platform. But the problem occurs: MT9D111 camera cannot be directly connected to GPIOs on DE10-Nano Kit. We design a pin-board to ***connect MT9D111 and DE10-Nano***. Following is the posture recognition platform.

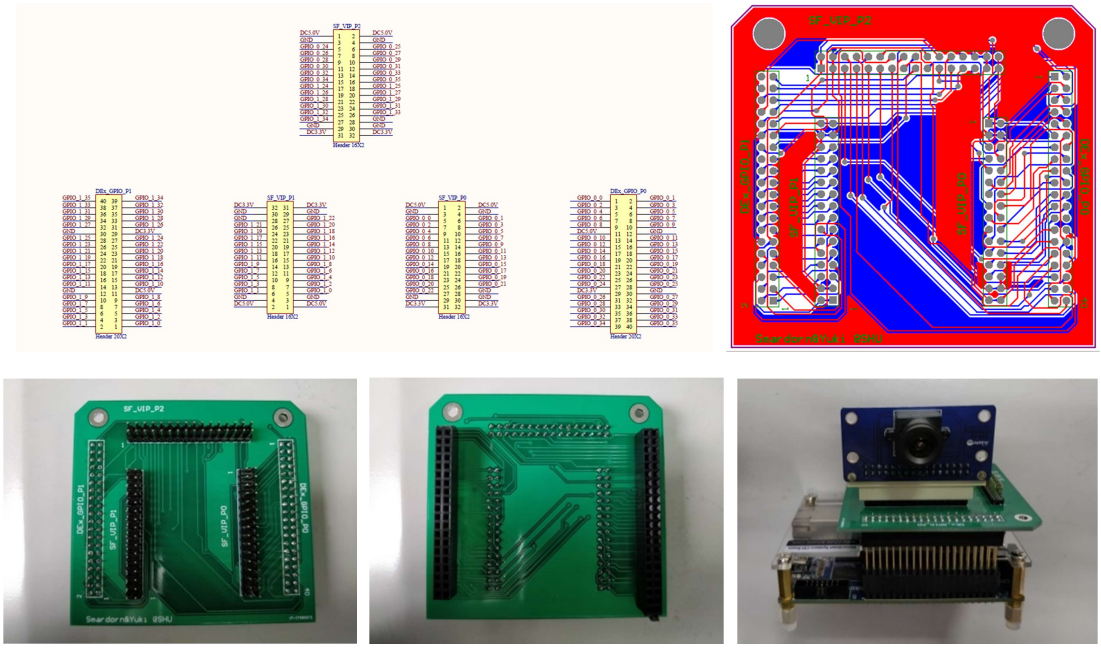


Fig. 3. The pin-board and the final posture recognition system

4. Intel FPGA Virtues in Our Project

4.1 Hardware

There are both FPGA logic and ARM processor on the SoC FPGA chip. And the resources on DE10-Nano SoC FPGA Kit are following:

4.1.1 On FPGA side

1. 5CSEBA6U23I7 device (110K LEs)
2. EPCS64 Serial configuration device
3. HDMI TX compatible with DVI 1.0 and HDCP v1.4

4.1.2 On HPS side

1. 800MHz Dual-core ARM Cortex-A9 processor
2. 1GB DDR3 SDRAM (32-bit data bus)
3. 1 Gigabit Ethernet PHY with RJ45 connector
4. Micro SD card socket
5. UART to USB, USB Mini-B connector

4.1.3 AXI-Bridge

On SoC FPGA there are three AXI-Bridges to make the communication between FPGA and ARM much more convenient and faster: HPS2FPGA, FPGA2HPS and Light-Weighted HPS2FPGA.

4.2 Software

The most interesting point is ***mmap()*** on Linux OS. Using mmap(), HPS can access external devices, such as DDR3, HPS-to-FPGA interface, etc. And thus we can transfer NPU commands via HPS-to-FPGA interface, and access FPGA status at the same time.

4.3 Conclusion

Using DE10-Nano SoC FPGA Kit, the proposed posture recognition system can be ***realized within a tiny board*** with a camera. Moreover, ***the recognition process can be accelerated and optimized*** using both HPS and ISA-NPU. Finally, the proposed system is ***connected to Ethernet***, making it convenient to monitor.

5. Design Introduction

The process of recognizing human posture can be separated into two steps. Firstly, the algorithm should find out where human is; and secondly, the algorithm should find out what posture human makes.

5.1 Pedestrian Detection

For the first problem, there are two main ways to find human in a video stream, static image analysis (HOG+SVM for example) and dynamic video analysis (Optical Flow for example). Both methods work well in specific scenes, but have their own deficiencies meanwhile.

Tbl. 1. Comparison of static and dynamic method of pedextrian detection

Method	**HOG + SVM**	**Optical Flow**
Static / Dynamic	static	dynamic
Goal	find out specific figures	find out moving things
Advantages	independent on pedestrian's or camera's motion	low FP rate
Disadvantages	high FP rate	1. suffer from camera's motion 2. not useful once pedestrian doesn't move

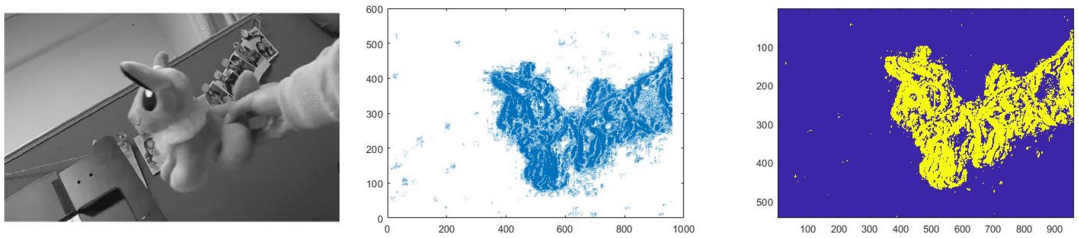


Fig. 4. The dynamic detection based on L.K. optical flow.
Once Yibu and the hand moves, the algorithm can detect.

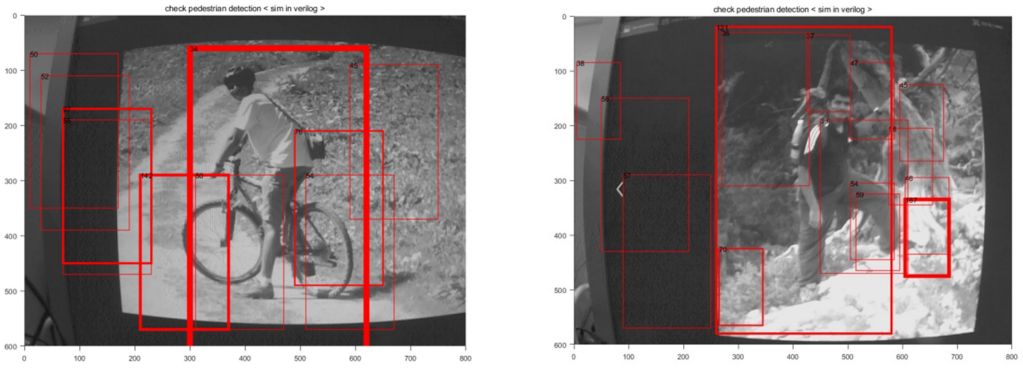


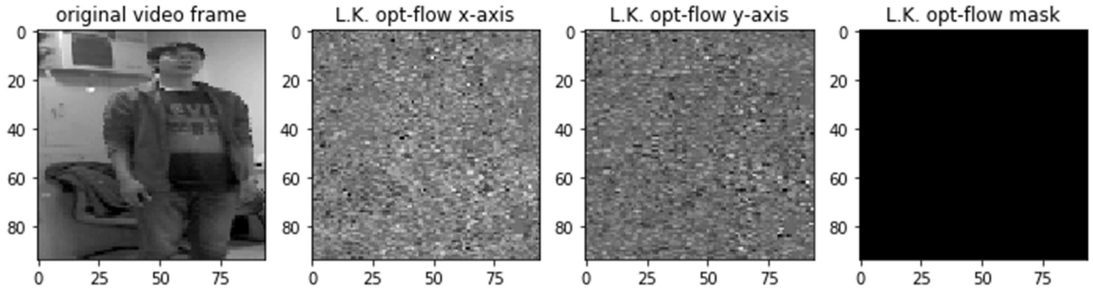
Fig. 5. The static pedestrian detection based on HOG + SVM
And we can see, there are many false-positive detection results.

So the best way to detect pedestrians in the video stream is to realize both static and dynamic method. When the pedestrian moves, the dynamic method based on optical flow can help to find out the moving part; and when the pedestrian does not move, the static method such as HOG + SVM can help to find the pedestrian. ***Moreover, once optical flow and HOG + SVM are computed, the results of both methods should be merged.***

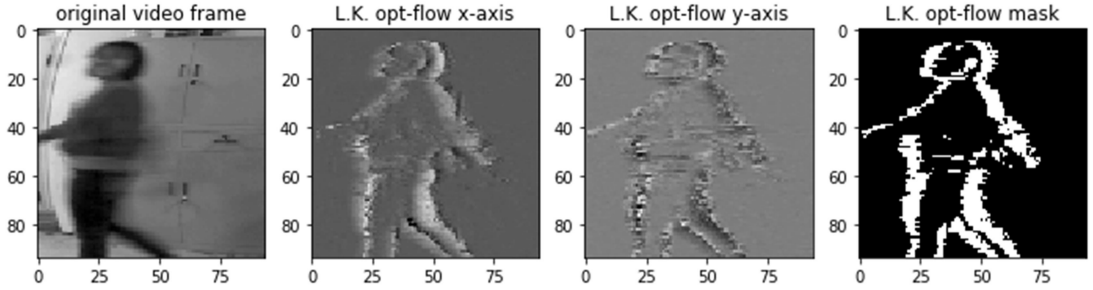
P.S. We can also use CNN to find person in the image (e.g. LeNet, VGG-16, YOLO). But the problem occurs that the computation is too complex and time-consuming. And compared to CNN, optical flow and HOG+SVM can be both parallelized on FPGA. So we decide to reserve NPU, which can be used to compute CNN model, to posture recognition.

5.2 Posture Recognition

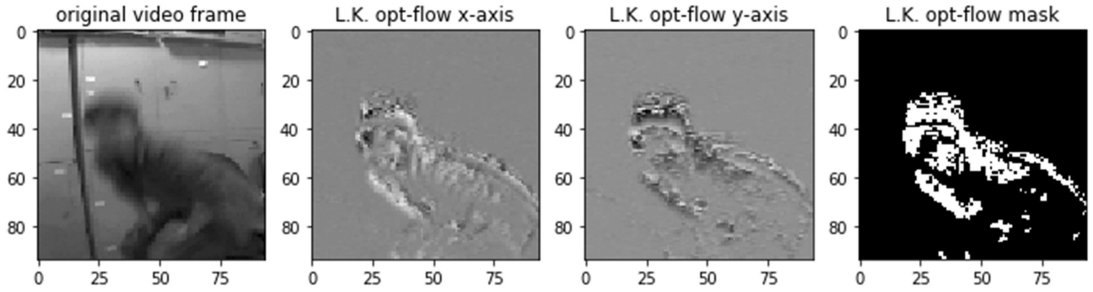
Once we find where human is, we can get the gray image and optical flow inside the zone. Following are images and flows for different postures. As is shown, optical flows can be decomposed into x-axis and y-axis, and the optical flow mask occurs "1" for the area where the motion is severe and "0" otherwise.



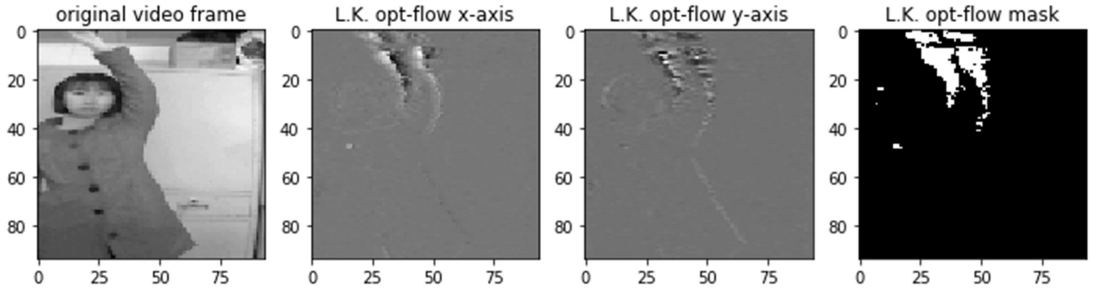
(a)



(b)



(c)



(d)

Fig. 6. Images and flows for different postures.
(a) standing samples; (b) walking samples;
(c) squatting samples; (d) waving samples;

We can see, the image from original video frame consists of different objects in the scene, and

meanwhile, L.K. optical flow consists of motion information. So both of them are important for posture recognition, and indeed, our two-stream-CNN model ***takes image, x-axis motion, y-axis motion and flow mask as its input channels***, and infers what posture human takes.

5.3 Device Choice

Here we propose a posture recognition system based on DE10-Nano SoC FPGA Kit, with both FPGA part and HPS part on it. And the interfaces between the two parts are based on AXI-bridge, which can be translated into Avalon-Bus using Qsys IP cores.

The FPGA part can be used to configure the camera, and capture video streams, and computes ***L.K. optical flow*** and ***HOG + SVM*** to detect pedestrians in the scene. All video, optical flow and pedestrian detection results are stored into DDR3, through ***FPGA-to-SDRAM interface***. Moreover, an ***ISA-NPU*** that realizes 2-D convolution / pooling, matrix addition / multiplication, and non-linear functions are designed. And to ***output HDMI stream***, FPGA should also configure ADV7513 and generate specific interface timing as well.

The HPS part is used to read video / computation results, and configure CNN weights / biases inside DDR, using mmap() function. And ***transmit NPU instructions***, and ***monitor the computation process*** via ***HPS-to-FPGA interfaces***. Moreover, users can use UART or SSH to login Linux OS.

Considering that the posture recognition system consists of both FPGA logic and C program, an SoC FPGA platform is suitable for our project.

