A Tutorial on FlexCNN

Source Paper:

End-to-End Optimization of Deep Learning Applications

HLS codes

Top Kernel

→ These two pointers point to the same buffer
→ Since they want to be accessed in different modules
→ And we are using a dataflow architecture
→ We should define separate pointers

```
void top_kernel(
        bus_t0 *global_cin,
        bus_t0 *global_prev_cin,
        bus_t0 *global_cout,
        bus_t1 *global_weight,
        bus_t2 *global_bias,
        bus_t3 *layer_config
){   …
        for (layer = 1; layer <= LAYER_NUM; layer++)
                engine(global_cin, global_prev_cin, global_weight, global_bias, global_cout, config);
    …
}
```

The pointers to DRAM addresses for each buffer

The pointer to DRAM address where all the instructions are stored

Contains all the modules connected as a dataflow architecture

Engine

```
void engine(…){
#pragma HLS DATAFLOW
// ----------------------------------------
// Definitions of data fifos
// ----------------------------------------
hls::stream<CinLoadData0Type> fifo_cin_load_0;
#pragma HLS STREAM variable=fifo_cin_load_0 depth=128
…
// ----------------------------------------
// Definitions of config fifos
// ----------------------------------------
hls::stream<ConfigInst> config_prev_load;
#pragma HLS STREAM variable=config_prev_load depth=16
…
      cin_load(
            global_cin,
            config,
            fifo_cin_load_0,
            config_prev_load
      );
      …

}
```

- The modules are connected in dataflow format
- The modules are connected via two kinds of FIFOs
  - Data and Config/Instruction
- Each FIFO must be read/write in only one module
- For each new module:
  - Define the required FIFOs here
  - Add its function call here (the order matters)

Instructions

- Each layer has 5 192-bit instructions → 6 float numbers

The actual size of each dimension

Size of each dimension after padding due to the filter size and tiling is applied

Inst0: in_num_hw	out_num_hw	in_h_hw	in_w_hw	out_h_hw	out_w_hw
Inst1: in_num	out_num	in_h	in_w	out_h	out_w
Inst2: cin_offset	weight_offset	bias_offset	cout_offset	filter_s1, filter_s2	stride
Inst3: layer_en	prev_cin_offset	in_num_t, out_num_t	in_h_t	in_w_t	nxt_layer_batch
Inst4: task_num1	task_num2	local_accum_num	local_reg_num	row_il_factor	col_il_factor

Controlling signals for the systolic array

Enable signals for the modules, DRAM location of the input to the previous layer, and tiling sizes

conv_1st_en, depth_conv_en, conv_en, relu_en, relu6_en, pool_en, up_sample_en, bias_en, inter_load_en, inter_write_en, batch_norm_en_conv, load_prev_cin, batch_norm_en_depth

DRAM location for each of the inputs + filter/stride sizes

General Structure of the Modules – Setup

```
/*
 * Function name: sample_module
 * Function description: A dummy module to show the general structure of the modules
 */
void sample_module(
    hls::stream<CinLoadData0Type>  &fifo_cin,
    hls::stream<ConfigInst>        &fifo_config_in,
    hls::stream<CinLoadData0Type>  &fifo_cout,
    hls::stream<ConfigInst>        &fifo_config_out
){
#pragma HLS INLINE off
        ConfigInst inst0 = fifo_config_in.read();
        fifo_config_out.write(inst0);
        ConfigInst inst1 = fifo_config_in.read();
        fifo_config_out.write(inst1);
        …

        while (!done){
            // inst0
            ap_uint<32> LAYER_IN_NUM_HW     = inst0(32*0+31, 32*0);
            ap_uint<32> LAYER_OUT_NUM_HW = inst0(32*1+31, 32*1);
            …
            // set up some configuration signals
            uint FILTER_S = (DEPTH_CONV_EN == 1)? (uint)FILTER_S1: ((CONV_EN == 1)? (uint)FILTER_S2: 1);
            bool separable_conv = (DEPTH_CONV_EN == 1) && (CONV_EN == 1);
            …
```

They all have a short description

You should first read the instructions and fill the corresponding output FIFO

Extract the info from inst. according to inst. description

Set up the required signals

General Structure of the Modules - Bypass

```
switch(SAMPLE_EN){
        // bypass this module
        case 0:
          if (out_num_iter == 0) {
                int o = 0, h = 0, w = 0;
                bool done1 = 0;
                while(!done1){
#pragma HLS PIPELINE II=1
                        CinLoadData0Type tmp = fifo_cin.read();
                        fifo_cout.write(tmp);

                        // Repeat until the whole tile is read
                        w++;
                        if (w == w_bound){
                                w = 0;
                                h++;
                                if (h == h_bound){
                                        h = 0;
                                        o++;
                                        if (o == LAYER_IN_NUM_T / SIMD_LANE){
                                                o = 0;
                                                done1 = 1;
                }}}}}
        break;
```

- Adjust the frequency of running this section
- Generally, each tile is processed LAYER_OUT_NUM times

