Kneron Inc

KL520 DME Tutorial

Kneron Inc

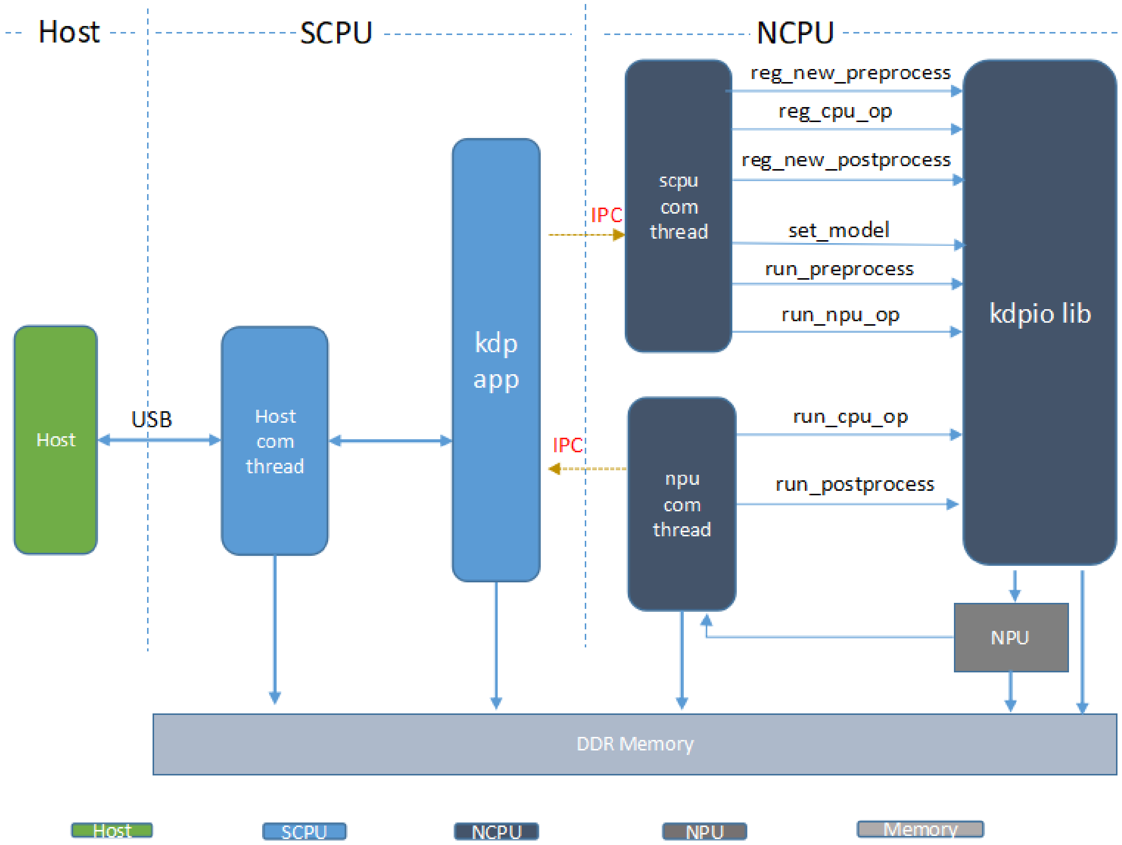
Table of Contents

1. Support New Models in DME Mode

This chapter describes the overview flow in DME mode, and the key points which need to be pay attention to during implementation. The example in this chapter is based on Tiny-YOLO-V3 model, and the dimension of Input Node is 3x224x224 (Channel*Height*Width) in ONNX format.

1.1 DME Overview Chart

The below diagram shows the flow chart from host to NPU and the key components involved in DME mode.



1.1.1 Host Side

In DME mode, Host is responsible for communicating with SCPU, sending DME commands, firmware information (fw_info.bin), model data (all_models.bin), configuration, and input images to SCPU, and retrieving back the output results from SCPU through USB interface.

1.1.2 SCPU

In DME mode, SCPU is responsible for parsing DME commands from Host, writing firmware information, model data, and input images to assigned memory address, launching application, communicating with NCPU, and sending back the output results to Host through USB interface.

1.1.3 NCPU and NPU

In DME mode, NCPU is responsible for communicating with SCPU and NPU, registering new preprocess/cpu_op/postprocess functions, setting model, running preprocess/cpu_op/post_process, and enabling NPU. NPU is responsible for running npu_op, interrupting when meeting cpu_op, and continuing running npu_op.

1.1.4 Memory

The DDR memory is the space which can be accessed by SCPU, NCPU and NPU. The firmware information, model data, input images, and output results are located in this space.

1.2 Limitations for KL520 SDK

(1) The input width and height of models is limited to be less than 256 if keeping aspect ratio.

In this tutorial, the input dimension is 3x224x224 for Tiny-YOLO-V3.

(2) The input width and height of model larger than 255 is supported when doing inference with 1 model, and aspect ratio is changed by default in image preprocessing.

(3) Right shift in image preprocessing is no supported by NPU.

For Tiny-YOLO-V3, the input pixel value range is 0~1 (x/256, x ranges from 0~255). After compiling with compiler, we get that the input_radix is 7 and input_scale is 1.0 for this model. If using NPU to do preprocessing with RGB data, the data is converted to fixed point with the formula x/256 * 2^radix * scale (equals to x/2), which means that each value need right-shift 1 bit. As NPU does not support right-shift for the input data, there are 2 methods to support this model: (a) bypass preprocess and provide the preprocessed RGBA image input directly; (b) do right-shift with NCPU (time-consuming). In this tutorial, method (a) and (b) are introduced for the preprocessing of Tiny-YOLO-V3.

1.3 CPU and Output Node Info

The CPU and Output Node info is shown in ioinfo.csv. For Tiny-YOLO-V3, the ioinfo is summarized as below. From this table, cpu_op upsample need to be implemented and there are 2 output nodes used for postprocessing in this model.

Node Type	**Index**	**Name**	**Note**
c	0	up_sampling2d_1_o0	Upsample CPU Node
o	0	conv2d_10_o0	First Output Node
o	1	conv2d_13_o0	Second Output Node

1.4 Generate New Configuration Files

Use script dme_conf_generator.py to generate configuration files. Example code is in 2.3.1.

1.4.1 Parameter Description

Parameters	Description
Models_selection	Select models to be used in DME
output_num	Total output number for this model
image_col	Width of the input image
image_row	Height of the input image
image_ch	Channel of the input image (3 for RGB565, 4 for RGBA)
image_format	Input image format which contains settings of “do preprocess” / “bypass preprocess”, “output detection results” / “output raw results”, “subtract 128 or not”, “rotate or not”, “parallel processing or not”, and input image type.

1.4.2 Image Format

SDK code snippet of Image format flags is defined as below.

```
/* Image format flags */
#define IMAGE_FORMAT_SUB128                     BIT31
#define IMAGE_FORMAT_ROT_MASK                   (BIT30 | BIT29)
#define IMAGE_FORMAT_ROT_SHIFT                  29
#define IMAGE_FORMAT_ROT_CLOCKWISE              0x01
#define IMAGE_FORMAT_ROT_COUNTER_CLOCKWISE      0x02

#define IMAGE_FORMAT_RAW_OUTPUT                 BIT28

#define IMAGE_FORMAT_CHANGE_ASPECT_RATIO        BIT20

#define IMAGE_FORMAT_BYPASS_PRE                 BIT19
#define IMAGE_FORMAT_BYPASS_NPU_OP              BIT18
#define IMAGE_FORMAT_BYPASS_CPU_OP              BIT17
#define IMAGE_FORMAT_BYPASS_POST                BIT16

#define IMAGE_FORMAT_NPU                        0x00FF
#define NPU_FORMAT_RGBA8888         0x00
```

```
#define NPU_FORMAT_NIR              0x20
/* Support YCBCR (YUV) */
#define NPU_FORMAT_YCBCR422         0x30
#define NPU_FORMAT_YCBCR444         0x50
#define NPU_FORMAT_RGB565           0x60

/* Determine the exact format with the data byte sequence in DDR memory: [lowest byte]...[highest byte] */
#define NPU_FORMAT_YCBCR422_CRY1CBY0 0x30
#define NPU_FORMAT_YCBCR422_CBY1CRY0 0x31
#define NPU_FORMAT_YCBCR422_Y1CRY0CB 0x32
#define NPU_FORMAT_YCBCR422_Y1CBY0CR 0x33
#define NPU_FORMAT_YCBCR422_CRY0CBY1 0x34
#define NPU_FORMAT_YCBCR422_CBY0CRY1 0x35
#define NPU_FORMAT_YCBCR422_Y0CRY1CB 0x36
#define NPU_FORMAT_YCBCR422_Y0CBY1CR 0x37  // Y0CbY1CrY2CbY3Cr...
```

After combining different bit setting, the image_format is generated. The below table shows 8 types of image format which are used in DME mode. 0x10080000, 0x00080000, 0x10000060, and 0x00000060 are used for Tiny-YOLO-V3.