6. Function Description

The posture recognition system proposed consists of two main parts. Firstly, the hardware architecture, which is mainly realized in FPGA, is used to configure camera and HDMI chip, and process video stream input / output, and detect pedestrian in the video. Secondly, the software program, which is mainly realized in C language, is used to merge results of optical flow and HOG + SVM, and control the ISA-NPU on FPGA side.

6.1 Hardware function description

The hardware part of our posture recognition system is mainly made up with camera/HDMI configure module, LK optical flow module, HOG & SVM pedestrian detection module, video store & display module, ISA-NPU module, and R & W scheduler.

Once the camera and HDMI chip are both configured, we can capture video stream from the camera, and compute LK optical flow and HOG + SVM pedestrian detection at the same time. And the results are stored into DDR through F2S interface. Moreover, the C program on HPS side will control the neural process unit (NPU) to compute two-stream-CNN. And the result, which shows what class of posture is recognized, can be used to output subtitle.

And the DDR on DE10-Nano Kit is 1 GB, and it is divided into 6 parts. 0 ~ 480 MB is used for Linux OS, and we can set the space OS uses in u-boot; 480 MB ~ 512 MB is used to save four continuous frames of optical flow; and 512 MB ~ 544 MB is used to save video stream; 576 MB ~ 608 MB is used for static pedestrian detection result; and 608 MB ~ 640 MB is used for HPS to add boxes on video, and then the video will be displayed on monitor; and 640 MB ~ 1024 MB is used for ISA-NPU to compute CNN.

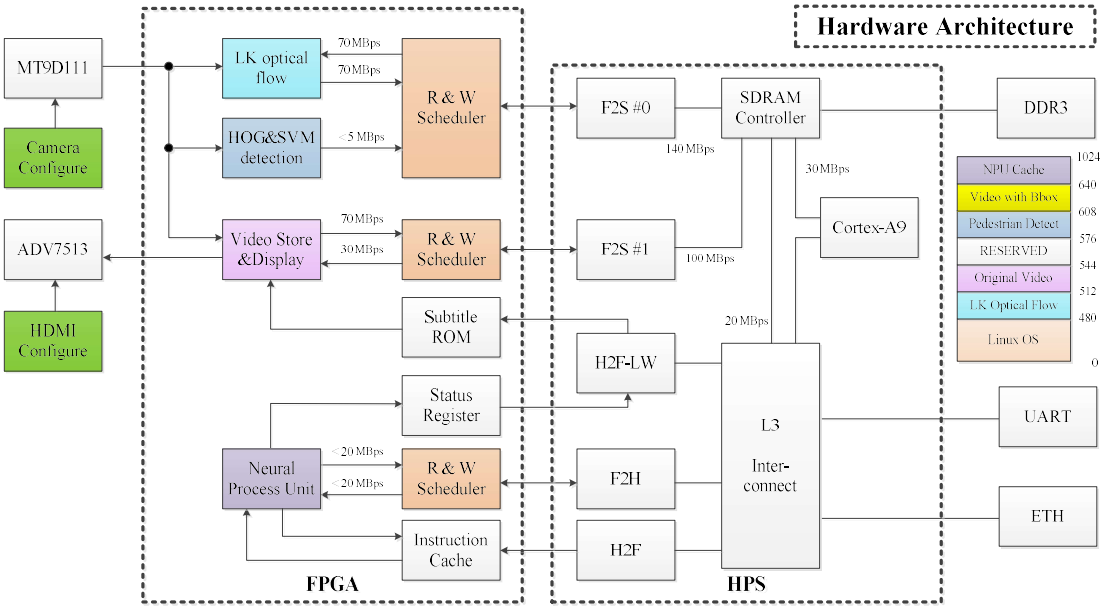


Fig. 7. Hardware Architecture

6.1.1 Configure MT9D111

Before receiving pixel stream from the camera, we need to ***configure MT9D111 via I2C interface***. And firstly we ***generate an MCLK at 25 MHz to MT9D111 (on its pin CLKIN)*** as its operating clock. We configure the camera to output video at the resolution of ***800x600 @ 15 fps, in RGB565 format***.

Registers in MT9D111 are separated into 3 pages: Page #0 for sensor core registers, Page #1 and Page #2 are for IFP registers. The I2C interface timing diagram can be described as following:

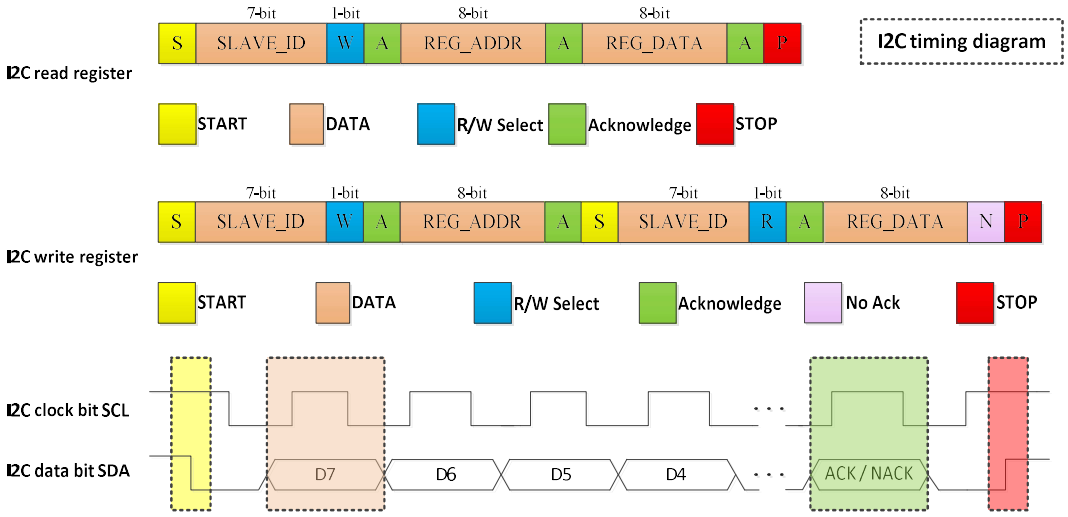


Fig. 8. I2C timing diagram

Moreover, we can describe a reading/writing operation with MT9D111 as { R/W, Page, RegAddr, RegData }, in which R/W is READ/WRITE selection, Page is 16-bit to choose which page the register is inside, RegAddr is 8-bit as the address of register, and RegData is 16-bit as the data to write into the register, and for reading, RegData==0x0000.

According to the MT9D111 datasheet, ***the I2C interface of MT9D111 supports both 16-bit and 8-bit mode***. Considering ***the portability of I2C Verilog HDL code*** (cause many sensors use 8-bit I2C, MPU6050, etc), we choose 8-bit mode. And the total operation goes as following:

```
Operation RW_MT9D111_REG (R/W, Page, RegAddr, RegData)
    i2c_write_8b( SlaveAddr, 0xF0, Page>>8)          // Page High 8b into Register 0xF0
    i2c_write_8b( SlaveAddr, 0xF1, Page&0xFF)        // Page Low 8b into Register 0xF0
    if READ :
        i2c_read_8b(SlaveAddr, RegAddr, NULL);  // Read Register RegAddr, High 8b
        i2c_read_8b(SlaveAddr, 0xF1, NULL);       // Read Register RegAddr, Low 8b
    else if WRITE:
        i2c_write_8b(SlaveAddr, RegAddr, RegData>>8);      // Write RegAddr, High 8b
        i2c_ write _8b(SlaveAddr, 0xF1, RegData&0xFF);     // Write RegAddr, Low 8b
```

6.1.2 Configure ADV7513

The HDMI chip ADV7513 on the DE10-Nano board is ADV7513 from ADI. Again, we configure the HDMI chip via I2C interface. And the pixel clock is 26 MHz, and the resolution of *HDMI output is 1024 x 768 @ 24 fps*.

6.1.3 LK Optical Flow

Optical Flow can be used to describe objects' motion in a video stream. We define $I(x, y, t)$ as the gray value of coordination (x, y) at the time t. And after a small interval Δt, the object moves from (x, y) at time t to $(x+\Delta x, y+\Delta y)$ at time $t+\Delta t$.

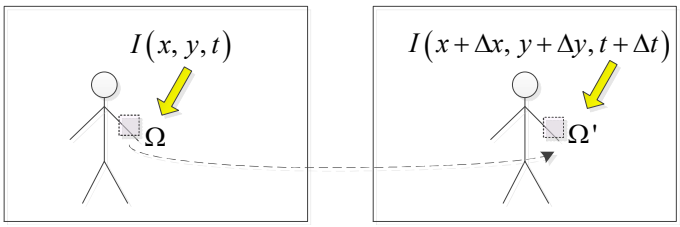


Fig. 9. optical flow

And we have the following assumption, that $I(x, y, t) = I(x+\Delta x, y+\Delta y, t+\Delta t)$, which can be transformed into following format: $\frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t = 0$. Considering the speed of the object can be computed as $\begin{cases} u = \partial x / \partial t \\ v = \partial y / \partial t \end{cases}$, we have the final optical flow equation:

$$I_x u + I_y v + I_t = 0$$

Usually, the equation cannot be solved. So, Lucas and Kanade proposed following optimization problem to solve the velocity field.

$$u^*, v^* = \arg\min_{u,v} \sum_{\Omega} \left(I_x u + I_y v + I_t\right)^2 + \lambda\left(u^2 + v^2\right)$$

where Ω is a very small area, where the optical flow field is consistent.

And the optimal solution of the problem can be expressed in the following way:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \left(A + \lambda I\right)^{-1} B$$

where $A = \begin{bmatrix} \sum_{\Omega} I_x^2 & \sum_{\Omega} I_x I_y \\ \sum_{\Omega} I_x I_y & \sum_{\Omega} I_y^2 \end{bmatrix}$, $B = -\begin{bmatrix} \sum_{\Omega} I_x I_t \\ \sum_{\Omega} I_y I_t \end{bmatrix}$, and parameter λ can be considered as a

regularization item. And the equation can be transformed into:

$$\begin{bmatrix} u & v \end{bmatrix} = \begin{bmatrix} \dfrac{(d+\lambda)e - bf}{(a+\lambda)(d+\lambda) - bc} & \dfrac{(a+\lambda)f - ce}{(a+\lambda)(d+\lambda) - bc} \end{bmatrix}$$

where $a = \sum_{\Omega} I_x^2, b = \sum_{\Omega} I_x I_y, c = \sum_{\Omega} I_x I_y, d = \sum_{\Omega} I_y^2, e = \sum_{\Omega} I_x I_t, f = \sum_{\Omega} I_y I_t$

In our posture recognition system , the Ω field is set to 5x5 pixel area, and regularization item $\lambda = 0.1$.

Once we learn how LK optical flow is computed, we can realize the module in Verilog HDL, as is shown in the Fig. 10. Thanks to the computation progress of LK optical flow, it is very suitable to be realized in parallel computation, in hardware circuits.

In DDR, we allocate 8 MB for each frame in the video stream. And when current frame arrives, the previous frame in DDR should be read out at the same time. So we use a FIFO to cache the pixels *I(x, y, t-1)* from last frame. And for current pixels *I(x, y, t)*, we use a line-shifter to generate *I(x, y-1, t)*, and a pixel shifter to generate *I(x-1, y, t)*. All these pixels are transformed to YUV field from RGB field. Then we can get gradient values I_x, I_y, I_t. The gradient values are then transferred into shifter-registers which generate Ω field, where we compute values of a ~ f in the equation, and finally we use pipelined divider module to generate optical flow, which are stored into DDR.

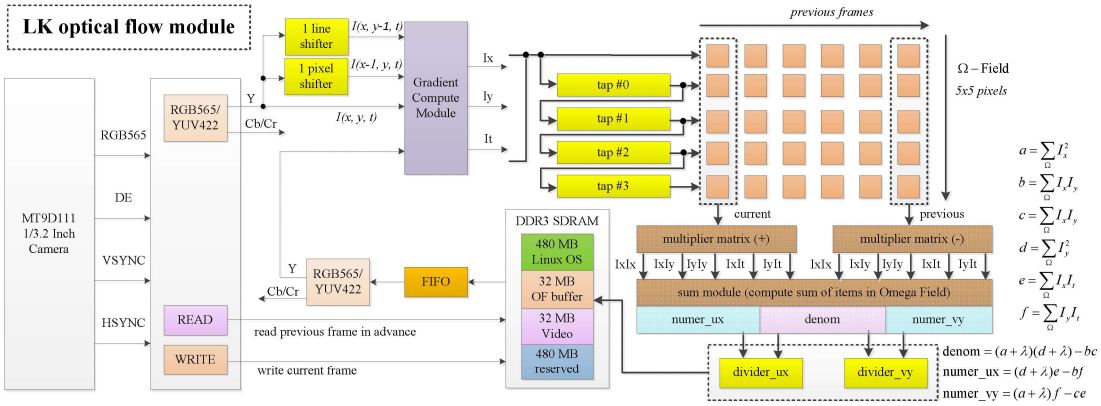


Fig. 10. LK optical flow module architecture

6.1.4 Static Detection Using HOG + SVM

Pedestrian detection is an important problem in our posture recognition system. Usually, we extract HOG (Histogram of Oriented Gradient) feature of a sliding window, and classify whether there is a person in the window using SVM (Support Vector Machine).

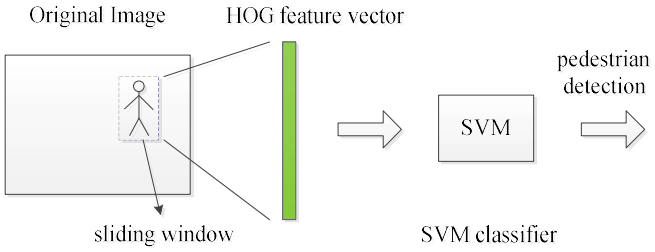
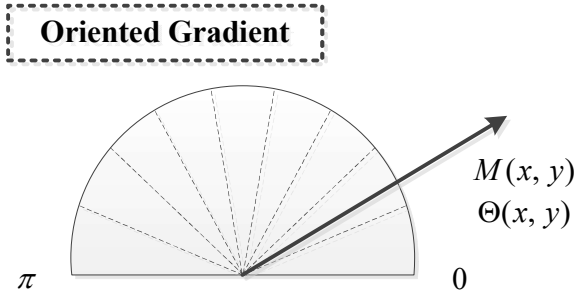


Fig. 11. How HOG+SVM works in pedestrian detection

To demonstrate how HOG feature is extracted, we assume the pixel value on (x, y) is $I(x, y)$. And we can compute gradient on the point as $I_x(x, y)$ and $I_y(x, y)$, which are the

difference of adjacent points. And additionally, we can compute the magnitude and orientation of gradient as $M(x,y)$ and $\Theta(x,y)$. As the name implies, if we need HOG feature, the first step is to compute histogram of gradient's orientation. We need to separate the gradient into N intervals, and each interval's length is π/N. For example, if $N=9$ and we have a gradient whose orientation is $\Theta=\pi/6$, then the gradient would be arranged into $B(x,y)=1$.



$$I_x(x,y) \approx I(x+1,y) - I(x-1,y)$$
$$I_y(x,y) \approx I(x,y+1) - I(x,y-1)$$
$$M(x,y) = \sqrt{I_x(x,y)^2 + I_y(x,y)^2}$$
$$\Theta(x,y) = \tan^{-1}(I_x(x,y)/I_y(x,y)) \,\%\, \pi$$
$$B(x,y) = \mathrm{floor}\left(\Theta(x,y)/\pi \times N\right)$$

Fig. 12. How magnitude and orientation of gradient are computed

Then we demonstrate how HOG feature is extracted with magnitude and orientation of gradient on each pixel point. We divide the sliding window, which is 140 x 80 in our project, into 14x8 cells, and each cell's size is 10 x 10. And we need so-called block structure as well, which is made up with 2x2 cells. And the total window feature consists of 13x7=91 block features, as the block slides. And the block feature consists of 2x2=4 cell feature, as the cell slides. So the total window feature can be expressed as a 91x4x9 = 3276 dimension vector.

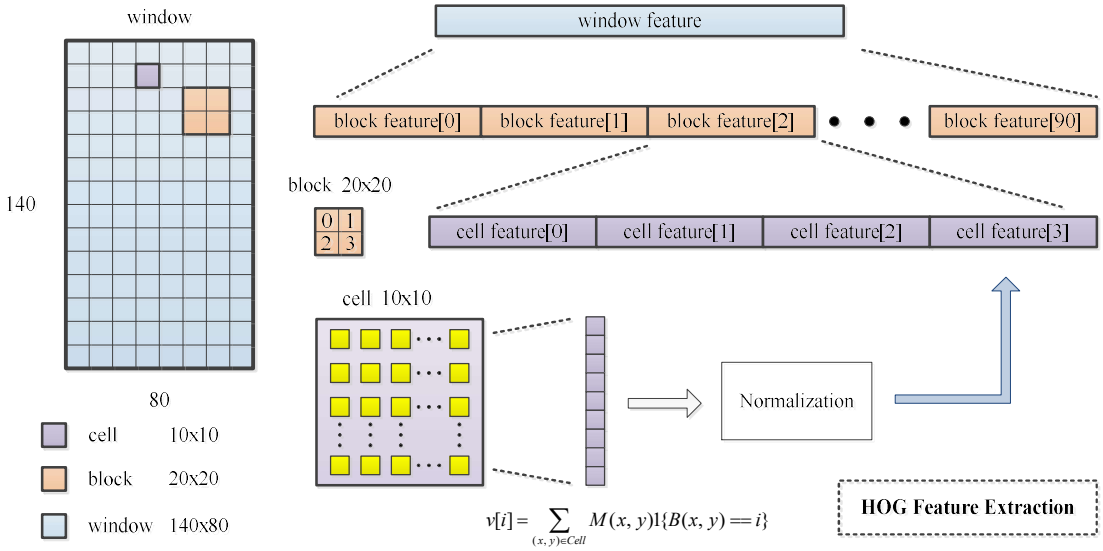


Fig. 13. How HOG feature is extracted

And now we compute histogram of orientated gradient in each cell. For k th cell, we compute the N-dimension vector $v^{(k)}$, and for each element $v^{(k)}[i]$ in the vector, we have:

$$v^{(k)}[i] = \sum_{(x,y)\in Cell(k)} M(x,y)1\{B(x,y) == i\}$$

Then we need to normalize the vector $v^{(k)}$, using L-2, L-1 or Binary strategy.

L-2 normalization: $v' = v/\sqrt{\|v\|^2 + \epsilon^2}$

L-1 normalization: $v' = v/\left(\|v\| + \epsilon\right)$

Binary normalization: $v' = \left(v > \mathrm{mean}(v)\right)$

In addition, we can use Fast-Square-Root method to complete L-2 normalization. But

considering the next computation complexity in SVM classification step, we choose ***Binary normalization method*** finally.

After the HOG feature extraction, we use a ***linear SVM*** to classify whether the object in the window is a person or not. And it can be demonstrated as following:

$$y = \mathrm{sign}\left(w^T x + b\right)$$

where w and b are the weights and biases of the SVM, and x is HOG feature in the sliding window, and the output $y=+1$ means it is a person, and $y=-1$ mean it is not.

We use INRIA pedestrian set to train our HOG+SVM model, and ***the ratio of positive and negative samples is 1:2 ~ 1:3***. Using binary normalization, the complex MAC in SVM can be reduced into simple addition operation, which is less resource-consuming.

Following Fig. 14 shows how our HOG feature extraction module works.

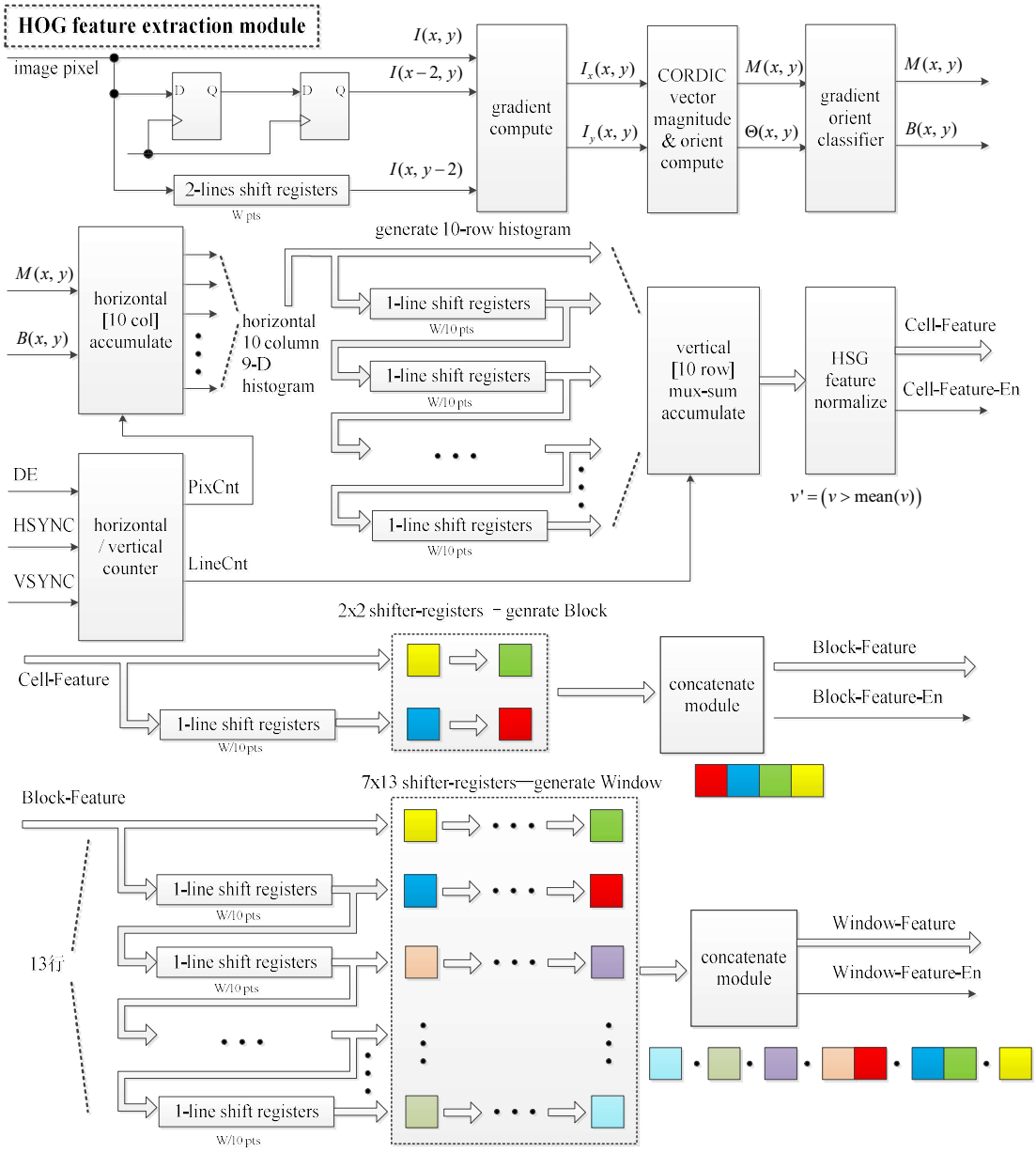


Fig. 14. HOG feature extraction module

Firstly, pixel data *I(x, y)* goes through 2 registers and one 2-line shifter-registers, and we get

I(x-2, y) and *I(x, y-2)*. Using gradient compute module, we can get gradient in both x-axis and y-axis, $I_x(x, y)$ and $I_y(x, y)$. And using CORDIC algorithm, we can get magnitude and orientation of gradient, *M(x, y)* and $\Theta(x, y)$. And then we classify orientation $\Theta(x, y)$, and get *B(x, y)*. And then *M(x, y)* and *B(x, y)* go into horizontal [10-column] accumulator, and count the histogram of oriented gradient for every 10-points, and we get a 9-D histogram. The 9-D histogram then goes into nine 1-line-shifter-registers, and we can accumulate these 9-D histograms in each orientation for 10-rows, and get the cell-feature without normalization. Using binary normalization method, we can compute *v' = (v > mean(v))*, which is the cell-feature. Then we can use shifter-registers to get block-feature and window-feature.

As we can see, the frequency of generating cell-feature, block-feature and window-feature depends on cell-size, which is 10 x 10 = 100. So, if we want the pedestrian detection module works in parallel, the computation of SVM should take less than 100 clocks. Here we use a SCFIFO (Single-Clock-FIFO) to cache vector under detection. And the FSM monitors whether the SCFIFO is empty or not. Once SCFIFO is not empty, the detection of pedestrian would be started, and the jumping-logic goes as following:

1. if a reset-n signal is asserted, jump to state 0;
2. when in state 0, if SCFIFO is not empty, generate read-request to SCFIFO, and jump to state 1;
3. when in state 1, read SVM-parameter ROM in each clock, and if rom_addr >= 91, which means the weights in SVM has already been read, jump to state 2;
4. When in state 2, delay for 5 clocks, which is for Adder-Tree to finish computation, and then jump to state 3;
5. when in state 3, delay for 1 clock, and jump to state 0.

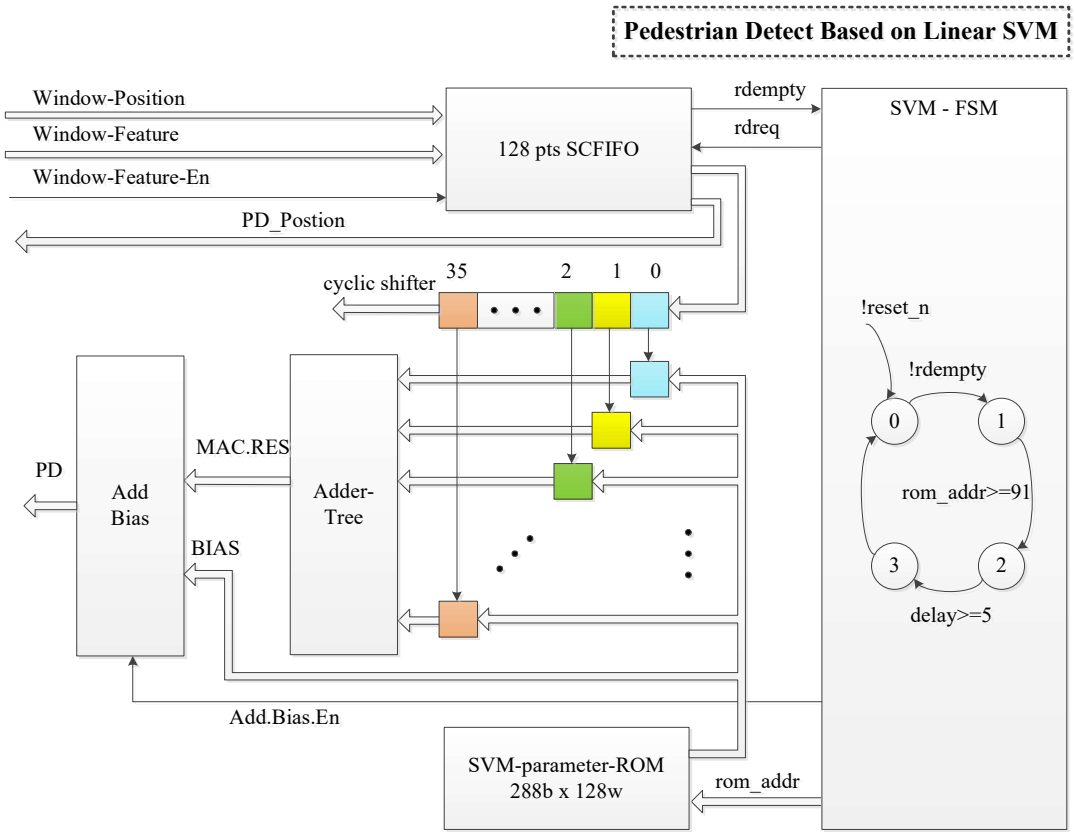


Fig. 15. Pedestrian Detection Based on Linear SVM

The window-feature read from the SCFIFO is 3276-D dimension. Using cyclic shift register,

in each clock, HSG feature has a 36-bit part for SVM weight gating; The ROM of the SVM parameter is configured to be 288 bits x 128 words, where the 288-bit data for each rom address can be separated into 36 8-bit data, and each data corresponds to one element of the weight in the SVM. There are 91 such 288-bit data and one offset. So rom_addr needs to be addressed to 91 in the FSM jump logic; Finally, 36 8-bit SVM weight parameters are selected (or gated to zero) per clock; and the selected 8-bit data (or zero) are then added, where the combinational logic is large and affects the timing, so 36 additions are split into 4 groups of 9 additions, achieving high data throughput through pipeline splitting.