Image Format	**Description**	**Output Mode**
0x10080000	Bypass postprocess, bypass preprocess, RGBA8888 input	Raw outputs
0x00080000	Bypass preprocess, RGBA8888 input	Detection outputs
0x80000060	Image pixel value subtract 128, RGB565 input with data byte sequence in DDR memory G0[2:0]B0[4:0]R0[4:0]G0[5:3]G1[2:0]B1[4:0]R1[4:0]G1[5:3]...	Detection outputs
0x80000037	Image pixel value subtract 128, YCBCR422 (YUV422) input with data byte sequence in DDR memory Y0CbY1CrY2CbY3Cr...	Detection outputs
0x10000060	Bypass postprocess, RGB565 input	Raw outputs
0x00000060	RGB565 input	Detection outputs
0x88000060	Image pixel value subtract 128, parallel processing with 2 image inputs, RGB565 input	Detection outputs

0x98000060	Image pixel value subtract 128, bypass postprocess, parallel processing with 2 image inputs, RGB565 input	Raw outputs

1.4.3 Example Configurations in Host

(1) The below Python code snippet of configuration is for outputting detection results of Tiny-YOLO-V3 with RGBA input. If bypassing preprocess for Tiny-YOLO-V3, the RGBA input is provided.

```
# example for Tiny-YOLO-V3
pre_bypass       = 1
post_bypass      = 0

models_selection = 0x00000001 # Reserved. 0x00000001 if there is only one model
output_num       = 2
image_col        = 224
image_row        = 224
image_ch         = 4
image_format     = 0
parallel         = 0

if pre_bypass:
    image_format = image_format | 1 << 19 # 1 << 19 to bypass pre

if post_bypass:
    image_format = image_format | 1 << 28 # 1 << 28 to bypass post and output raw results

if parallel:
    image_format = image_format | 1 << 27 # 1 << 27 to do parallel processing
```

(2) The below configuration is for outputting RAW results of Tiny-YOLO-V3 with RGBA input.

```
# example for Tiny-YOLO-V3
pre_bypass       = 1
post_bypass      = 1

models_selection = 0x00000001 # Reserved. 0x00000001 if there is only one model
output_num       = 2
image_col        = 224
```

```
image_row        = 224
image_ch         = 4
image_format     = 0
parallel         = 0

if pre_bypass:
    image_format = image_format | 1 << 19 # 1 << 19 to bypass pre

if post_bypass:
    image_format = image_format | 1 << 28 # 1 << 28 to bypass post and output raw results

if parallel:
    image_format = image_format | 1 << 27 # 1 << 27 to do parallel processing
```

(3) The below configuration is for outputting detection results of Tiny-YOLO-V3 with RGB565 input.

```
# example for Tiny-YOLO-V3
pre_bypass       = 0
post_bypass      = 0

models_selection = 0x00000001 # Reserved. 0x00000001 if there is only one model
output_num       = 2
image_col        = 640
image_row        = 480
image_ch         = 3
image_format     = 0
parallel         = 0x00000060

if pre_bypass:
    image_format = image_format | 1 << 19 # 1 << 19 to bypass pre

if post_bypass:
    image_format = image_format | 1 << 28 # 1 << 28 to bypass post and output raw results

if parallel:
    image_format = image_format | 1 << 27 # 1 << 27 to do parallel processing
```

(4) The below configuration is for outputting RAW results of Tiny-YOLO-V3 with RGB565 input.

```
# example for Tiny-YOLO-V3
pre_bypass       = 0
```

```
post_bypass      = 1

models_selection = 0x00000001 # Reserved. 0x00000001 if there is only one model
output_num       = 2
image_col        = 640
image_row        = 480
image_ch         = 3
image_format     = 0
parallel         = 0x00000060

if pre_bypass:
    image_format = image_format | 1 << 19 # 1 << 19 to bypass pre

if post_bypass:
    image_format = image_format | 1 << 28 # 1 << 28 to bypass post and output raw results

if parallel:
    image_format = image_format | 1 << 27 # 1 << 27 to do parallel processing
```

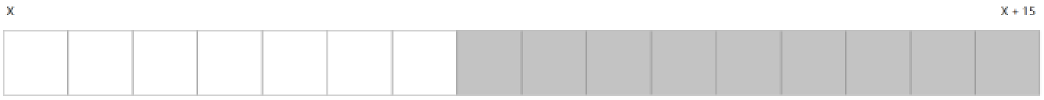
1.4.4 Struct to Store the Configuration in SCPU

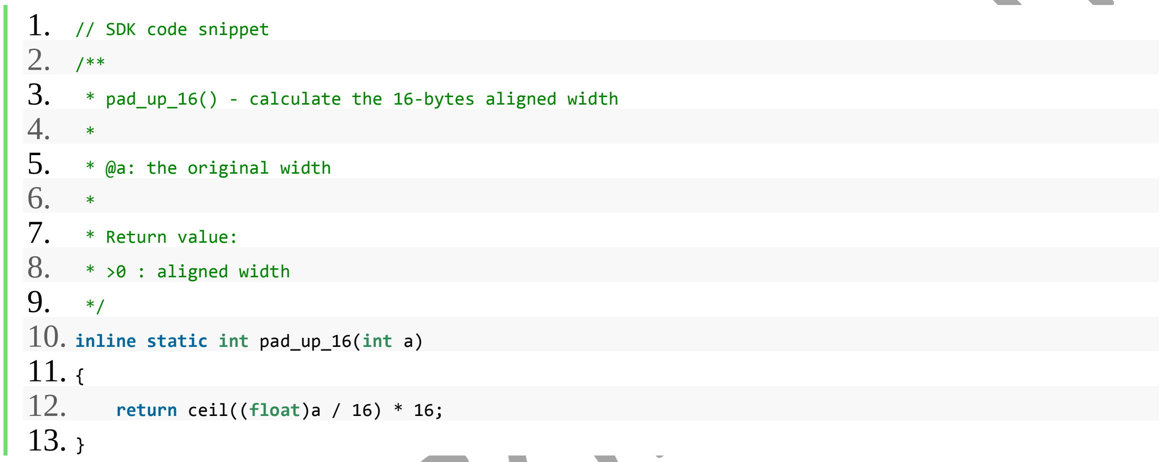
```
/* KDP image configuration structure */  // SDK code snippet
struct kdp_img_conf_s {
    int32_t     image_col;
    int32_t     image_row;
    int32_t     image_ch;
    uint32_t    image_format;
    uint32_t    image_memory_address; // not use in DME mode
};

typedef struct kdp_img_conf_s kdp_img_conf_t;

/* KDP dme configuration structure */
struct kdp_dme_conf_s {
    int32_t     models_selection;
    int32_t     output_num;
    struct kdp_img_conf_s img_conf;
};

```

1.5 Node Result Layout in Memory

(1) For KL520, the results are in 16-byte aligned format for each row.

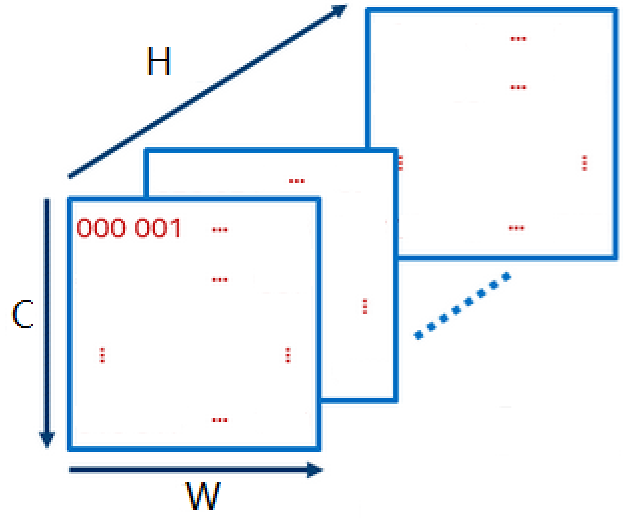
The following figure shows that if width mod 16 is equal to 7, then 9 bytes is inserted at the end of each row.





```
// SDK code snippet
/**
 * pad_up_16() - calculate the 16-bytes aligned width
 *
 * @a: the original width
 *
 * Return value:
 * >0 : aligned width
 */
inline static int pad_up_16(int a)
{
    return ceil((float)a / 16) * 16;
}
```

(2) For KL520, the data sequence of the CPU and Output node is HxCxW (Height*Channel*Width), which means that the data is stored in the sequence: row_1 of C1, row_1 of C2, ..., row_1 of Cn, row_2 of C1, row_2 of C2, ..., row_2 of Cn, row_H of C1, row_H of C2, ..., row_H of Cn.



(4) For KL520, the data type of CPU Node and Output Node is fixed-point (**1 byte for int8_t or 2 bytes for int16_t**).

The below SDK code snippet is to get the data_size.

```
data_size = (POSTPROC_OUTPUT_FORMAT(image_p) & BIT(0)) + 1;     /* 1 or 2 in bytes */
```

To do cpu operation or postprocess, the fixed-point data (**int8_t or int16_t**) need to be converted to float data with do_div_scale() function.

```
// SDK code snippet
/**
 * do_div_scale() - convert fixed point to float
 *
 * @v: fixed point value
 *
 * @div: 2 raised to the power of radix
 *
 * @scale: scale for fixed point
 *
 * Return float value
 */
static float do_div_scale(float v, int div, float scale)
{
    return ((v / div) / scale);
16. }
```

For example, the dimension of one output node for Tiny-YOLO-V3 is 7x255x7 (HxCxW), the actual memory size is 7x255x16. The dimension of another output node is 14x255x14 (HxCxW), the actual memory size is 14x255x16. The cpu_op and postprocess functions are based on this data layout.

1.6 Register Preprocess/CPU_Operation/Postprocess Functions

1.6.1 API Functions

API Functions	**Parameters**
int kdpio_cpu_op_register(int cpu_op, cpu_op_fn cof);	cpu_op: unique ID for this cpu_node cof: CPU operation function, such as **nearest_upsample_cpu**
int kdpio_pre_processing_register(int model_id, pre_proc_fn ppf);	model_id: unique ID for this model ppf: preprocess function, such as **kdp_preproc_inproc, preprocess_rgba_right_shift_yolo**
int kdpio_post_processing_register(int model_id, post_proc_fn ppf);	model_id: unique ID for this model ppf: postprocess function, such as **post_yolo_v3**
int kdp_preproc_inproc(int model_id, struct kdp_image_s *image_p);	model_id: unique ID for this model image_p: kdp_image_s struct with original image input

kdp_preproc_inproc() supports: Kneron preprocess method (RGB/256 - 0.5), Yolo preprocess method (RGB/256).

1.6.2 Allocate Unique ID for New Model and New CPU_OP

(1) Allocate new model ID in /common/include/model_type.h.

In SDK, a new model ID 19 is assigned for TINY-YOLO-V3. This ID is consistent with the model ID in fw_info.bin.

```
enum model_type {
    INVALID_ID,
    KNERON_FDSMALLBOX                   = 1,
    KNERON_FDANCHOR                     = 2,
    KNERON_FDSSD                        = 3,
    AVERAGE_POOLING                     = 4,
    KNERON_LM_5PTS                      = 5,
    KNERON_LM_68PTS                     = 6,
    KNERON_LM_150PTS                    = 7,
    KNERON_FR_RES50                     = 8,
    KNERON_FR_RES34                     = 9,
    KNERON_FR_VGG10                     = 10,
    KNERON_TINY_YOLO_PERSON             = 11,
    KNERON_3D_LIVENESS                  = 12,
    KNERON_GESTURE_RETINANET            = 13,
    TINY_YOLO_VOC                       = 14,
    IMAGENET_CLASSIFICATION_RES50       = 15,
    IMAGENET_CLASSIFICATION_RES34       = 16,
    IMAGENET_CLASSIFICATION_INCEPTION_V3= 17,
    IMAGENET_CLASSIFICATION_MOBILENET_V2= 18,
    TINY_YOLO_V3                        = 19,
};
```

(2) Allocate new CPU_OP ID in /ncpu/rtos2/cpu_ops_ex.h.

For a new CPU node, the first step is to get support by Compiler. Then compiler assigns a unique cpu_op_type to this type of CPU node when generating all_models.bin.

As there is a new cpu_op nearest_upsample for TINY-YOLO-V3, cpu_op_type 100 is assigned for it. This op_type is consistent with the one in all_models.bin.

```
/* Type of CPU Operations */
enum cpu_op_type {
    SOFTMAX,
    FLATTERN                = 1,
    DROPOUT                 = 2,
    NEAREST_UPSAMPLE        = 100,
    BILINEAR_UPSAMPLE       = 101,
    MYSTERY                 = 1000,
};
```

1.6.3 Struct kdp_image_s

The struct kdp_image_s is a key structure which exchange data and parameters between different API functions of NCPU: kdpio_set_model, kdpio_run_preprocess, kdpio_run_npu_op, kdpio_run_cpu_op, and kdpio_run_postprocess.

```
/* Structure of kdp_model_dim */
struct kdp_model_dim_s {
    /* CNN input dimensions */
    uint32_t    input_row;
    uint32_t    input_col;
    uint32_t    input_channel;
};

/* Structure of kdp_pre_proc_s */
struct kdp_pre_proc_s {
    /* input image in memory for NPU */
    uint32_t    input_mem_addr;
    int32_t     input_mem_len;

    /* Input working buffers for NPU */
    uint32_t    input_mem_addr2;
    int32_t     input_mem_len2;

    /* number of bits for input fraction */
    uint32_t    input_radix;

    /* Other parameters for the model */
    void        *params_p;
};

/* Structure of kdp_post_proc_s */
struct kdp_post_proc_s {
    /* output number */
    uint32_t    output_num;

    /* output data memory from NPU */
    uint32_t    output_mem_addr;
    int32_t     output_mem_len;

    /* result data memory from post processing */
    uint32_t    result_mem_addr;
    int32_t     result_mem_len;

    /* data memory for post processing */
    uint32_t    output_mem_addr3;

    /* output data format from NPU
     *     BIT(0): =0, 8-bits
     *             =1, 16-bits
     */
    uint32_t    output_format;

    /* output node parameter */
    struct out_node_s     *node_p;

    /* Other parameters for the model */
    void        *params_p;
};

/* Structure of kdp_cpu_op_s */
struct kdp_cpu_op_s {
    /* cpu op node parameter */
    struct cpu_node_s     *node_p;
};

/* KDP image structure */
struct kdp_image_s {
    /* Original image and model */
    struct kdp_img_raw_s   *raw_img_p;
    struct kdp_model_s     *model_p;

    int         model_id;
    /* Setup memory address contains information for input node, cpu node, output node */
    char        *setup_mem_p;

    /* Model dimension */
    struct kdp_model_dim_s  dim;

    /* Pre process struct */
    struct kdp_pre_proc_s   preproc;

    /* Post process struct */
    struct kdp_post_proc_s   postproc;

    /* CPU operation struct */
    struct kdp_cpu_op_s     cpu_op;
};
```

1.6.4 Register Preprocess Function

If the new model's preprocess method is not supported by **kdp_preproc_inproc**, the corresponding preprocess function need to be implemented and registered.

The preprocess function can be implemented in /ncpu/rtos2/pre_process_ex.c and registered in /ncpu/rtos2/main.c. The example code is in part (3) of Chapter 2.2.3.

(1) For Tiny-YOLO-V3, preprocess function **kdp_preproc_inproc** is registered for it in /ncpu/rtos2/main.c.

(2) For Tiny-YOLO-V3, another preprocess function **preprocess_rgba_right_shift_yolo** is also implemented in pre_process_ex.c as an example of how to write a new preprocess function. If using this function, the input image is RGBA image with dimensions (HWC, 224x224x4) and each R/G/B value ranges from 0 to 255.

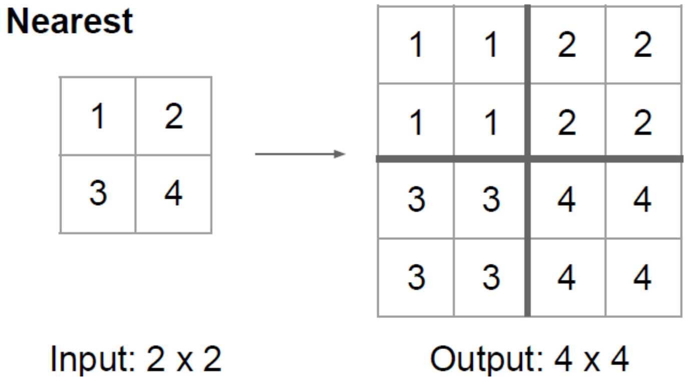
(3) Only 1 preprocess function is registered for each model.

1.6.5 Register CPU_Operation Function

By following the results data layout in KL520 and cpu_op algorithm, the corresponding CPU_Operation function can be implemented and registered in SDK.

(1) Nearest Upsample

The following figure shows the nearest upsampling with scale factor of 2. For Tiny-YOLO-V3, the cpu_op function nearest_upsample_cpu() is implemented in /ncpu/rtos2/cpu_ops_ex.c and registered in /ncpu/rtos2/main.c. The example code is in Chapter 2.2.4.



1.6.6 Register Postprocess Function

Different models have their specific postprocess method to output the detection results. By following results layout in KL520 and the postprocess algorithm, the corresponding postprocess function can be implemented and registered in SDK.

If bypass postprocess, SDK output the raw results of all output nodes. And no postprocess function is needed to be registered for any models.

For Tiny-YOLO-V3, the postprocess function post_yolo_v3() is implemented in /ncpu/rtos2/post_processing_ex.c and registered in /ncpu/rtos2/main.c. The example code is in Chapter 2.2.5.

1.7 Inference with New Models by Calling APIs in Host Library (Serial Mode)

After preparing the input files and code changes to support new model in SDK, call APIs in host library to run test for new model in DME mode.

Input Files	**Description**	**How to generate**
image.bin	Image input binary file	Convert jpg/png file into binary file (RGB565/RGBA/YCbCr422)
all_models.bin	Model data file	Generate by toolchain

fw_info.bin	Firmware information data file	Generate by toolchain
config.bin	Configuration binary file	Use script dme_conf_generator.py to generate

(1) In order to get the matching results between toolchain and SDK, the preprocessed RGBA image input need to be generated by img_preprocess.py in toolchain. Use ‘python img_preprocess.py -h’ for more details.

1.8 Results to Host

1.8.1 Detection Results

The detection results layout follows the definition of struct kdp_app_dme_res_t in kdp_app_dme.h: TOTAL_CLASS (4 bytes, int) + BOXES_COUNT (int) + BOXES_COUNT * BOXES (x1 (float), y1 (float), x2 (float), y2 (float), score (float), class (int)).

```
typedef struct kdp_app_bounding_box_s {
    float x1;           // top-left corner:  x
    float y1;           // top-left corner:  y
    float x2;           // bottom-right corner:  x
    float y2;           // bottom-right corner:  y

    float score;        // probability score
    int class_num;      // class # (of many) with highest probability
} kdp_app_bounding_box_t;

#define OBJECT_DETECTION_MAX    80

struct kdp_app_dme_res_s {
    uint32_t    class_count;            // total class count
    uint32_t    box_count;              /* boxes of all classes */
    kdp_app_bounding_box_t  boxes[OBJECT_DETECTION_MAX];       /* [box_count] */
} kdp_app_dme_res_t;
```

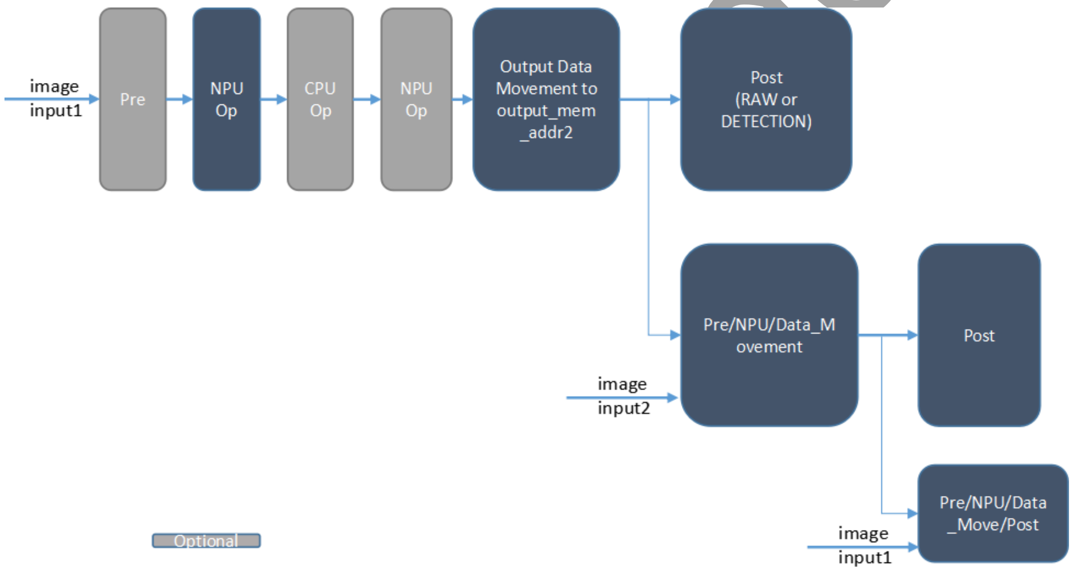
1.8.2 RAW Results

The RAW results layout is: OUTPUT_NUM (4 bytes, int) + OUTPUT_NUM * HCW ([height (int), channel, width, h2, c2, w2, ...]) + RAW_DATA (RAW1 (float), RAW2, ...).