6.1.5 Multi-Scale Detection for HOG+SVM

Apparently, the performance of HOG+SVM relies on the size of window, block and cell. And the most serious problem is that, if the person in the frame is so large (e.g. 280 x 160) that a 140 x 80 window cannot include it, then HOG+SVM model would fail. And we use a method called "Image Pyramid" to solve the problem. We can detect pedestrians in multi-scale of frame images.

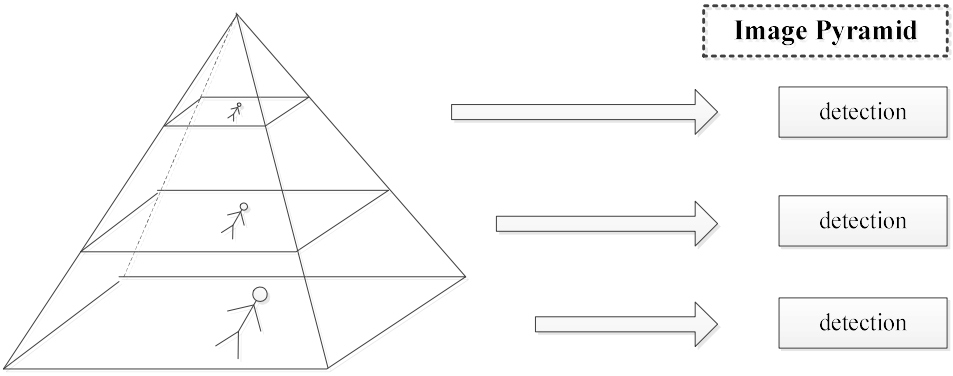


Fig. 15. image pyramid structure

In hardware design, we use 3 different scale to detect pedestrian, which are 1:1, 1:2, and 1:4, because we can use shifter-registers to cache one-line pixels, and use linear interpolation to generate half-resolution images.

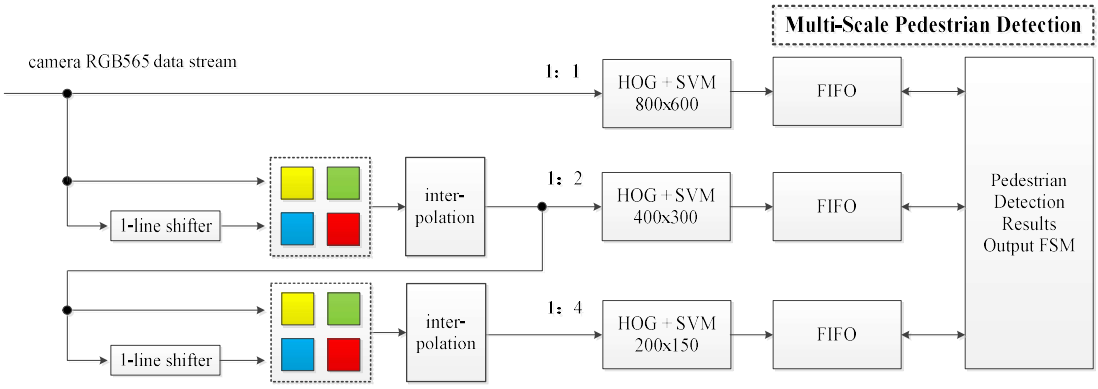


Fig. 15. Pedestrian Detection in multi-scale

6.1.6 ISA-NPU

In the computation progress of CNN model, the most time-consuming steps are convolution and non-linear function such as sigmoid, tanh and so on. Although there is one NEON float-point arithmetic core in Cortex-A9, it is still considerable to do inference in fixed-point operations, instead of float-point arithmetic. So we design an ISA-NPU in FPGA logic, in which simple instructions such as CONV, POOL, ADD, SIGM, TANH are all implemented, to save time for

ARM A9 to do something more suitable for CPU, instead of arithmetic operations.

We learn from RISC's experience, and design 128-bit instructions in the following format. Bits [127:124] represents operation number; bits [123:92] represents the address of input vector or matrix $1; bits [91:60] represents the address of input vector or matrix $2, or immediate number IMM; bits [59:28] represent the address of output vector or matrix $3; and bits [27:0] are flexible parameters.

Tbl. 3. NPU instruction format

127:124	123:92	91:60	59:28	27:0
OP	input $1	input $2 / IMM	output $3	parameter

We implement the following instructions. In the chart, $1 represents input parameter 1; $2 represents input parameter 2; $3 represents output matrix / vector 3; IMM represents immediate number; "x" means matrix multiplier; ".x" means pointwise multiplier of matrix; "*" means convolution.

Tbl. 4. NPU instruction chart

symbol	function	127:124	123:92	91:60	59:28	27:0
ADD	$3=$1+$2	4'B0000	$1 addr	$2 addr	$3 addr	[27:20] is row size; [19:12] is column size; [11:0] is reserved
ADDi	$3=$1+IMM	4'B0001	$1 addr	IMM	$3 addr	[27:20] is row size; [19:12] is column size; [11:0] is reserved
SUB	$3=$1-$2	4'B0010	$1 addr	$2 addr	$3 addr	[27:20] is row size; [19:12] is column size; [11:0] is reserved
SUBi	$3=$1-IMM	4'B0011	$1 addr	IMM	$3 addr	[27:20] is row size; [19:12] is column size; [11:0] is reserved
MULT	$3=$1x$2	4'B0100	$1 addr	$2 addr	$3 addr	[27:20] is row of $1; [19:12] is col of $1; [11:4] is col of $2; [3:0] is reserved
MULTi	$3=$1xIMM	4'B0101	$1 addr	IMM	$3 addr	[27:20] is row size; [19:12] is column size; [11:0] is reserved
DOT	$3=$1.x$2	4'B0110	$1 addr	$2 addr	$3 addr	[27:20] is row size; [19:12] is column size; [11:0] is reserved
TRAN	$3=($1)' transpose	4'B1101	$1 addr	reserved	$3 addr	[27:20] is row of $1; [19:12] is col of $1
CONV	$3=$1*$2 $1 is image $2 is kernel	4'B0111	$1 addr	$2 addr	$3 addr	[27:20] is row of $1; [19:12] is col of $1; [11:6] is row of $2; [5:0] is col of $2

POOL	$3=down($1) subsample	4'B1000	$1 addr	MODE 0: max 1: mean	$3 addr	[27:20] is row of $1; [19:12] is col of $1; [11:6] is row of $2; [5:0] is col of $2
SIGM	$3=sigm($1)	4'B1001	$1 addr	reserved	$3 addr	[27:20] is row of $1; [19:12] is col of $1
RELU	$3=relu($1)	4'B1010	$1 addr	reserved	$3 addr	[27:20] is row of $1; [19:12] is col of $1
TANH	$3=tanh($1)	4'B1011	$1 addr	reserved	$3 addr	[27:20] is row of $1; [19:12] is col of $1
GRAY	$3=gray($1)	4'B1100	$1 addr RGB565	reserved	$3 addr YCbCr	[27:20] is row of $1; [19:12] is col of $1
NOP	no operation	0	0	0	0	0
RESET	reset instruct RAM	0	0	0	0	1
START	start operate	0	0	0	0	2

And the total framework is shown in the following figure.

In FPGA, an ISA-NPU is implemented, which realize arithmetic operations inside the module, and the DDR read / write request can be made through FPGA-to-HPS interface and L3 interconnection network.

And there is also an instruction parser that is able to analyze what kind of instruction HPS send. And if the instruction is RESET, then the write-address for instruction cache is set to 0; and if the instruction is valid or NOP, then the instruction is saved into cache; and if the instruction is START, then the FSM is touched, and start computation.

And for the FSM in the NPU instruction executor, once it receives the start-signal, it should loop as following: generate instruction RAM address, and read out the instruction; if the instruction is NOP, then all instructions have been executed, and return to idle; otherwise, transfer the instruction into NPU module, and wait for the NPU to finish the execution of the instruction.

On HPS, the C language program need to send instructions to FPGA (1. send RESET; 2. send arithmetic instructions; 3. send NOP instruction; 4. send START instruction), and wait for the state feedback that implies NPU has already finish the instruction execution.

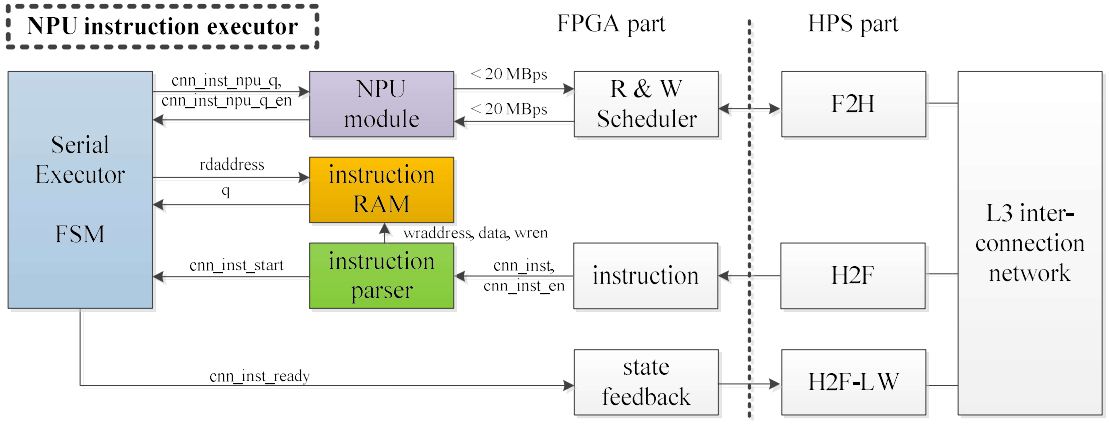


Fig. 16. Architecture of an NPU instruction executor

And for serial executor FSM, we can draw out the jumping logic as following, where black words mean the jumping condition, and brown words mean the executions:

1. when receive the ***reset_n*** signal, FSM jump to IDLE;
2. when in IDLE, if ***cnn_inst_start*** signal is asserted, jump to READ;
3. when in READ, delay for 3 clocks, and we can get NPU instruction. If the instruction is NOP, then jump back to IDLE; else, assert ***cnn_inst_npu_q_en*** and send ***cnn_inst_npu_q*** to NPU module, then jump to SEND;
4. when in SEND, delay for one clock, and de-assert ***cnn_inst_npu_q_en***, and jump to DELAY;
5. when in DELAY, delay for 5 clocks, for the NPU module to receive instruction ***cnn_inst_npu_q***, then jump to WAIT;
6. when in WAIT state, FSM waits for the NPU module to finish the execution of the instruction. And then jump to ADDR;
7. when in ADDR, the address to instruction RAM is increased, and then jump to READ.

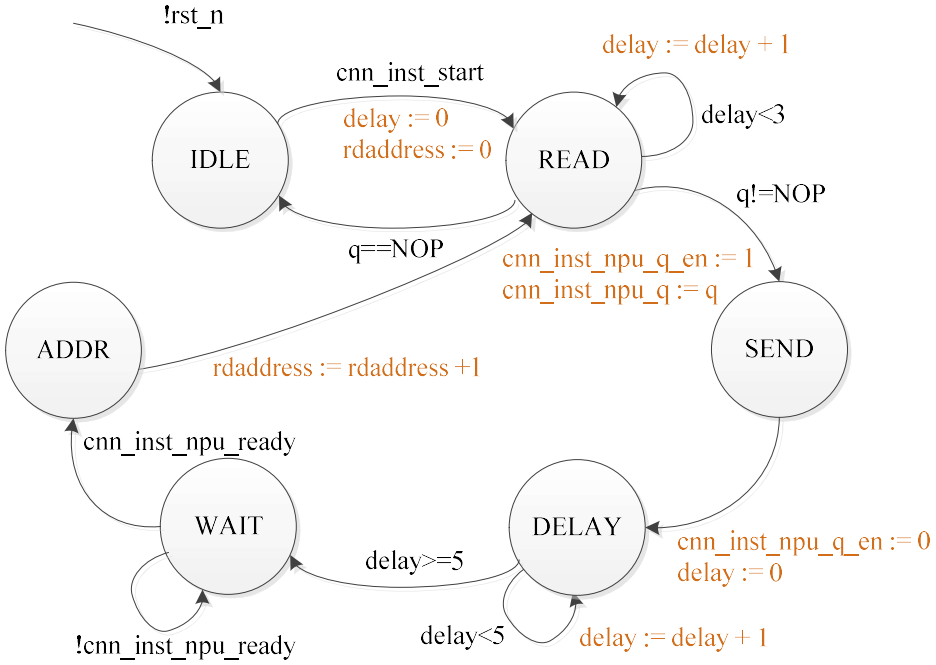


Fig. 17. FSM logic for an ISA-NPU instruction executor

Following is the description of our ISA-NPU's architecture. The module can be separated into four main parts: FSM controlling logic, NPU instruction parser, NPU arithmetic operator and DDR read & write interface.

If the NPU is ready, that means the NPU is not doing any arithmetic operation and ready to receive a new instruction, then the signal ***npu_inst_ready*** will be asserted. ***npu_inst*** is the input port where an NPU instruction is entered. And the signal ***npu_inst_en*** is asserted when ***npu_inst*** is valid. And DDR interfaces are made up with signals ***ddr_write_**** and ***ddr_read_****, and both conform to the Avalon-MM interface protocol. Read and write addresses are given by ****_addr***, and signals ****_req*** give read and write requests, and there is signal named ***ddr_write_data*** that means data written into DDR. These signals must be asserted until the DDR controller gives a ****_ready*** signal, which means a read or a write request has already been completed.