1.9 Parallel Processing (Parallel Mode)

This chapter focuses on the introduction of parallel processing with 2 input images (RAW images or preprocessed RGBA images) and 2 output memory buffers.

1.9.1 Parallel Processing Overview Chart



1.9.2 Limitations

(1) Current parallel processing part in parallel processing example SDK is an example of how to do parallel processing and get the timing results. The APIs in host library and logic of sending the postprocessing results to host is not implemented.

(2) Parallel processing in SDK does not always bring the performance improvement due to the presence of CPU operation in preprocessing (such as right-shift), CPU node, time-consuming data movement, etc.

(3) Current example design is to support running parallel processing with 1 model repeatedly.

1.9.3 Input and Output Memory

Memory	**To Store**	**Defined in**
input_mem_addr	Preprocessed RGBA input1	struct kdp_model_s of ipc.h
input_mem_addr2	Preprocessed RGBA input2	struct scpu_to_ncpu_s of ipc.h
KDP_DDR_BASE_IMAGE_BUF	Raw image input1	kdp_ddr_table.h
KDP_DDR_BASE_IMAGE_BUF2	Raw image input2	kdp_ddr_table.h
output_mem_addr2	Previous output data of all output nodes	struct scpu_to_ncpu_s of ipc.h

Output_mem_addr2 is used as new base address in postprocessing when using parallel processing. The actual addresses of Output nodes are calculated by adding output_mem_addr2 and offsets. The offsets are calculated by subtracting model buffer address from output node address. The below SDK code snippet shows how to copy data of all output nodes to output_mem_addr2.

```
static void npu_out_data_move(void *dst_p)
{
    /* Move data out from npu output buffer for parallel handling */
    int idx = 0;
    int len, data_size, grid_w, grid_h, grid_c, grid_w_bytes_aligned;
    int offset = sizeof(struct out_node_s);
    int8_t *src_p;
    struct out_node_s *out_p;
    data_size = (POSTPROC_OUTPUT_FORMAT(pp_img_p) & BIT(0)) + 1;      /* 1 or 2 in bytes */

    for (idx = 0; idx < POSTPROC_OUTPUT_NUM(pp_img_p); idx++) {
        out_p = (struct out_node_s *)((uint32_t)POSTPROC_OUT_NODE(pp_img_p) + idx * offset);
        if (idx != 0) {
            offset = idx * sizeof(struct out_node_s);
            out_p = (struct out_node_s *)((uint32_t)POSTPROC_OUT_NODE(pp_img_p) + offset);
        } else {
            out_p = POSTPROC_OUT_NODE(pp_img_p);
        }
        src_p = (int8_t *)OUT_NODE_ADDR(out_p);
        grid_w = OUT_NODE_COL(out_p);
        grid_h = OUT_NODE_ROW(out_p);
        grid_c = OUT_NODE_CH(out_p);

        len = grid_w * data_size;
        grid_w_bytes_aligned = pad_up_16(len);

        int total = grid_h*grid_c*grid_w_bytes_aligned;
        int mem_offset = OUT_NODE_ADDR(out_p) - MODEL_BUF_ADDR(pp_img_p);  // calculate offset
        memcpy((int8_t *)((uint32_t)dst_p + mem_offset), src_p, total);     // copy to new address
    }
    /* Notify scpu now */
    scu_ipc_trigger_to_scpu_int();
}
```

1.9.4 Struct scpu_to_ncpu_s

This struct contains the global parameters to be passed from SCPU to NCPU when enabling parallel processing: parallel_start, parallel_total, parallel_count, input_mem_addr2, output_mem_addr2, etc.

```
/* Structure of sCPU->nCPU Message */
struct scpu_to_ncpu_s {
    ...

    /* Images being processed via IPC */
    int32_t             active_img_index;
    struct kdp_img_raw_s raw_images[IPC_IMAGE_MAX];

    /* parallel processing */
    uint32_t    parallel_start;  // start parallel processing or not (1 for start, 0 for not start)
    uint32_t    parallel_total;  // total times to run parallel processing, 10 in SDK
    uint32_t    parallel_count;  // count the finished cycle (from pre to post)

    /* Input/Output working buffers for NPU */
    uint32_t    input_mem_addr2; // second RGBA input buffer if using parallel processing
    int32_t     input_mem_len2;

    uint32_t    output_mem_addr2; // second output buffer is using parallel processing
    int32_t     output_mem_len2;

    uint32_t    output_mem_addr3;
};
```

1.9.5 Struct kdp_img_raw_s

(1) This struct contains the parameters to be used in NCPU when enabling parallel processing: parallel_switch_input, parallel_start, parallel_total, parallel_count, tick_start_first_pre, and tick_end_last_post.

(2) The method to generate configuration for parallel processing is shown in chapter 1.4. Bit 27 in image_format is used to enable or disable parallel processing.

Parameters	**Description**
parallel_switch_input	Parameter for ping-pong operations to switch between input1 and input2
parallel_start	Start parallel processing or not (1 for start, 0 for not start)
parallel_total	Total times to run parallel processing (10 in SDK)
parallel_count	Count the finished cycle (from preprocessing to postprocessing)
tick_start_first_pre	Record the tick for the start of first preprocessing
tick_end_last_post	Record the tick for the end of last postprocessing

```
* Raw image structure */
struct kdp_img_raw_s {
    ...

    /* Test: NCPU parallel processes */
    uint32_t    parallel_switch_input;   // switch to use input1 or input2
    uint32_t    parallel_start;   // start parallel processing or not (1 for start, 0 for not start)
    uint32_t    parallel_count;   // total times to run parallel processing, 10 in SDK
    uint32_t    parallel_total;   // count the finished cycle (from pre to post)
    uint32_t    tick_start_first_pre;    // tick for the start of first preprocessing
    uint32_t    tick_end_last_post;      // tick for the end of last postprocessing
};
```

1.9.6 Code Change for Existing Postprocess Function

The change in postprocess function is the method to calculate the address of output node. This method is compatible with serial mode. The example code is shown as below.

```
//src_p         pointer to the output data
//image_p       pointer to kdp_image_s
//out_p         pointer to out_node_s
int8_t *src_p = (int8_t *)(POSTPROC_OUTPUT_MEM_ADDR(image_p) + OUT_NODE_ADDR(out_p) - MODEL_BUF_ADDR(image_p));
```

1.9.7 Input Files for Parallel Processing

2 input binary files are needed when running parallel preprocessing.

Input Files	**Description**	**How to generate**
image1.bin and image2.bin	Image input binary files	Convert jpg/png file into binary files (RGB565/RGBA/YCbCr422)
all_models.bin	Model data file	Generate by toolchain
fw_info.bin	Firmware information data file	Generate by toolchain
config.bin	Configuration binary file	Use script dme_conf_generator.py to generate

2. DME Example Code

This chapter focuses on the explanation of example code for Tiny-YOLO-V3. Tiny-YOLO-V3 is a model for multiple-classes object detection. The following sample code implements the object detection function for 80 classes.

2.1 SCPU

2.1.1 cmd_parser Running in Host Communication Thread

The parser function is to parse different types of commands from Host, write firmware information, model data, and input images to assigned memory address, launch application, and send back the output results to Host through USB interface.

DME Commands	**Description**
CMD_DME_START	Send START command with FW_INFO size, FW_INFO_DATA (fw_info.bin) and MODEL_DATA (all_models.bin) from Host to KL520 through USB interface
CMD_DME_CONFIG	Send CONFIG command with output_num, image_input_w/h/c, and image_format from Host to KL520 through USB interface
CMD_DME_SEND_IMAGE	Send IMAGE command with IMAGE_DATA from Host to KL520 through USB interface, and do inference
CMD_ACK_NACK	Send ACK command and retrieve back output results from KL520 to Host through USB interface

(1) CMD_DME_START

After receiving CMD_DME_START from Host, SCPU read firmware information in the message from Host, and write it into the assigned address of DDR memory; then SCPU read the model data from USB interface, and write it into the assigned address of DDR memory: firmware info to KDP_DDR_MODEL_INFO_TEMP, model data to command start address.

```c
//Code snippet from cmd_parser() in host_com.c
case CMD_DME_START:
{
    // NOTE: Need to determine if DME structure needs to be set up first before processing images
    info_msg("[Host_com] Start Object Detection in DME mode.\r\n");
    //Get size of inference mode (4 bytes)
    uint32_t model_size = msgcmd_arg->addr;

    // Load firmware info. The address and size of firmware info is inside the command
    memcpy((uint32_t*)KDP_DDR_MODEL_INFO_TEMP, &(msgcmd_arg->len), msghdr->mSize - 4);

    uint32_t cmd_addr_offset = 9*sizeof(uint32_t); //cmd addr offset in fw_info
    uint32_t cmd_start_addr = *(uint32_t*)(KDP_DDR_MODEL_INFO_TEMP + cmd_addr_offset);

    if (model_size <= 0) {
        msgrsp_arg->error = MSG_DME_DATA_FAIL;
        msgrsp_arg->bytes = CMD_DME_START;
        msg_pack(CMD_ACK_NACK, NULL, 0);
        break;  // early abort
    }

    //Load setup, weights and commands
    ret = host_mem_read(cmd_start_addr, model_size);  // specify how many bytes
    if (ret == model_size) {
        APP_id = 3;  // enter DME mode
        msgrsp_arg->error = NO_ERROR;
        msgrsp_arg->bytes = ret;
        msg_pack(CMD_DME_START, NULL, 0);
    } else {
        msgrsp_arg->error = MSG_DME_DATA_FAIL;
        msgrsp_arg->bytes = 0;
        msg_pack(CMD_DME_START, NULL, 0);
    }

    kdp_app_dme_mode(1/*is_dme*/);
    break;
}
```

(2) CMD_DME_CONFIG

After receiving CMD_DME_CONFIG from Host, SCPU read configuration information in the message from Host, and copy it to the global variable struct kdp_dme_conf_s.

```c
//Code snippet from cmd_parser() in host_com.c
```

```
case CMD_DME_CONFIG:
{
    info_msg("[Host_com] DME Config.\r\n");
    if (APP_id == 3) {
        memcpy(&dme_conf, &(msgcmd_arg->addr), sizeof(struct kdp_dme_conf_s));
        msgrsp_arg->error = NO_ERROR;
    }
    else
        msgrsp_arg->error = MSG_APP_NOT_CONFIG;  // not in DME mode
    msgrsp_arg->bytes = 0;
    msg_pack(CMD_DME_CONFIG, NULL, 0);  // use zero payload response
    break;
}
```

(3) CMD_DME_SEND_IMAGE

After receiving CMD_DME_SEND_IMAGE from Host, SCPU read image input from USB interface, and copy it to different assigned address of DDR memory according to configuration: RGBA image input to DDR_MEM_BASE; other types of image input to KDP_DDR_BASE_IMAGE_BUF. Then SCPU allocates memory for final results in advance (the size of allocated memory is greater or equal to the exact size of results), launch kdp_app_dme application to do inference, and send back the exact size of results to Host. Code snippet from cmd_parser() in host_com.c is shown as below.

```
case CMD_DME_SEND_IMAGE:
{
    dbg_msg("Received cmd: CMD_DME_SEND_IMAGE\n");

    if ((0 == dme_conf.output_num) || (APP_id != 3)) { // check if we have DME structures set up?
        if (APP_id == 3)
            msgrsp_arg->error = MSG_DME_NO_CONFIG;  // image processing not set up yet
        else
            msgrsp_arg->error = MSG_APP_NOT_CONFIG;  // not in DME mode
        msgrsp_arg->bytes = CMD_DME_SEND_IMAGE;  // add source id
        msg_pack(CMD_ACK_NACK, NULL, 0);
        break;  // early abort
    }

    if (!(dme_conf.img_conf.image_format & IMAGE_FORMAT_BYPASS_PRE)) {
        // write RGB565 image into input base addr1 of DDR
        ret = host_mem_read(KDP_DDR_BASE_IMAGE_BUF, msgcmd_arg->addr);  // specify how many bytes
        // write RGB565 image into input base addr2 of DDR if parallel
        if (dme_conf.img_conf.image_format & IMAGE_FORMAT_PARALLEL_PROC) {
            ret = host_mem_read(KDP_DDR_BASE_IMAGE_BUF2, msgcmd_arg->addr);  // specify how many bytes
        }
    }  else {
        // write RGBA image into input addr1 of DDR
        ret = host_mem_read(DDR_MEM_BASE, msgcmd_arg->addr);  // specify how many bytes
        // write RGBA image into input addr2 of DDR if parallel
        if (dme_conf.img_conf.image_format & IMAGE_FORMAT_PARALLEL_PROC) {
            ret = host_mem_read(get_out_com_p()->input_mem_addr2, msgcmd_arg->addr);  // specify how many bytes
        }
    }

    uint32_t raw_len = 0; //For allocating memory in advance
    uint32_t output_num = dme_conf.output_num;

    // Support parallel processing with 2 inputs, and reset the execution times to 10
    if (dme_conf.img_conf.image_format & IMAGE_FORMAT_PARALLEL_PROC) {
        get_out_com_p()->parallel_start = 1;
        get_out_com_p()->parallel_total = 10;
        get_out_com_p()->parallel_count = 0;
    } else {
        get_out_com_p()->parallel_total = 0;
    }

    if (dme_conf.img_conf.image_format & IMAGE_FORMAT_RAW_OUTPUT) {
        //Get output length in fw_info
        uint32_t output_len = *(uint32_t*)(KDP_DDR_MODEL_INFO_TEMP + 6*sizeof(uint32_t));
        raw_len = (1 + 3*output_num + output_len)*sizeof(uint32_t);
        if (p_raw_out_s == 0) {
            p_raw_out_s = kdp_ddr_malloc(raw_len);
        }
        ret = kdp_app_dme((kdp_app_dme_res_t *)p_raw_out_s, dme_conf);
    } else {
        if(p_od_out_s == 0) {
            p_od_out_s = (kdp_app_dme_res_t *) kdp_ddr_malloc(
                    sizeof(kdp_app_dme_res_t));
        }
        ret = kdp_app_dme(p_od_out_s, dme_conf);
    }

    //For sending results to HOST
    if (dme_conf.img_conf.image_format & IMAGE_FORMAT_RAW_OUTPUT) {
        int output_len = 0;
        for (int i = 0; i < output_num; i++){
            int h = *(uint32_t *)(p_raw_out_s + (3*i+1)*sizeof(uint32_t));
            int c = *(uint32_t *)(p_raw_out_s + (3*i+2)*sizeof(uint32_t));
            int w = *(uint32_t *)(p_raw_out_s + (3*i+3)*sizeof(uint32_t));
            output_len = output_len + h * c * w;
        }

        //output_num+HCW+H2C2W2+DATA1+DATA2
        final_len = (1 + 3*output_num + output_len)*sizeof(uint32_t);
    } else {
        int det_num = p_od_out_s->box_count;
        final_len = 2*sizeof(uint32_t) + det_num * sizeof(kdp_app_bounding_box_t);
    }
    msgrsp_arg->error = NO_ERROR;
    msgrsp_arg->bytes = final_len;

    //Msg for the length of raw results
    msg_pack(CMD_DME_SEND_IMAGE, NULL, 0);
    break;
}
```

(4) CMD_ACK_NACK

After receiving CMD_ACK_NACK from Host, SCPU send back the results to Host. Depending on the configuration in CMD_DME_CONFIG step, detection results or raw results are sent back to Host.

```
//Code snippet from cmd_parser() in host_com.c
case CMD_ACK_NACK:
{
    dbg_msg("[Host_com] Received ACK/NACK\r\n");
    // Send results back to host
    if (msgcmd_arg->addr) {
        no_host_reply();
        break;  // if NACK, ignore the command and no reply
    }
    // We don't enforce Source ID check yet, since only DME is using this
    if (dme_conf.img_conf.image_format & IMAGE_FORMAT_RAW_OUTPUT){
        data_write((u8 *) p_raw_out_s, final_len);
    } else {
        data_write((u8 *) p_od_out_s, final_len);
    }
    break;
}
```

2.2 NCPU

2.2.1 SCPU-to-NCPU Communication Thread

This thread is to register preprocess/cpu_op/postprocess function, wait for all events from SCPU, configure kdp_image_s struct, set model, and run preprocess/npu_op.

(1) Part 1 of scpu_comm_thread

Preprocess/cpu_op/postprocess functions are registered in this part of code.

```
//SCPU Communication and Main Thread
void scpu_comm_thread(void *argument)
{
    int32_t flags;
    struct kdp_img_raw_s *raw_img_p;
    struct kdp_model_s *model_p;
    int model_slot_index, image_index;

    /* Init kdpio lib and interrupt/status */
    kdpio_init();

    /* Register preprocess function dynamically */
    // Register kdp_preproc_inproc for Tiny-YOLO-V3
    kdpio_pre_processing_register(TINY_YOLO_V3, kdp_preproc_inproc);
    /* An example to register preprocess function in pre_processing_ex.c */
    //kdpio_pre_processing_register(TINY_YOLO_V3, preprocess_rgba_right_shift_yolo);
    /* Register cpu_op function dynamically */
    kdpio_cpu_op_register(NEAREST_UPSAMPLE, nearest_upsample_cpu);
    /* Register post_processing function dynamically */
    kdpio_post_processing_register(TINY_YOLO_V3, post_yolo_v3);

    // Enable ARM Snoop Control Unit (SCU)'s inter-processor communication to SCPU interrupt
    scu_ipc_enable_to_scpu_int();

    // Enable SCPU interrupt request
    NVIC_EnableIRQ((IRQn_Type)SCPU_IRQ);
    // Enable NPU interrupt request
    NVIC_EnableIRQ((IRQn_Type)NPU_NPU_IRQ);

    // Set NPU to SCPU ID and status
    out_comm_p->id = NCPU2SCPU_ID;
    out_comm_p->status = STATUS_OK;

    critical_msg("ncpu: Ready!\n");

    /* Let's run tickless */
    OS_Tick_Disable();
```