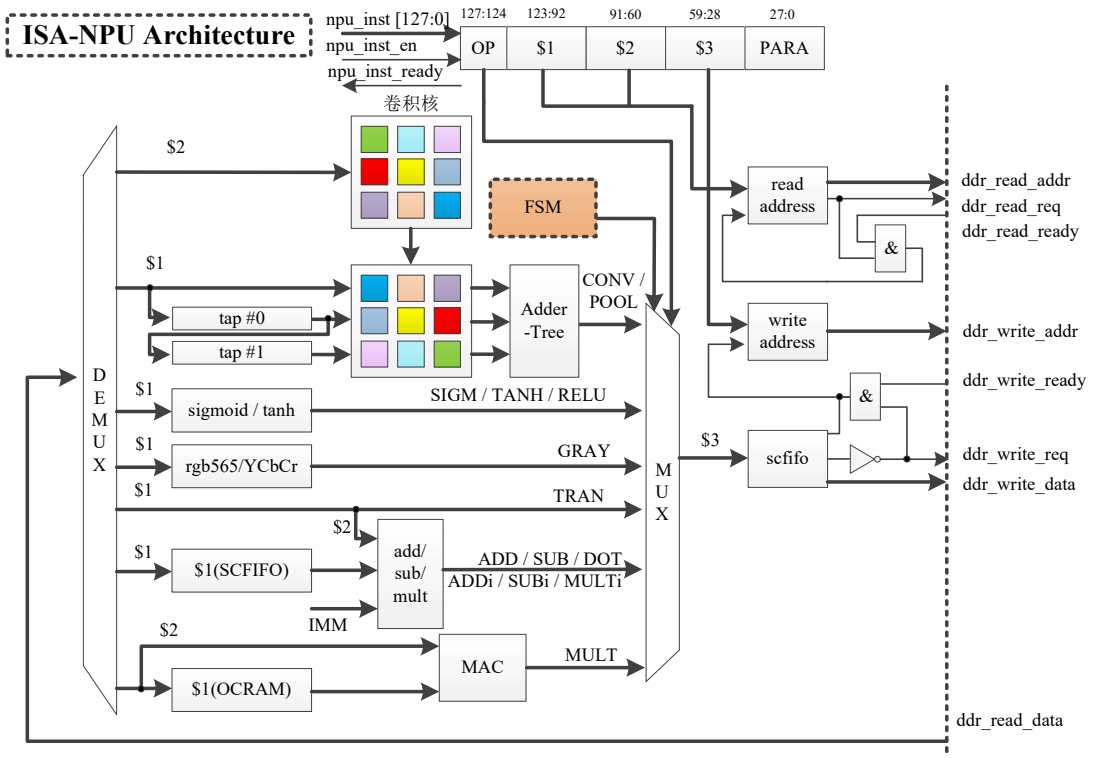


Fig. 18. Architecture of an ISA-NPU module

For two-matrix pointwise arithmetic operation, such as ADD / SUB / DOT, we should notice that data of $1 can be read out first, and then ***at the same time that data of $2 is read out, the function $3 = function($1, $2) can be executed***. So the jumping logic for FSM can be demonstrated as following:

0. when in state 0, if ADD / SUB / DOT has already been finished for each row, then jump out; else, jump into state 1;
1. when in state 1, if one line of $1 is read out, then jump into state 2; else, continue reading the line of $1;
2. when in state 2, wait for the N elements of $1 (one line) to fill into SCFIFO, and then jump to state 3;
3. when in state 3, read one line of $2, and jump to state 4;
4. when in state 4, write back data of $3 from SCFIFO.

And for one-matrix pointwise operation, such as ADDi / SUBi / MULTi / SIGM / TANH / RELU / GRAY / CONV / POOL, we can see that, ***at the same time that data of $1 is read out, the function $3 = function($1) can be executed***. So the jumping logic is also very simple, as following:

0. when in state 0, if ADDi / SUBi / MULTi / SIGM / TANH / RELU / GRAY / CONV / POOL has already been finished for each row, then jump out; else, jump into state 1;
1. when in state 1, if one line of $1 is read out, then jump into state 2; else, continue reading the line of $1;
2. when in state 2, write back data of $3 from SCFIFO.

In order to verify the correctness of the CNN operation based on the NPU instruction, we use Python + Modelsim + Matlab to design a platform to verify the accuracy of the calculation results in the CNN running process. As is shown in the figure below, Python is mainly used to generate

NPU instruction files, SSRAM initialization files, and comparison files of CNN operation results; Modelsim needs to complete the loading of NPU instructions, SSRAM initialization files, and storage of NPU operation results; Matlab compares the results of the two Python and Modelsim operations, and return a visual display.

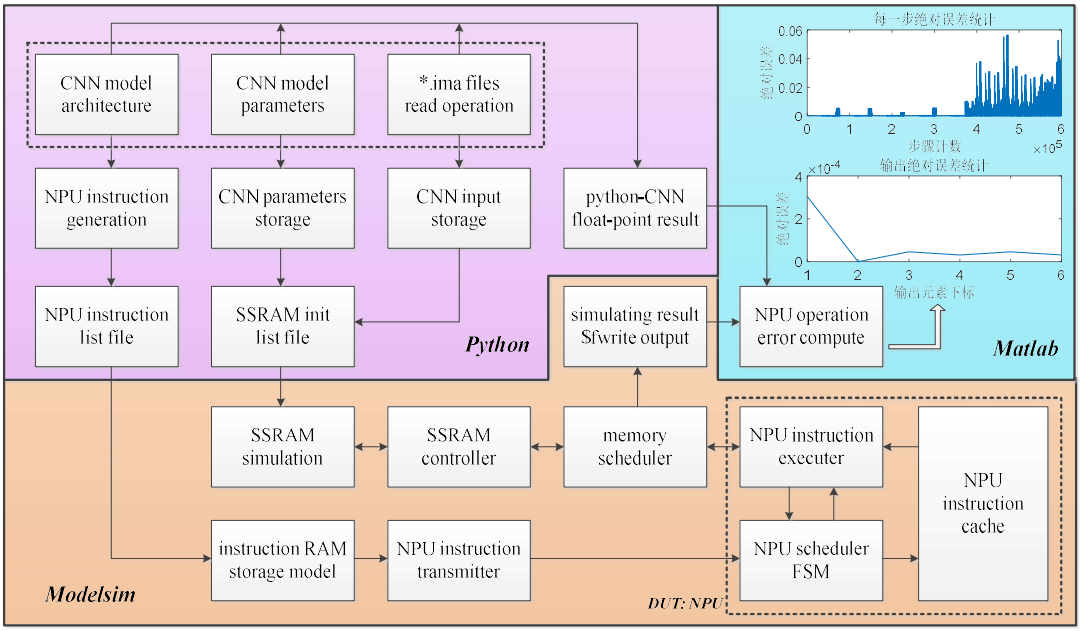


Fig. 19. Architecture of test ISA-NPU precision

Using such a platform, we can accurately know the absolute error of the Verilog version of CNN in the calculation process, the calculation results in every step, and the final network output, compared with single-precision floating-point operations in Python. Through a large number of sample statistics, the final output error is about 1/10,000 level. In the middle calculation process, the absolute error can be controlled within 0.1, and most of the calculation results have no errors. And the computation precision needs of posture recognition are realized.

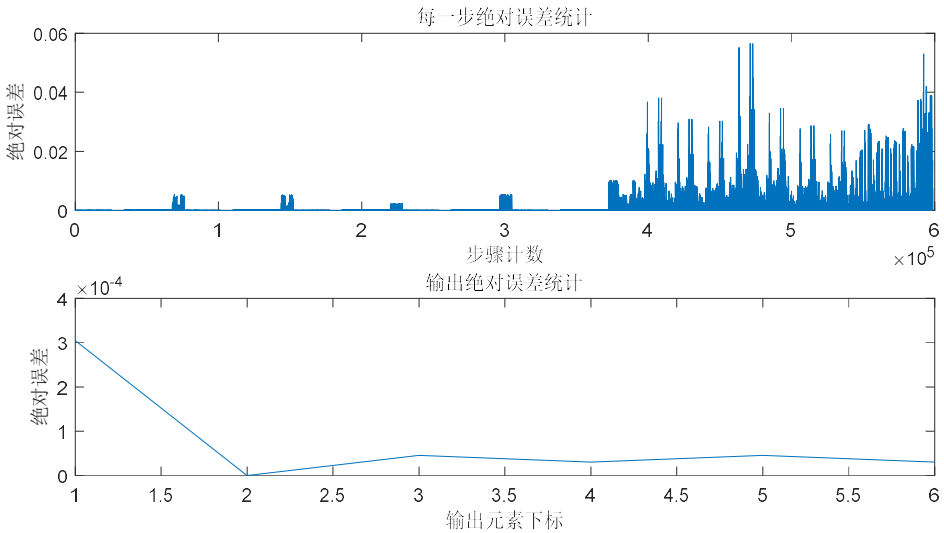


Fig. 20. Error of ISA-NPU computation

6.2 Software function description

For C program on HPS, it needs to do the following tasks:

1. initiate interfaces, use mmap() function to map HPS-to-FPGA and DDR into user space, so that other modules can load / store data from / to FPGA / DDR;
2. respond key-press, use < termios.h> to realize the utility that once a key is pressed, the program is able to know what key it is;
3. load optical flow and HOG+SVM pedestrian detection results from DDR, and generate static / dynamic windows;
4. able to compute IoU of two boxes in an image, and merge the dynamic and static windows to show where person is;
5. sample the image and optical flow in the specific box from DDR, and it is important for us to sample training data / label;
6. send NPU instructions to ISA-NPU through HPS-to-FPGA interface, and read the state of NPU through light-weighted HPS-to-FPGA interface;
7. save the image with box into DDR, for FPGA to load and display.

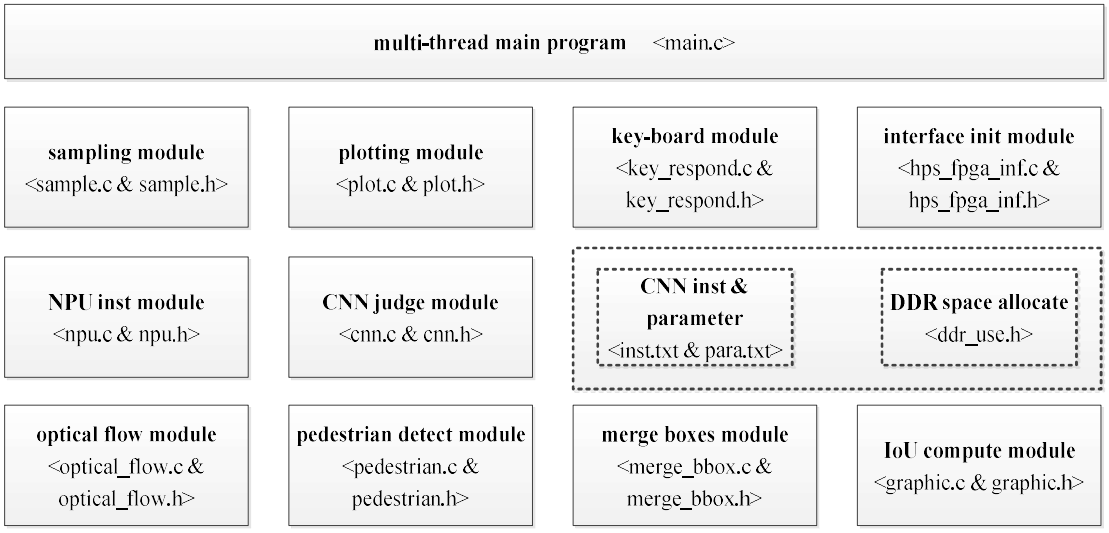


Fig. 21. Architecture of software program

6.2.1 Merge Dynamic and Static Windows

Generating a dynamic window based on optical flow mask is simple: we need to find the minimal zone that is filled up with optical flow mask. And using NMS algorithm, we can generate static windows as well.

The difficult task is how to merge these dynamic and static windows. And here we propose one solution, as is shown in the following figure, And we assume the final box can be described as (C_{final}, S_{final}), where C_{final} is the center position of the final box, and S_{final} is the horizontal and vertical size of the final box.

Firstly, we get dynamic box as $B_{of} = (C_{of}, S_{of})$, where C_{of} is the center position of the dynamic box, and S_{of} is the horizontal and vertical size of the dynamic box.

Secondly, if B_{of} is not NULL, then we interpolate C_{final} and C_{of}, and get the center of testing-window, as $C_{test} = aC_{final} + (1-a)\ C_{of}$. Otherwise, if B_{of} is NULL, then we set $C_{test} = C_{final}$.

Thirdly, we smooth S_{final} in the past several seconds, so that we can get a good description of what size person is. And we get the size of testing window, $S_{test} = smooth(S_{final})$. Then we have the

testing window, $B_{test} = (C_{test}, S_{test})$.

Then we load static windows B_{pd} that are merged using NMS algorithm, and judge if B_{pd} and B_{test} have a large intersection. We use IoU to estimate how large the intersection is:

$$IoU\left(B_{pd}, B_{test}\right) = \frac{S\left(B_{pd} \wedge B_{test}\right)}{S\left(B_{pd} \vee B_{test}\right)}$$

if $IoU(B_{pd}, B_{test}) > \delta$, then we need to merge static window into dynamic window using interpolation, $B_{final} = merge(B_{pd}, B_{of})$.

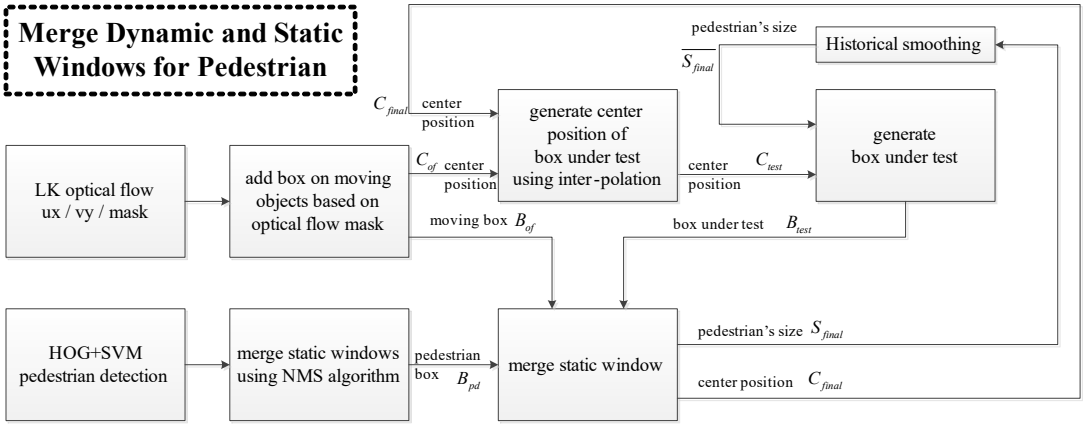


Fig. 22. merge dynamic and static windows

Fig. 23. Experimental result of window merging

6.2.2 Compile the Source Code

In order to better manage the HPS-side program, the files are grouped here, as shown below. Among them, the source folder is the C file of all modules; in the folder "include" there are header files *.h for each module; in the folder "scripts" there are *.bat files that are used to communicate with the HPS via SSH; in the folder "make", there is the compiling rule makefile and compiled intermediate file *.o.