(2) Part 2 of scpu_comm_thread

NCPU wait for all events from SCPU infinitely. If the event flag is FLAG_SCPU_COMM_ISR, NCPU configures kdp_image_s struct, set model, and run preprocess/npu_op.

```
    /* Waiting for all events from SCPU */
    for (;;) {
        // Wait forever for Thread Flags of SCPU thread to become signaled
        flags = osThreadFlagsWait(FLAG_SCPU_ALL, osFlagsWaitAny, osWaitForever);
        // Clear thread flags
        osThreadFlagsClear(flags);

        // Check the flags to determine whether it is a Interrupt Service Routines from SCPU
        if (flags & FLAG_SCPU_COMM_ISR) {
            dbg_msg("ncpu: got IPC interrupt\n");

            // Check the command from SCPU. If it is CMD_RUN_NPU, execute:
            // - Set model and write weights to DMA
            // - Preprocess image input
            // - Start NPU
            if (in_comm_p->cmd == CMD_RUN_NPU) {
                // Enable os tick for profiling or disable for saving power
                if (log_get_level_ncpu() >= LOG_PROFILE)
                    OS_Tick_Enable();
                else
                    OS_Tick_Disable();

                // Get the model index. If there is only 1 model, the index is 0.
                model_slot_index = in_comm_p->model_slot_index;
                // Get the model
                model_p = &in_comm_p->models[model_slot_index];

                dbg_msg("NPU: model_slot_index=%d, NPU output mem addr=0x%x, len=%d\n", model_slot_index, (uint32_t)model_p->output_mem_addr, model_p->output_mem_len);
                dbg_msg("NPU: input_mem_addr = 0x%x, len=0x%x, cmd_mem_addr=0x%x, len=0x%x, wt_addr=0x%x, wt_len=0x%x, setup_addr=0x%x, setup_len=0x%x\n",
                   (uint32_t)model_p->input_mem_addr, (uint32_t)model_p->input_mem_len,
                   (uint32_t)model_p->cmd_mem_addr, (uint32_t)model_p->cmd_mem_len,
                   (uint32_t)model_p->weight_mem_addr, (uint32_t)model_p->weight_mem_len,
                   (uint32_t)model_p->setup_mem_addr, (uint32_t)model_p->setup_mem_len);

                // Get the image index. If there is only 1 model, the index is 0.
                image_index = in_comm_p->active_img_index;
                // Get the image input
                raw_img_p = &in_comm_p->raw_images[image_index];
                npu_img_p = &image_s[npu_img_idx];
                npu_img_idx = !npu_img_idx;

                npu_img_p->raw_img_p = raw_img_p;
                npu_img_p->model_id = in_comm_p->models_type[model_slot_index];
                // Set the input memory addr and input memory length for kdp_image_s
                // For parallel processing
                if (in_comm_p->output_mem_len2 && (RAW_FORMAT(npu_img_p) & IMAGE_FORMAT_BYPASS_PRE)) {
                    if (0 == RAW_PARALLEL_SWITCH_INPUT(npu_img_p)) {
                        PREPROC_INPUT_MEM_ADDR(npu_img_p) = model_p->input_mem_addr;
                        PREPROC_INPUT_MEM_LEN(npu_img_p) = model_p->input_mem_len;
                    } else {
                        PREPROC_INPUT_MEM_ADDR(npu_img_p) = in_comm_p->input_mem_addr2;
                        PREPROC_INPUT_MEM_LEN(npu_img_p) = in_comm_p->input_mem_len2;
                    }
                    RAW_PARALLEL_SWITCH_INPUT(npu_img_p) = !RAW_PARALLEL_SWITCH_INPUT(npu_img_p);
                } else {
                    PREPROC_INPUT_MEM_ADDR(npu_img_p) = model_p->input_mem_addr;
                    PREPROC_INPUT_MEM_LEN(npu_img_p) = model_p->input_mem_len;
                }

                // Set the result memory addr and result memory length for kdp_image_s.
                // This memory space is used to output result from NPU to NCPU and SCPU
                POSTPROC_RESULT_MEM_ADDR(npu_img_p) = raw_img_p->results[model_slot_index].result_mem_addr;
                POSTPROC_RESULT_MEM_LEN(npu_img_p) = raw_img_p->results[model_slot_index].result_mem_len;

                dbg_msg("NPU: model_type = %d\n",npu_img_p->model_id);

                out_comm_p->img_results[image_index].seq_num = raw_img_p->seq_num;
                out_comm_p->img_results[image_index].status = IMAGE_STATE_ACTIVE;

                // Set the second input memory addr and input memory length for kdp_image_s
                // This memory space is used for parallel postprocessing which has not been supported.
                PREPROC_INPUT_MEM_ADDR2(npu_img_p) = in_comm_p->input_mem_addr2;
                PREPROC_INPUT_MEM_LEN2(npu_img_p) = in_comm_p->input_mem_len2;

                // Set the third output memory addr for kdp_image_s
                // This memory space is used by postprocess function which need a large space (> 20k bytes) to store
                // the parameters.
                POSTPROC_OUTPUT_MEM_ADDR3(npu_img_p) = in_comm_p->output_mem_addr3;

                // Set model to be used for processing the input image
                kdpio_set_model(npu_img_p, model_p);

                // Run preprocess
                kdpio_run_preprocess(npu_img_p);
                // Run NPU Operation
                // - If there is no CPU Node, NPU finishes all NPU Operations and get ready for all Outpu
t Node
                // - If there is CPU Node, NPU finishes all NPU Operations before the first CPU Node.
                kdpio_run_npu_op(npu_img_p);
            }

            // SCU clear SCPU interrupt
            scu_ipc_clear_from_scpu_int();
            // Nested Vectored Interrupt Controller (NVIC) clear SCPU interrupt from pending
            NVIC_ClearPendingIRQ((IRQn_Type)SCPU_IRQ);
            // NVIC enable SCPU interrupt
            NVIC_EnableIRQ((IRQn_Type)SCPU_IRQ);
        }
    }
}
```

2.2.2 NCPU-to-NPU Communication Thread

This thread is to wait for all events from NPU and run cpu_op/postprocess.

NCPU wait for all events from NPU infinitely. If the event flag is FLAG_NPU_COMM_ISR, NCPU run cpu_op/postprocess, and triggers SCPU interrupt after postprocessing.

```
//NCPU Communication Thread
void npu_comm_thread(void *argument)
{
    int32_t flags;
    int rc, i;

    /* Waiting for all events from NPU */
    for (;;) {
        // Wait forever for Thread Flags of NPU thread to become signaled
        flags = osThreadFlagsWait(FLAG_NPU_COMM_ISR, osFlagsWaitAny, osWaitForever);
        osThreadFlagsClear(flags);

        // Check the flags to determine whether it is a Interrupt Service Routines from NPU
        if (flags & FLAG_NPU_COMM_ISR) {
            // Run CPU Operation
            // - If there is no CPU Node, this function return RET_NO_ERROR
            // - If there is CPU Node, NCPU finishes the CPU Operation, continue NPU Operations,
            //   and return RET_NEXT_NPU.
            rc = kdpio_run_cpu_op(npu_img_p);
            if (rc == RET_NO_ERROR) {
                i = npu_img_p->raw_img_p->ref_idx;

                dbg_msg("npu out: img[%d] state: %d -> %d\n", i, out_comm_p->img_results[i].status, IMAGE_STATE_NPU_DONE);

                out_comm_p->img_index_done = i;
                out_comm_p->img_results[i].status = IMAGE_STATE_NPU_DONE;
                // Assign the second output memory addr to kdp_image_s
                // This memory space is used for parallel postprocessing which has not been supported.
                POSTPROC_OUTPUT_MEM_ADDR(npu_img_p) = in_comm_p->output_mem_addr2;
                pp_img_p = npu_img_p;

                // NCPU run postprocess
                rc = kdpio_run_postprocess(pp_img_p, npu_out_data_move);

                if (rc == RET_NO_ERROR) {
                    i = pp_img_p->raw_img_p->ref_idx;

                    dbg_msg("npu out: img[%d] state: %d -> %d\n", i, out_comm_p->img_results[i].status, IMAGE_STATE_DONE);

                    out_comm_p->img_index_done = i;
                    out_comm_p->img_results[i].status = IMAGE_STATE_DONE;
                    // SCU trigger SCPU interrupt
                    scu_ipc_trigger_to_scpu_int();
                }
            }

            NVIC_ClearPendingIRQ((IRQn_Type)NPU_NPU_IRQ);
            NVIC_EnableIRQ((IRQn_Type)NPU_NPU_IRQ);
        }
    }
}
```

2.2.3 Pre-processing

Preprocess function is registered in scpu_comm_thread(). Kdp_image_s struct provides all information needed for preprocessing: image format, dimension, cropping parameters, and padding parameters of input raw image; dimension of input node, radix, scale, input memory address, and output memory address.

(1) kdp_preproc_inproc

Kneron provides preprocess API function kdp_preproc_inproc(). The example of how to register it is shown in 2.2.1.

```
/**
 * kdp_preproc_inproc() - Kneron preprocessing procedure
 *
 * @image_p: pointer to struct kdp_image with buffer of raw image
 *          and dimension, and data for pre/cpu/post processing.
 *
 * @model_id: the model id this function was registered for
 *
 * This is a preprocess function which uses Kneron NPU to accelerate the
 * processing and uses Kneron NCPU to do right-shift. It can take parameters passed in to do resize, crop, padding with NPU and do right-shift with NCPU.
 * normalization like -128 and rotation.
 *
 * Return value:
 * 0 : success
 * <0 : error
 */
int kdp_preproc_inproc(int model_id, struct kdp_image_s *image_p);
```

The preprocessing operations of kdp_preproc_inproc is in the order of crop (original format, such as RGB565/YCBCR422, by NPU), format (original format to RGBA8888, by NPU), resize (RGBA8888, by NPU), left-shift if needed (RGBA8888, by NPU, not implemented in SDK), subtract (RGBA8888, by NPU), pad (RGBA8888, by NPU), right-shift if needed (RGBA8888, by NCPU).

(2) Use kdp_preproc_inproc to do preprocess

For a model using Kneron preprocess (RGB/256 - 0.5), the input pixel value range is -0.5~0.5 (x/256 - 0.5, x ranges from 0~255). After compiling with compiler, we get that the input_radix is 8 and input_scale is 1.0 for this model. If using NPU to do preprocessing with RGB data, the data is converted to fixed point with the formula **(x/256 - 0.5) * (2^radix) * scale** (equals to x - 128).

For a model using Yolo preprocess (RGB/256), the input pixel value range is 0~1 (x/256, x ranges from 0~255). After compiling with compiler, we get that the input_radix is 7 and input_scale is 1.0 for this model. If using NPU/NCPU to do preprocessing with RGB data, the data is converted to fixed point with the formula **(x/256) * (2^radix) * scale** (equals to x/2).

The following example shows how to convert 640x480x3 RGB565 data to 224x224x4 RGBA data with Kneron preprocess method.

- Generate configuration file with below configuration. This configuration is used for preprocessing RGB565 input.

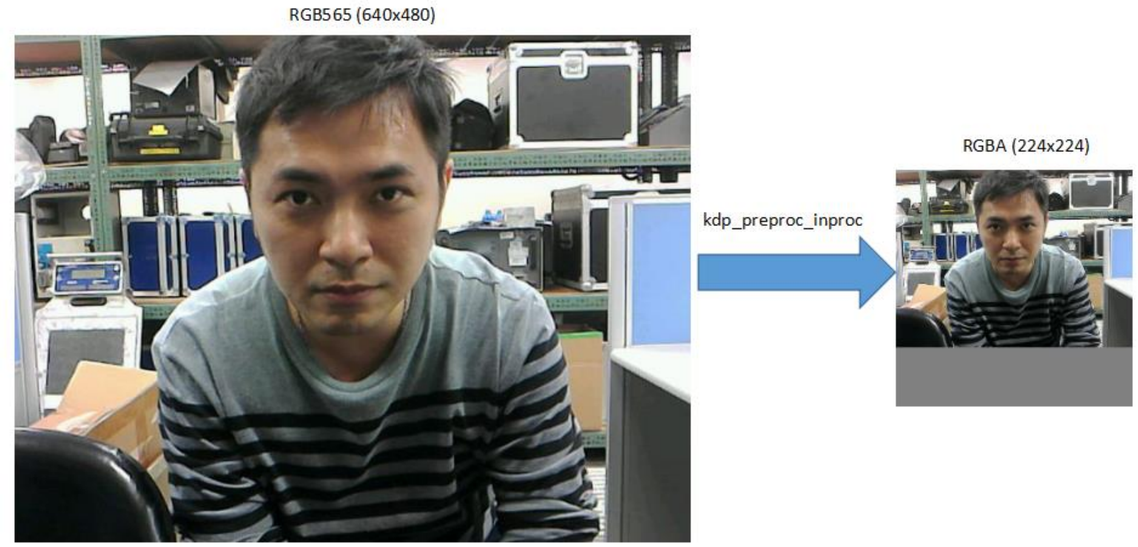
```
#python code snippet
    output_num   = 1          # (one output for this model)
    image_col    = 640        # (input width)
    image_row    = 480        # (input height)
    image_ch     = 3          # (input channel)
    image_format = 0x80000060 # (bit31 - subtract 128, 0x60 - RGB565 input, not bypass pre/post)
```

- Register preprocess function in scpu_comm_thread() for this model in SDK. The input width and height is 224x224 for MODEL_XXX.

```
/* Register preprocess function dynamically */
kdpio_pre_processing_register(MODEL_XXX, kdp_preproc_inproc);
```

- After sending RGB565 data to KL520 and starting inference, the 640x480 RGB565 data is converted to 224x224 RGBA data by kdp_preproc_inproc.

The preprocessing operations of kdp_preproc_inproc is in the order of crop (top_crop: 0, bottom_crop: 0, left_crop:0, right_crop: 0), format (RGB565 to RGBA8888), resize (640x480 to 224x168 and keep aspect ratio), left-shift (shift: 0), subtract (-128), pad (pad 56 rows with 128), right-shift (shift: 0). The RGBA image below is just for illustration.



(3) preprocess_rgba_right_shift_yolo

preprocess_rgba_right_shift_yolo() is an example of how to implement a customized preprocess function. The example of how to register it is shown in 2.2.1.

```
/**
 * preprocess_rgba_right_shift_yolo() - right shift RGBA values for 1 bit
 *
 * @model_id: the model id this function was registered for
 *
 * @image_p: pointer to struct kdp_image with buffer of raw image
 *           and dimension, and data for pre/cpu/post processing.
 *
 * This is a preprocess example function which uses Kneron NCPU to do right-shift for the input image (HeightxWidthxChannel: 224x224x4).
 *
 * Return value:
 * 0 : success
 * <0 : error
 */
int preprocess_rgba_right_shift_yolo(int model_id, struct kdp_image_s *image_p)
{
    int out_row, out_col;
    int input_radix, bit_shift;
    uint8_t *src_p, *dst_p;

    //Get model input row and column size
    out_row = DIM_INPUT_ROW(image_p);
    out_col = DIM_INPUT_COL(image_p);

    //Get model input radix
    input_radix = PREPROC_INPUT_RADIX(image_p);
    //Calculate the number of shift bit
    bit_shift = 8 - input_radix; // 1 byte (8 bits) for every R/G/B/A data

    //Get source RGBA input address: R/G/B value ranges from 0 to 255
    src_p = (uint8_t *)RAW_IMAGE_MEM_ADDR(image_p);
    //Get destination RGBA address
    dst_p = (uint8_t *)PREPROC_INPUT_MEM_ADDR(image_p);
    //Do right-shift
    pre_proc_unsign_right_shift(src_p, dst_p, out_row, out_col, bit_shift);

    return 0;
}
```

(4) Use preprocess_rgba_right_shift_yolo to do preprocess

For a model using Yolo preprocess (RGB/256), the input pixel value range is 0~1 (x/256, x ranges from 0~255). After compiling with compiler, we get that the input_radix is 7 and input_scale is 1.0 for this model. If using NCPU to do preprocessing (only right-shift) with RGBA data (224x224x4), the data is converted to fixed point with the formula **(x/256) * (2^radix) * scale** (equals to x/2).

The following example shows how to convert 224x224x4 RGBA data (R/G/B: 0~255) to 224x224x4 RGBA data (R/G/B: 0~128).

- Generate configuration file with below configuration. This configuration is used for preprocessing RGBA input (R/G/B: 0~255).

```
#python code snippet
    output_num    = 2         # (two outputs for this model)
    image_col     = 224       # (input width)
    image_row     = 224       # (input height)
    image_ch      = 4         # (input channel)
    image_format = 0x00000000 # (0x00 - RGBA input, not bypass pre/post)
```

- Register preprocess function in scpu_comm_thread() for YOLO in SDK. The input width and height is 224x224 for TINY-YOLO-V3. Only one preprocess function is registered for each model.

```
/* Register pre_processing function dynamically */
//kdpio_pre_processing_register(TINY_YOLO_V3, kdp_preproc_inproc);
/* An example to register preprocess function in pre_processing_ex.c */
kdpio_pre_processing_register(TINY_YOLO_V3, preprocess_rgba_right_shift_yolo);
```

- After sending RGBA data (R/G/B: 0~255) to KL520 and starting inference, the 224x224x4 RGBA input data is converted to 224x224x4 RGBA data (R/G/B: 0~128) by preprocess_rgba_right_shift_yolo.

2.2.4 CPU_OP

CPU_OP function is registered in scpu_comm_thread(). The implementation of this function follows the rule of 16-bytes alignment and HCW data layout. Kdp_image_s struct provides all information needed for cpu_op: dimension of cpu node, radix, scale, input memory address, output memory address.

For Tiny-YOLO-V3, nearest_upsample_cpu() function is implemented and registered. The example of how to register it is shown in 2.2.1.

```
/**
 * nearest_upsample_cpu() - execute nearest upsampling CPU operations
 *
 * @image_p: pointer to struct kdp_image with buffer of raw image
 *          and dimension, and data for pre/cpu/post processing.
 *
 * @cpu_op: the CPU operation type id this function was registered for
 *
 * This is a cpu function which uses Kneron NCPU to do nearest upsampling.
 *
* Return value:
* 0 : success
* <0 : error
```

```
 */
int nearest_upsample_cpu(int cpu_op, struct kdp_image_s *image_p) {
    // Get the input dimension
    int row_num_in = CPU_OP_NODE_INPUT_ROW(image_p);
    int col_num_in = CPU_OP_NODE_INPUT_COL(image_p);
    int chnl_num_in = CPU_OP_NODE_INPUT_CH(image_p);
    // Get the output dimension
    int row_num_out = CPU_OP_NODE_OUTPUT_ROW(image_p);
    int col_num_out = CPU_OP_NODE_OUTPUT_COL(image_p);
    // Calculate the input and output 16-bytes aligned column size
    int col_num_pad_in = pad_up_16(col_num_in);
    int col_num_pad_out = pad_up_16(col_num_out);

    int oi, oj, ii, ij, k;
    // Calculate the upsampling scales
    int h_scale = row_num_out / row_num_in;
    int w_scale = col_num_out / col_num_in;
    int out_index, in_index;
    // Get the input memory address and output memory address
    uint8_t *data_array = (uint8_t*)CPU_OP_NODE_INPUT_ADDR(image_p);
    uint8_t *dst_data_array = (uint8_t*)CPU_OP_NODE_OUTPUT_ADDR(image_p);
    // HCW for DRAM
    for (oj = 0; oj < row_num_out; oj++) {
        for (k = 0; k < chnl_num_in; ++k) {
            for (oi = 0; oi < col_num_out; oi++) {
                ij = oj / h_scale;
                ii = oi / w_scale;
                // Calculate the output index. As each row is 16-bytes aligned, col_num_pad_out is used
                // to calculate the output location
                out_index = oi + col_num_pad_out * (k + chnl_num_in * oj);

                // Calculate the input index. As each row is 16-bytes aligned, col_num_pad_in is used
                // to calculate the input location
                in_index = ii + col_num_pad_in * (k + chnl_num_in * ij);

                // Fill the input data to the output location
                dst_data_array[out_index] = data_array[in_index];
            }
        }
    }
    return 0;
}
```

2.2.5 Post-processing

Postprocess function is registered in scpu_comm_thread(). The implementation of this function follows the rule of 16-bytes alignment and HCW data layout.

For Tiny-YOLO-V3, post_yolo_v3() function is implemented and registered. The example of how to register it is shown in 2.2.1.

(1) Part 1 of post_yolo_v3

This part of code does preparation for the postprocessing logic: get the result memory address; calculate total class of this model.

```c
/**
 * post_yolo_v3() - post process function for yolo v3
 *
 * @model_id: the model id this function was registered for
 *
 * @image_p: pointer to struct kdp_image with buffer of raw image
 *           and dimension, and data for pre/cpu/post processing.
 *
 * This is a post process function which outputs the detection results.
 *
 * Return value:
 * 0 : success
 * <0 : error
 */
int post_yolo_v3(int model_id, struct kdp_image_s *image_p)
{
    struct yolo_v3_post_globals_s *gp = &u_globals.yolov3;
    int i, j, k, div, ch, row, col, max_score_class, good_box_count, class_good_box_count, good_result_count, len;
    float box_x, box_y, box_w, box_h, box_confidence, max_score;
    int8_t *src_p, *x_p, *y_p, *width_p, *height_p, *score_p, *class_p;
    struct bounding_box_s *bbox;
    struct yolo_result_s *result;
    struct bounding_box_s *result_box_p, *result_tmp_p, *r_tmp_p;

    int data_size, grid_w, grid_h, class_num, grid_w_bytes_aligned, w_bytes_to_skip;
    int incr_off;

    // Get the data size, the data size is 1 byte for KL520
    data_size = (POSTPROC_OUTPUT_FORMAT(image_p) & BIT(0)) + 1;     /* 1 or 2 in bytes */

    // Get the result memory address
    result = (struct yolo_result_s *)(POSTPROC_RESULT_MEM_ADDR(image_p));
    result_box_p = result->boxes;

    result_tmp_p = gp->result_tmp_s;

    // Calculate the total classes for this model
    class_num = POSTPROC_OUT_NODE_CH(image_p) / YOLO_V3_CELL_BOX_NUM - YOLO_BOX_FIX_CH;

    result->class_count = class_num;
    bbox = gp->bboxes_v3;
    good_box_count = 0;

    float anchers_v[3][2];
    int idx;
    int offset = sizeof(struct out_node_s);
    struct out_node_s *out_p;
```