Fig. 24. Folder display

And we can source the script "generate_hps_qsys_header.sh" in SoC EDS environment, and the information of modules and interfaces can be loaded from Qsys file *.sopcinfo, and generate hps_0.h file, which can be used in HPS program. Then we change directory to "make" folder, and run make, and we have the app file for HPS.

```
23  /*
24   * Macros for device 'avalon_h2f', class 'altera_merlin_slave_translator'
25   * The macros are prefixed with 'AVALON_H2F_'.
26   * The prefix is the slave descriptor.
27   */
28  #define AVALON_H2F_COMPONENT_TYPE altera_merlin_slave_translator
29  #define AVALON_H2F_COMPONENT_NAME avalon_h2f
30  #define AVALON_H2F_BASE 0x0
31  #define AVALON_H2F_SPAN 16777216
32  #define AVALON_H2F_END 0xffffff
33
```

Fig. 25. generated hps_0.h file where modules / interfaces in Qsys are defined

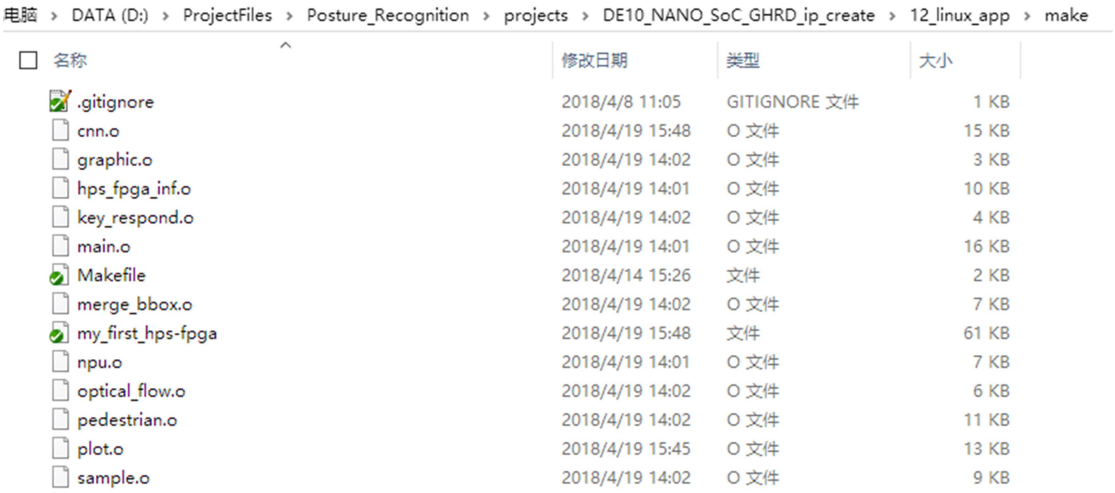


Fig. 26. use makefile to compile the whole HPS program

6.2.3 Sample and Filter

We use two-stream CNN to recognize that posture the person makes, so we should collect samples of different postures to train CNN's parameters. And following is the flow we use. Firstly, as the video stream goes from the camera, LK optical flow and HOG+SVM are both computed at the same time, and the results are saved into DDR, through FPFA-to-SDRAM interfaces. Then the HPS would read out the results from DDR using mmap() function, and merge the dynamic and static windows to figure out where the person is. And then the image and optical flow in the area is sampled and saved into an image file *.ima, which is then transferred to PC via SSH protocol. And on the PC, we use matlab to filter the samples, and delete ambiguous samples, and then train the CNN in the TensorFlow framework. The parameters in CNN are then saved, and instructions generated. Finally, CNN parameters and instructions are then sent to HPS via SSH again, and

parameters would be saved into DDR, and instructions sent to FPGA.

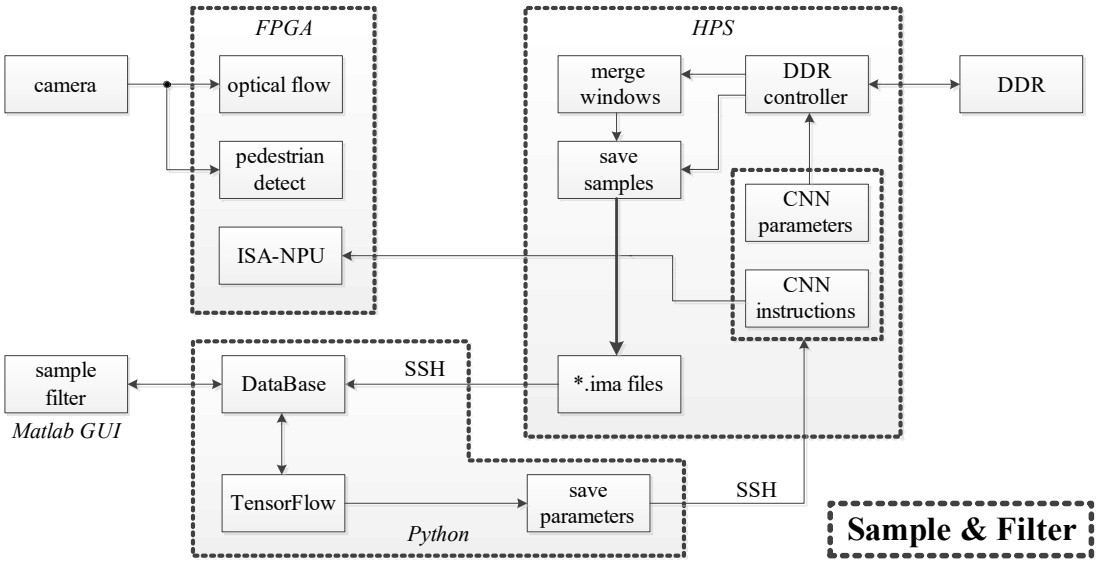


Fig. 27. The flow of sample collection, filtering and CNN training

Samples collected can be not good enough for CNN training, as the following figure shows. For single-hand waving, the image and optical flow is good, while for squatting, the samples can be ambiguous with samples of "walking". And we need to delete these ambiguous samples, using a GUI program in Matlab.

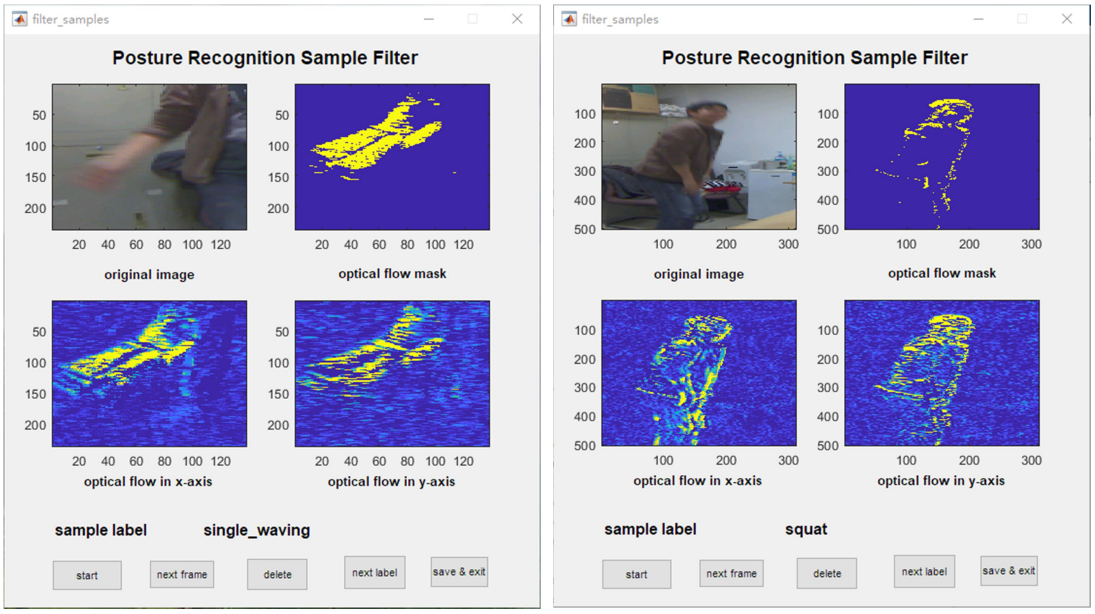
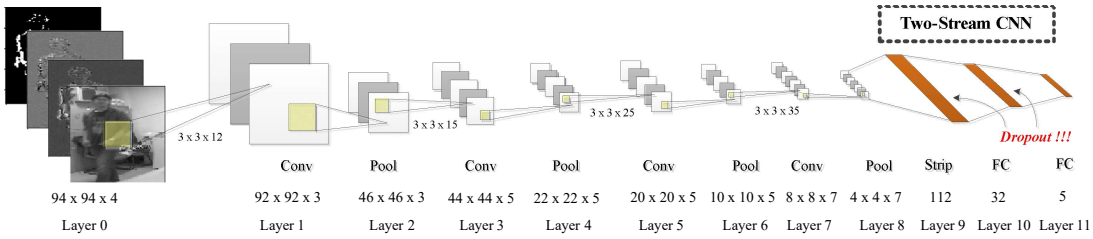


Fig. 28. The posture sample filter in Matlab

6.2.4 Train Two-Stream CNN

As is shown previously, we use TensorFlow framework to train our two-stream CNN. And the architecture of the network is similar to LeNet-5, as is shown following. The input has 4 channels, and each channel is 94 x 94 size; then we use 12 convolution kernels in 3x3 size, to get convolution results, which are then input to pooling layer; and then continue convolution and pooling layer, until the image is small enough, we strip the output channels of pooling layer 8, and then use fully-connection layer 10 and layer 11 to finally get the judgement result. And to improve the generalization performance of the CNN model, we use L-2 regularization in convolution

kernels and dropout in fully-connection layer.



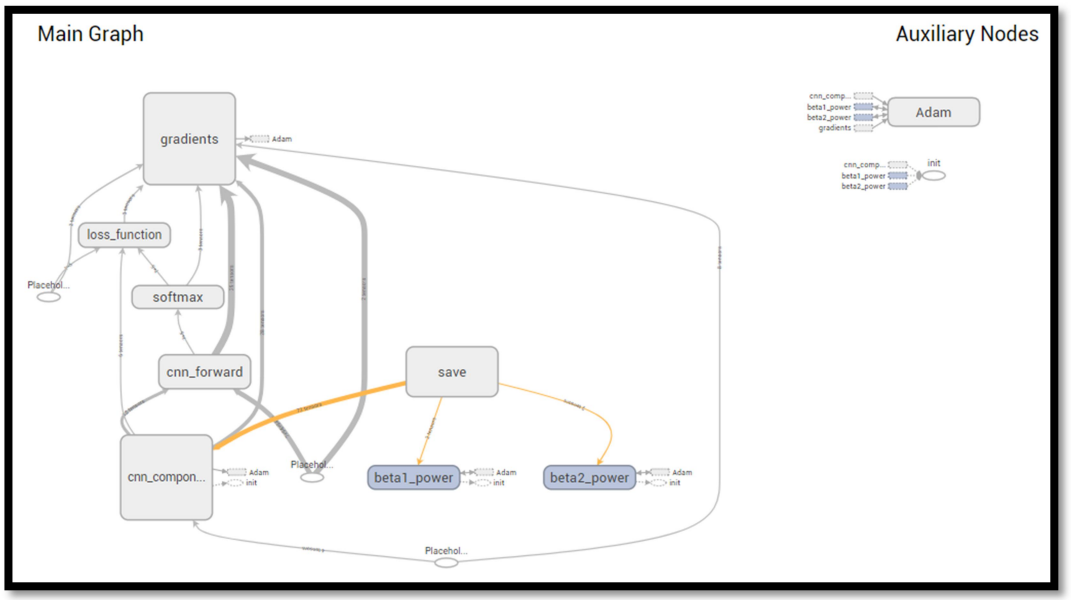


Fig. 29. The proposed two-stream CNN for posture recognition

And we can train the CNN model in TensorFlow with GPU. The total training progress takes a half of hour to reach 90% precision rate on training set, and 88% on testing set.

Moreover, we can also test TP rate (true positive) and FP rate (false positive) on samples of different postures. And the result shows “null”, “standing” and “squatting” are the best, and “waving” and “walking” are not as good as the others.

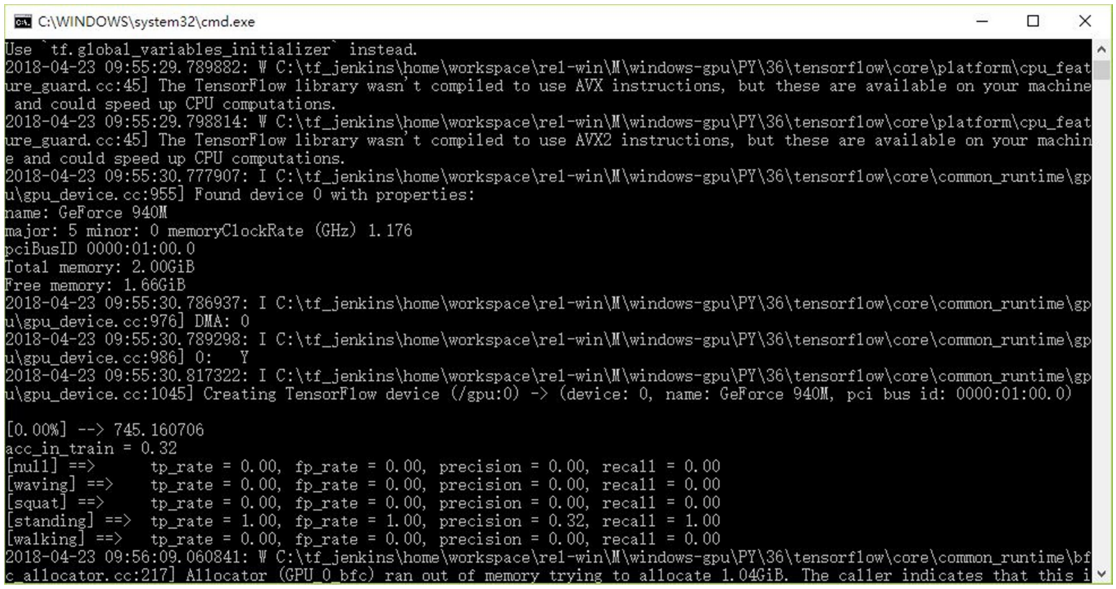


Fig. 30. Start training the CNN model

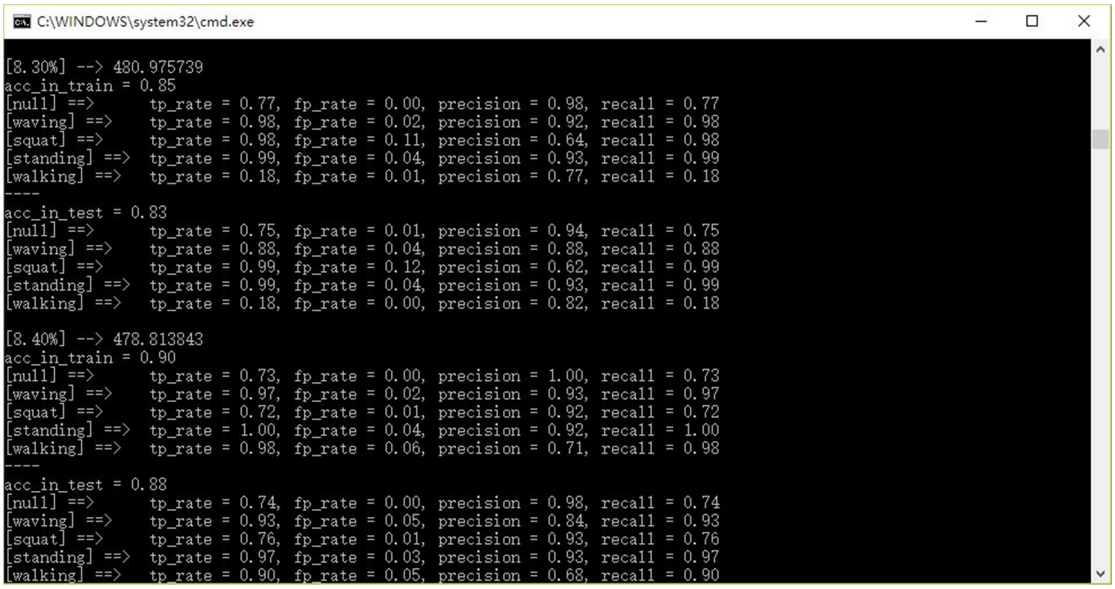


Fig. 31. finish training the CNN model

Tbl. 4. The performance of two-stream CNN

postures		null	waving	squatting	standing	walking
training	**TP rate**	0.73	**0.97**	0.72	**1.00**	**0.98**
	FP rate	0.00	**0.02**	0.01	**0.04**	**0.06**
testing	**TP rate**	0.74	**0.93**	0.76	**0.97**	**0.90**
	FP rate	0.00	**0.05**	0.01	**0.03**	**0.05**

6.2.5 Generate NPU Instructions

As we know, the computation of CNN is complicated, but we can still divide the whole computation into single instructions such as CONV, POOL, ADD, ADDi, etc. And here we use a Python scripts to transform the CNN model into our ISA-NPU instructions. And the progress is also very simple.

For each layer, we execute following steps:

1. if the layer is inputting, then we initiate input and output addresses in DDR;
2. else if the layer is convolution, then for each output channel, we execute:
    a) for each input channel, we convolve the input channel with its own convolution kernel, using CONV instruction;
    b) accumulate the results of convolution, using ADD instruction;
    c) add bias to the accumulation, using ADDs instruction, which means a matrix add a scalar;
    d) map the sum with a non-linear function such as sigmoid or tanh, using SIGM or TANH instruction;
3. else if the layer is pooling, then we use POOL instruction to map each input channel to output channel;
4. else if the layer is strip, then we use ADDi instruction to concatenate the input channels;
5. else if the layer is fully-connection, then we execute:
    a) use MULT instruction to multiply the input and the weight;
    b) use ADD instruction to add bias to the multiplication result;
    c) use SIGM or TANH to map the result to output channel;

And we can then get ISA-NPU instructions as following figure. And the picture below shows how many DDR read / write requests the ISA-NPU rises, and if the DDR controller can respond to each request in one clock, then we can compute the period of the computation of CNN. And if we assume the module can work with a 50 MHz clock, then we can see that the module takes about 16.28 ms to compute CNN once.

npu_inst.txt

```
[['I', 94, 94, 4], ['C', 3, 3, 3], ['S', 2, 2], ['C', 3, 3, 5], ['S', 2, 2], ['C', 3, 3, 5], ['S', 2,
layer 0: input
layer 1: convolution
CONV, @0E000000, @0A000000, @0C000000, M=94, N=94, Km=3, Kn=3
    inst=70E0000000A0000000C0000005E5E0C3
CONV, @0E010000, @0A010000, @0C010000, M=94, N=94, Km=3, Kn=3
    inst=70E0100000A0100000C0100005E5E0C3
CONV, @0E020000, @0A020000, @0C020000, M=94, N=94, Km=3, Kn=3
    inst=70E0200000A0200000C0200005E5E0C3
CONV, @0E030000, @0A030000, @0C030000, M=94, N=94, Km=3, Kn=3
    inst=70E0300000A0300000C0300005E5E0C3
ADD, @0C000000, @0C010000, @0C000000, M=92, N=92
    inst=00C0000000C0100000C0000005C5C000
ADD, @0C000000, @0C020000, @0C000000, M=92, N=92
    inst=00C0000000C0200000C0000005C5C000
ADD, @0C000000, @0C030000, @0C000000, M=92, N=92
    inst=00C0000000C0300000C0000005C5C000
ADDs, @0C000000, @0A040000, @0C000000, M=92, N=92
    inst=E0C0000000A0400000C0000005C5C000
SIGM, @0C000000, xx, @0F000000, M=92, N=92
    inst=90C0000000000000000F0000005C5C000

layer 10: fully_connection
MULT, @0F000000, @0A6B0000, @0C000000, M=1, N=112, P=32
    inst=40F0000000A6B00000C0000000170200
ADD, @0C000000, @0A6C0000, @0C010000, M=1, N=32
    inst=00C0000000A6C00000C0100000120000
SIGM, @0C010000, xx, @0E000000, M=1, N=32
    inst=90C0100000000000000E0000000120000
layer 11: fully_connection
MULT, @0E000000, @0A6D0000, @0C000000, M=1, N=32, P=5
    inst=40E0000000A6D00000C0000000120050
ADD, @0C000000, @0A6E0000, @0C010000, M=1, N=5
    inst=00C0000000A6E00000C0100000105000
SIGM, @0C010000, xx, @0F000000, M=1, N=5
    inst=90C0100000000000000F0000000105000


DDR-READ = 481358, DDR_WRITE=332819,
TOTAL-TIME[estimated in 50 MHz]=16.283540 ms
```

Fig. 32. generated NPU instructions

7. Performance Parameters

The proposed posture recognition system is mainly made up with following components:

1. LK optical flow;
2. HOG + SVM pedestrian detection;
3. Extract and merge dynamic and static windows;
4. Posture recognition with two-stream CNN.

And the performance of the posture recognition can be considered from the following four aspects:

1. Speed of each module, which is influenced by the max frequency of the RTL module, and how many clocks the module takes to finish the whole computation flow;
2. Area of each module, which means how many ALMs, memory bits and DSPs the module consumes;

The following table shows the speed and resource-consumption of each main module. For LK optical flow computation module, Fmax is 118.68 MHz, and considering that the module is designed for 800 x 600 resolution, we can compute up to 247 frames per second; and for HOG + SVM pedestrian detection module, Fmax is not fast, and is 76.8 MHz, and we can compute up to

160 frames per second; and for ISA-NPU, Fmax is 79.6 MHz, and for the two-stream CNN, if DDR controller can respond to each read / write request per clock, then the module can compute up to 97.7 times per second.

Tbl. 5. The theoretic limit of performance of modules

Performance		LK Optical Flow	HOG + SVM	ISA-NPU
Speed	**Fmax**	118.68 MHz	76.8 MHz	79.6 MHz
	Period	800 x 600	800 x 600	814,177
	FPS	247	160	97.7
Area	**ALMs**	7,799	14,105	3,530
	Memory Bits	188,202	1,167,512	73,984
	DSPs	41	16	43

However, as the experiment implies, the speed of the posture recognition system is finally limited by the DDR bandwidth. And it is predictable to some extent.

Firstly, the camera MT9D111 has got a clock in 35 MHz from FPGA, and it is set to output video stream in the resolution of 800 x 600 at 20 fps, and the format is RGB565. So the max data bandwidth is 70 MBps.

Secondly, Considering the computation of optical flow requires reading previous frame in DDR, and the results should be stored again, then the optical flow requires the DDR bandwidth up to 140 MBps.

Thirdly, for HDMI output, we set the pixel clock to 26 MHz, and output HDMI video in the resolution of 1024 x 768 @ 24 fps, so the max data bandwidth is 104 MBps.

So the optical flow and the video store & display modules requires 314 MBps in total. And the ISA-NPU is loaded to FPGA-to-HPS interface, and access DDR through L3 interconnection in HPS. As the experimental result shows, ***the DDR bandwidth distributed to ISA-NPU is less than 20 MBps, which made the NPU run much slower than expected***, and the total computation of CNN takes about 0.261 s. In addition, dynamic window extraction is also time-consuming, which taske about 70 us.

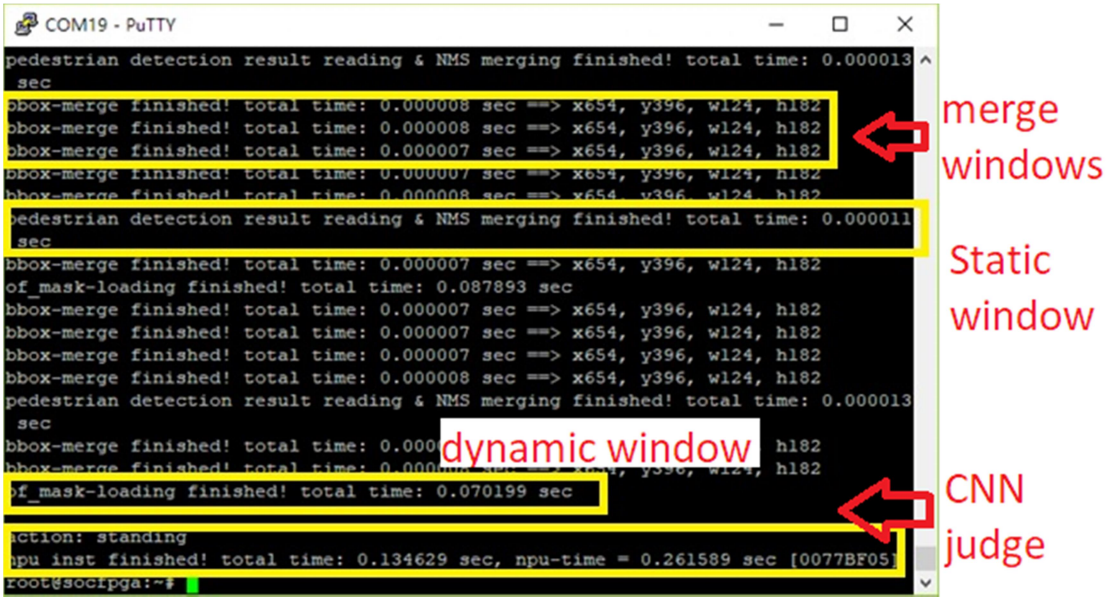


Fig. 33. The experimental performance of modules

Tbl. 6. The experimental performance of hardware modules

Performance		LK Optical Flow	HOG + SVM	ISA-NPU
Speed	Fmax	35 MHz	35 MHz	66.67 MHz
	Period	800 x 600	800 x 600	814,177
	FPS	20	20	3.84

Tbl. 7. The resource utility of the total hardware system

Item	ALMs	Memory Bits	DSPs
Utility	37,933 (91 %)	2,513,876 (44 %)	106 (95 %)

Tbl. 8. The experimental performance of software modules

Performance	Dynamic Window	Static Window	Merge Window
Time	70.2 ms	11 us	8 us

And the following figure is the Chip-Planner view after Placement and Routing. As is shown in the figure, the resources on FPGA are mostly consumed by static pedestrian detection, optical flow computation, ISA-NPU and DDR read and write schedulers.