(2) Part 2 of post_yolo_v3

This part of code calculates all valid detection boxes from the 2 output nodes of Tiny-YOLO-V3: calculate the aligned byte size of a row; get the radix and scale of output node; convert fixed point value back to float; calculate all valid detection boxes. Kdp_image_s struct provides all information needed for postprocessing: dimension of output node, radix, scale, output memory address, result memory address.

```
/* Second part of post_yolo_v3 */

    // Get the valid detections from the 2 output nodes
    for (idx = 0; idx < POSTPROC_OUTPUT_NUM(image_p); idx++) {
        // Get the pointer to output node
        out_p = (struct out_node_s *)((uint32_t)POSTPROC_OUT_NODE(image_p) + idx * offset);

        // Get the pointer to output memory address of output node
        src_p = (int8_t *)OUT_NODE_ADDR(out_p);

        // Get width and height of output node
        grid_w = OUT_NODE_COL(out_p);
        grid_h = OUT_NODE_ROW(out_p);
        // Calculate the byte size of a row
        len = grid_w * data_size;
        // Calculate the aligned byte size of a row: 16-byte alignment
        grid_w_bytes_aligned = round_up(len);
        // Calculate the skipped bytes for each row
        w_bytes_to_skip = grid_w_bytes_aligned - len;
        len = grid_w_bytes_aligned;

        // Get the start address of x, y, w, h, confidence_score, class probability
        x_p = src_p;
        y_p = x_p + len;
        width_p = y_p + len;
        height_p = width_p + len;
        score_p = height_p + len;
        class_p = score_p + len;

        // Get the radix of output node and calculate 2 raised to the power of radix
        div = 1 << OUT_NODE_RADIX(out_p);
        // Get the scale of output node
        float fScale = *(float *)&OUT_NODE_SCALE(out_p);

        // Set different anchors for output node
        if (0 == idx) {
            memcpy(anchers_v, anchers_v0, 6*sizeof(float));
        } else {
            memcpy(anchers_v, anchers_v1, 6*sizeof(float));
        }

        // The data layout is in HCW
        for (row = 0; row < grid_h; row++) {
            for (ch = 0; ch < YOLO_V3_CELL_BOX_NUM; ch++) {
                for (col = 0; col < grid_w; col++) {
                    if (data_size == 1) {
                        box_x = (float)*x_p;
                        box_y = (float)*y_p;
                        box_w = (float)*width_p;
                        box_h = (float)*height_p;
                        box_confidence = (float)*score_p;
                    } else {
                        box_x = (float)*(uint16_t *)x_p;
                        box_y = (float)*(uint16_t *)y_p;
                        box_w = (float)*(uint16_t *)width_p;
                        box_h = (float)*(uint16_t *)height_p;
                        box_confidence = (float)*(uint16_t *)score_p;
                    }

                    // Calculate the confidence score with sigmoid function
                    box_confidence = sigmoid(do_div_scale(box_confidence, div, fScale));

                    /* Get scores of all class */
                    for (i = 0; i < class_num; i++) {
                        if (data_size == 1)
                            gp->box_class_probs[i] = (float)*(class_p + i * len);
                        else
                            gp->box_class_probs[i] = (float)*(uint16_t *)(class_p + i * len );
                        // Calculate the class probability with sigmoid function
                        gp->box_class_probs[i] = sigmoid(do_div_scale(gp->box_class_probs[i], div, fScale));
                    }

                    //increase pointer to next position
                    x_p += data_size;
                    y_p += data_size;
                    width_p += data_size;
                    height_p += data_size;
                    score_p += data_size;
                    class_p += data_size;

                    /* Find highest score class */
                    max_score = 0;
                    max_score_class = -1;
                    for (i = 0; i < class_num; i++) {
                        gp->box_class_probs[i] = gp->box_class_probs[i] * box_confidence;
                        if (gp->box_class_probs[i] >= prob_thresh_yolov3) {
                            if (gp->box_class_probs[i] >= prob_thresh_yolov3 && gp->box_class_probs[i] > max_score) {
                                max_score = gp->box_class_probs[i];
                                max_score_class = i;
                            }
                        }
                    }
                    /* Calculate the valid detection box: top-left, right-bottom */
                    if (max_score_class != -1) {
                        box_x = do_div_scale(box_x, div, fScale);
                        box_y = do_div_scale(box_y, div, fScale);
                        box_w = do_div_scale(box_w, div, fScale);
                        box_h = do_div_scale(box_h, div, fScale);

                          box_x = (sigmoid(box_x) + col) / grid_w;
                          box_y = (sigmoid(box_y) + row) / grid_h;
                          box_w = expf(box_w) * anchers_v[ch][0] / DIM_INPUT_COL(image_p);
                          box_h = expf(box_h) * anchers_v[ch][1] / DIM_INPUT_ROW(image_p);

                          bbox->x1 = (box_x - (box_w / 2)) * DIM_INPUT_COL(image_p);
                          bbox->y1 = (box_y - (box_h / 2)) * DIM_INPUT_ROW(image_p);
                          bbox->x2 = (box_x + (box_w / 2)) * DIM_INPUT_COL(image_p);
                          bbox->y2 = (box_y + (box_h / 2)) * DIM_INPUT_ROW(image_p);
                          bbox->score = gp->box_class_probs[max_score_class];
                          bbox->class_num = max_score_class;

                          bbox++;
                          good_box_count++;
                    }
                }

                /* go to next grid_w*85 values for next anchors*/
                // The output dimension is grid_h * 255 * grid_w, and there are 3 anchors
                // The data sequence is:
                // 1st anchor: row of x, row of y, row of w, row of h, row of confidence score, row of class_prob1, ..., row of class_prob80;
                // 2nd anchor: row of x, row of y, row of w, row of h, row of score, row of class1, ..., row of class80;
                // 3rd anchor: row of x, row of y, row of w, row of h, row of score, row of class1, ..., row of class80;
                incr_off = w_bytes_to_skip + grid_w_bytes_aligned*84; //84 = 85 -1
                x_p += incr_off;
                y_p += incr_off;
                width_p += incr_off;
                height_p += incr_off;
                score_p += incr_off;
                class_p += incr_off;
            }
        }

    }

    good_result_count = 0;
```

(3) Part 3 of post_yolo_v3

This part of code does non max suppression (NMS) on all valid detection boxes. Then final detection boxes are generated.

```
/* Third part of post_yolo_v3 */

    // Do NMS for all valid detection boxes, and get the final good result
    for (i = 0; i < class_num; i++) {
        bbox = gp->bboxes_v3;
        class_good_box_count = 0;
        r_tmp_p = result_tmp_p;

        for (j = 0; j < good_box_count; j++) {
            if (bbox->class_num == i) {
                memcpy(r_tmp_p, bbox, sizeof(struct bounding_box_s));
                r_tmp_p++;
                class_good_box_count++;
            }
            bbox++;
        }

        if (class_good_box_count == 1) {
            memcpy(&result_box_p[good_result_count], &result_tmp_p[0], sizeof(struct bounding_box_s));
            good_result_count++;
        } else if (class_good_box_count >= 2) {
            qsort(result_tmp_p, class_good_box_count, sizeof(struct bounding_box_s), box_score_comparator);
            for (j = 0; j < class_good_box_count; j++) {
                if (result_tmp_p[j].score == 0)
                    continue;
                for (k = j + 1; k < class_good_box_count; k++) {
                    if (box_iou(&result_tmp_p[j], &result_tmp_p[k]) > nms_thresh_yolov3) {
                        result_tmp_p[k].score = 0;
                    }
                }
            }

            for (j = 0; j < class_good_box_count; j++) {
                if (result_tmp_p[j].score > 0) {
                    memcpy(&result_box_p[good_result_count], &result_tmp_p[j], sizeof(struct bounding_box_s));
                    good_result_count++;
                }
            }
        }
    }

    dbg_msg("good_result_count: %d\n", good_result_count);
    result->box_count = good_result_count;
    for (unsigned int i = 0; i < good_result_count; i++) {
        dbg_msg("post_yolo3 %f %f %f %f %f %d\n", result->boxes[i].x1, result->boxes[i].y1, result->boxes[i].x2, result->boxes[i].y2, result->boxes[i].score, result->boxes[i].class_num);
    }
    len = good_result_count * sizeof(struct bounding_box_s);

    return len;
}
```

2.3 dme_conf_generator.py

The python script is modified accordingly to the model input dimension by referring to 1.4. It generates different binary configuration file which is sent to KL520 through USB interface. And each binary file is consistent with one type of configuration.

```python
# scripts to generate configuration file in DME mode

# example for Tiny-YOLO-V3
pre_bypass       = 1
post_bypass      = 0

models_selection = 0x00000001 # Reserved. 0x00000001 if there is only one model
output_num       = 2
image_col        = 224
image_row        = 224
image_ch         = 4
image_format     = 0
parallel         = 0

if pre_bypass:
    image_format = image_format | 1 << 19 # 1 << 19 to bypass pre

if post_bypass:
    image_format = image_format | 1 << 28 # 1 << 28 to bypass post and output raw results

if parallel:
    image_format = image_format | 1 << 27 # 1 << 27 to do parallel processing

models_selection   = models_selection.to_bytes(4, byteorder='little')
output_num         = output_num.to_bytes(4, byteorder='little')
image_col          = image_col.to_bytes(4, byteorder='little')
image_row          = image_row.to_bytes(4, byteorder='little')
image_ch           = image_ch.to_bytes(4, byteorder='little')
image_format       = image_format.to_bytes(4, byteorder='little')

img_conf = models_selection + output_num + image_col + image_row + image_ch + image_format

with open("config.bin", 'wb') as binary_file:
    binary_file.write(img_conf)
```