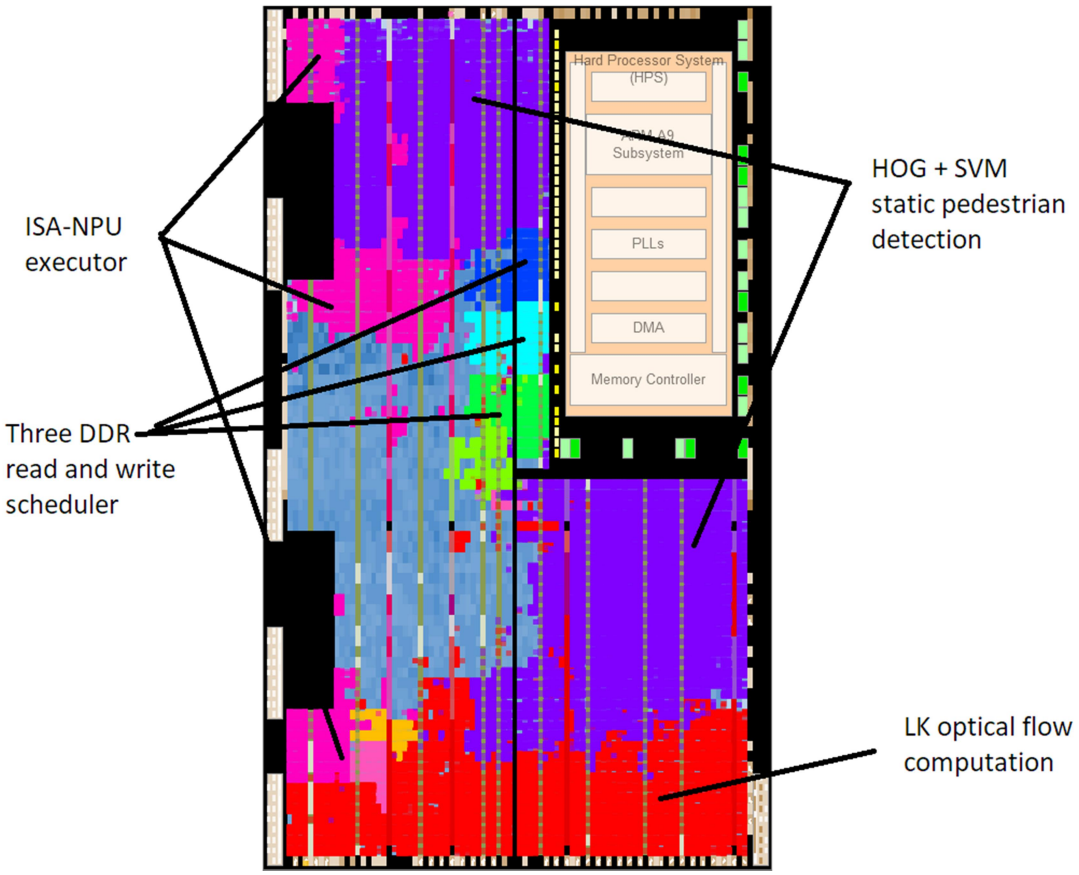


Fig. 34. The Chip-Planner view of our posture recognition system

8. Design Architecture

8.1 Hardware Architecture

The hardware part of our posture recognition system is mainly made up with camera/HDMI configure module, LK optical flow module, HOG & SVM pedestrian detection module, video store & display module, ISA-NPU module.

And since the optical flow module, video output module, and ISA-NPU run in different clock field, they cannot be directly connected to FPGA-to-SDRAM or FPGA-to-HPS interfaces. So we design a Read & Write Scheduler to perform read and write operations alternately.

Moreover, as HDMI output video should be divided into several parts, where the original video, and optical flow, and video with boxes should be displayed, the video display module should read different blocks of DDR sdram, and output subtitles as well.

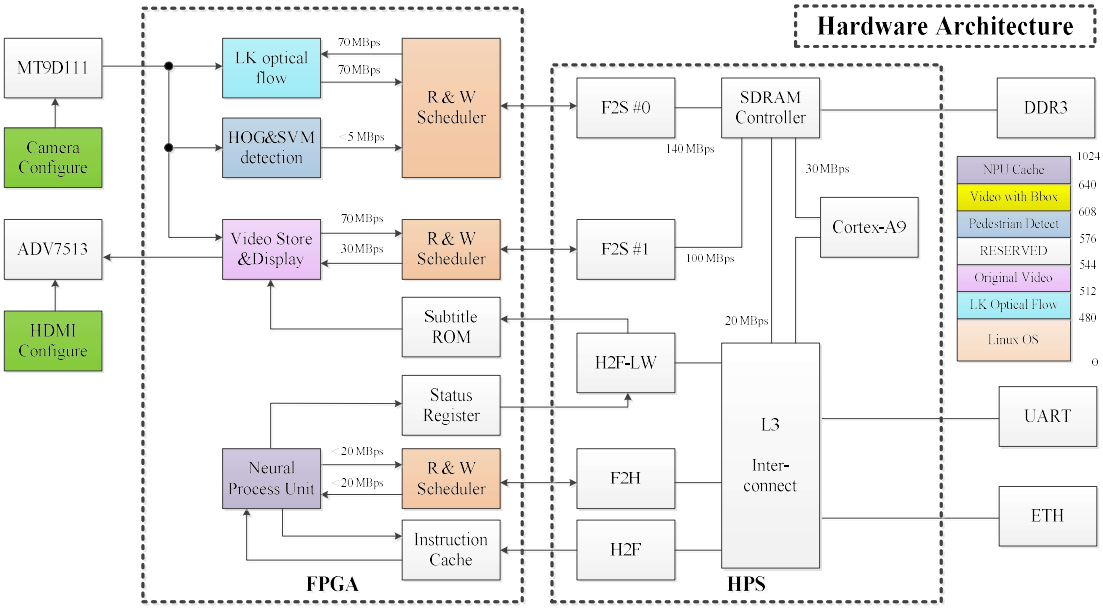


Fig. 35. The hardware architecture of posture recognition system

8.2 Software Flow

In software flow, HPS should firstly initiate all interfaces, and map HPS-to-FPGA and DDR to user space; and then reset FPGA modules; and load CNN parameters from para.txt, and save them into DDR; then HPS sends CNN instructions to FPGA, which are read from inst.txt; after clearing exit_flag, HPS creates threads that are used in posture recognition, and goes into the infinite loop, until exit_flag is asserted.

And there are 5 threads that are used in posture recognition:

1. thread of generating dynamic window, in which HPS reads optical flow results from DDR, and then generate the box inside which, there are lots of optical mask;
2. thread of generating static window, in which HPS reads pedestrian detection result from DDR, and use NMS algorithm to get most possible boxes;
3. thread of merging dynamic and static windows. The merging method has already been introduced previously;
4. thread of CNN computing, in which HPS send START instruction to FPGA, and wait for ISA-NPU to finish CNN computing;

5. thread of key-board monitoring, in which HPS monitor the input from the serial port, and once it detects “ESC” is pressed, exit_flag should be asserted, and all program will exit later.

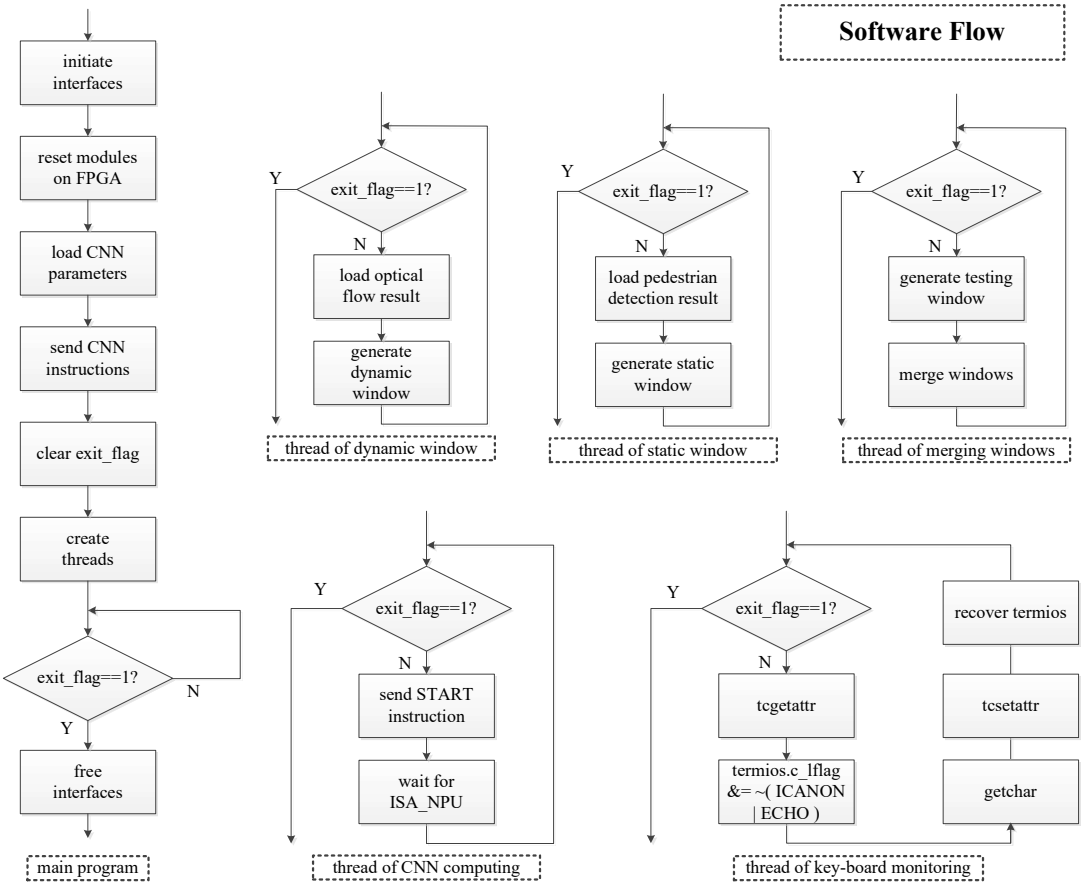


Fig. 36. The software flow of posture recognition system

8.3 Physical Display of the System

We connect the camera with DE10-Nano FPGA Kit, and connect to HDMI monitor. And following figures are the physical display of the proposed posture recognition system.

We can login to the board via the serial port, or Ethernet via SSH protocol.

And as the person walks ahead of the camera, the system can capture video stream, and analyze where the person is, and add boxes in the video at the same time. And the system finally uses CNN to judge what posture the person makes.

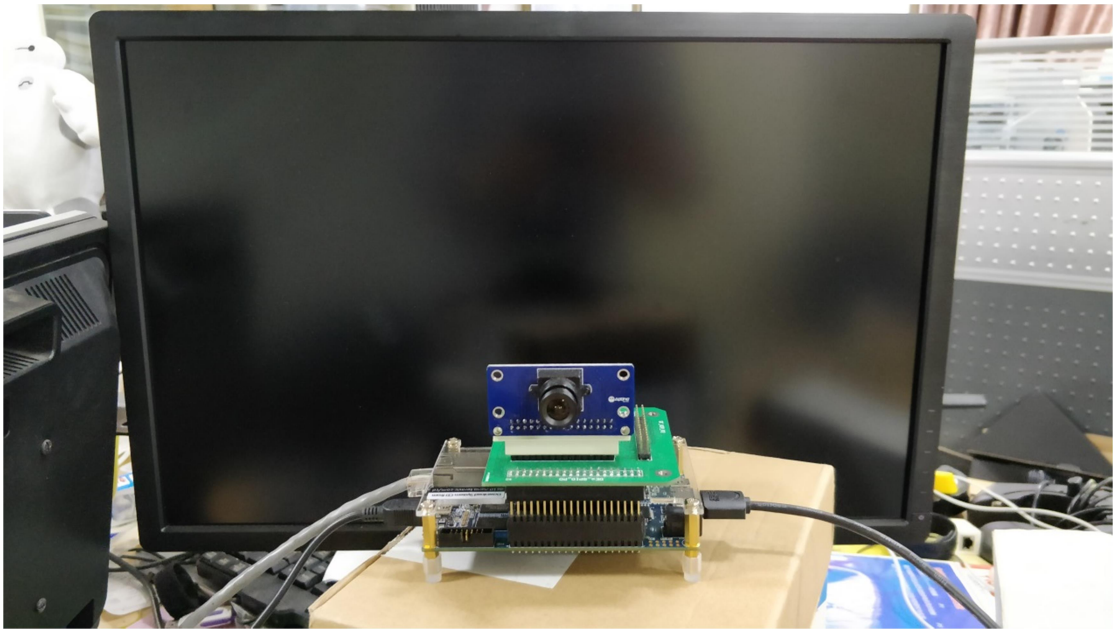


Fig. 37. The physical display of posture recognition system

And in the following picture, the image can be divided into four parts.

1. Left top: the original video;
2. Right Top: Add boxes into video; red box is static box, dark blue is dynamic box, light-blue is testing window, and the green one is the final merged window;
3. Left Below: the LK optical flow;
4. Right Below: the video with subtitle which shows what posture it is.

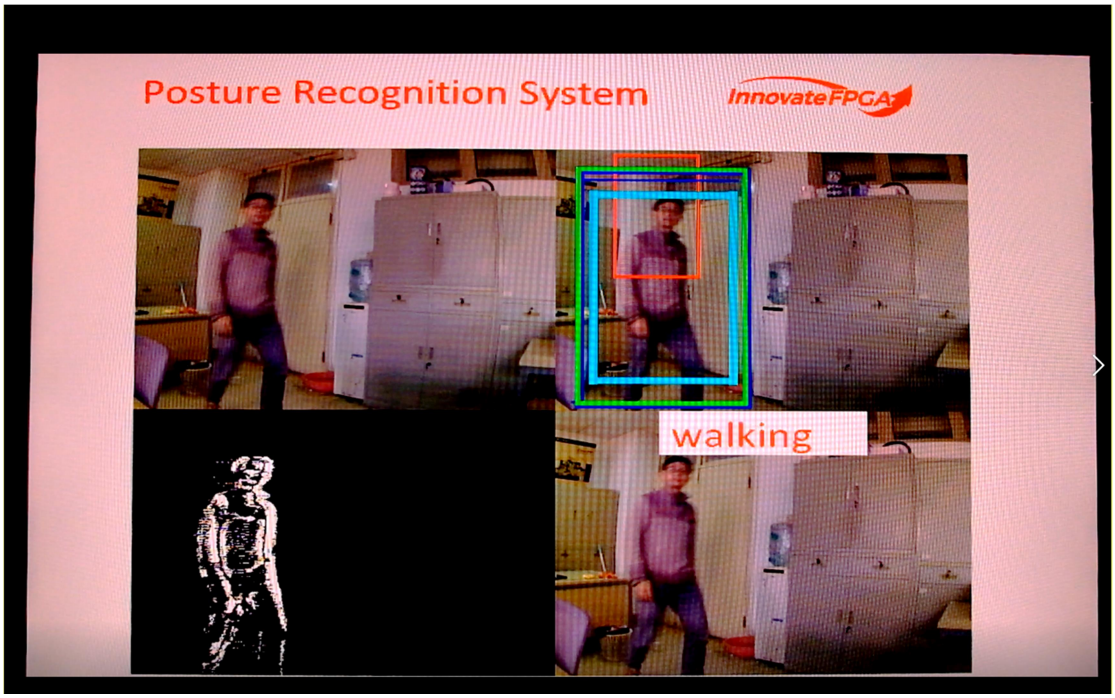


Fig. 38. The Performance display of posture recognition system