Bypass the module by simply passing the data

- Determine the tile bound
  - Usually, it should be Tw/S + F – 1
    - (or Th/S + F -1)
    - S: stride
    - F: filter size

- Each entry in FIFO is SIMD_LANE data elements
  - The data are packed along the IN_NUM dimension to decrease the comm. time

General Structure of the Modules - Compute

```
case 1:
   // Implement your compute engine
break;

// Repeat until all the tiles are read
// Must repeat the computation until LAYER_OUT_NUM output feature maps are generated
in_num_iter += LAYER_IN_NUM_T;
if (in_num_iter >= LAYER_IN_NUM){
                in_num_iter = 0;
                in_h_iter += LAYER_IN_H_T;
                if (in_h_iter >= LAYER_IN_H){
                                in_h_iter = 0;
                                in_w_iter += LAYER_IN_W_T;
                                if (in_w_iter >= LAYER_IN_W){
                                                in_w_iter = 0;
                                                out_num_iter += LAYER_OUT_NUM_T;
                                                if (out_num_iter >= LAYER_OUT_NUM){
                                                                out_num_iter = 0;
                                                                layer_iter += 1;
                                                                layer_start = 1;
                                                                if (layer_iter == LAYER_BATCH){
                                                                                layer_iter = 0;
                                                                                done = 1;
}}}}}
```

Packing and Unpacking

- You may need to pack/unpack the data
  - Remember that each entry in FIFO contains SIMD_LANE elements
- Suggested coding style:

```
// Read data
CinLoadData0Type cin_tmp = fifo_cin.read();
// Unpack the data based on the SIMD_LANE
for (int lane = 0; lane < SIMD_LANE; lane++){
#pragma HLS UNROLL
                ap_uint<DATA_W0> u32_tmp = cin_tmp(DATA_W0 - 1, 0);
                cin_buf[lane] = Reinterpret<data_t0>(u32_tmp);
                cin_tmp = cin_tmp >> DATA_W0;
 }
```

```
// Pack the data according to SIMD_LANE
CoutLoadData0Type wide_tmp = (
#if SIMD_LANE == 16
                cout_buf[15], cout_buf[14], cout_buf[13], cout_buf[12],
                cout_buf[11], cout_buf[10], cout_buf[9], cout_buf[8],
                cout_buf[7], cout_buf[6], cout_buf[5], cout_buf[4],
                cout_buf[3], cout_buf[2], cout_buf[1], cout_buf[0]
#elif SIMD_LANE == 8
                cout_buf[7], cout_buf[6], cout_buf[5], cout_buf[4],
                cout_buf[3], cout_buf[2], cout_buf[1], cout_buf[0]
#elif SIMD_LANE == 4
                cout_buf[3], cout_buf[2], cout_buf[1], cout_buf[0]
#elif SIMD_LANE == 2
                cout_buf[1], cout_buf[0]
#elif SIMD_LANE == 1
                cout_buf[0]
#endif
);
// Write data
fifo_cout.write(wide_tmp);
```

- The rest of the details for each module is commented through the code

Auto Compile

Preprocessing Steps

- To setup the HW config you have to follow 3 steps:
  - 1. Protobuf Translation
    - Input: Protobuf file from TensorFlow containing the graph description
    - Output: A file containing the network architecture needed by the hardware
      - If an operation is not supported by the HW, you will get a warning here
  - 2. DSE
    - Input: The generated file of the previous step
    - Output: A file containing the network architecture with the best HW config for each layer
  - 3. Instruction Generation
    - Input: The generated file of the previous step
    - Output: A file containing the instructions to run each of the layers

Protobuf Translation

```
"""
   Pass the following command line arguments or change the default value

    -p : The location where the protobuf text file is stored (do not pass the binary format)
    -m : The name of the output file. This file will have the information needed by the hardware. You should pass the
file to DSE in the next step.
    -i : The name of the json file containing format of the image
    -g : Only specify it if you have used a name for your graph in the protobuf text file. Otherwise leave it blank.
    -n : The name of the first input tensor of your graph
    -o : The name of the last tensor in your graph
 """
```

- It first sorts the graph in topological order and then process the operations one by one
  - The operations between two consecutive convolutional layers will be fused as one layer
    - Except for pooling and upsampling that is counted as a new layer
- Contains a lot of helper functions that you can use to extract the info of the new layers
  - Such as: get_tensor_dest_ops, _ensure_op, get_conv_layer_info
  - Be sure to check the available functions first if you wanted to add support for a new operation

DSE

```
"""
   Pass the following command line arguments or change the default value

    -m          : The generated file from protobuf_translation
    -i          : The name of the json file containing format of the image
    -b          : The name of the json file containing the number of resources of the target FPGA board
    --parallel : (True/False) Specify if you want to run the multi-threaded version of this code or not
    --systolic : (True/False) Specify whether you want to search for the shape of systolic array or not
    -dt         : The dynamic tiling level you want to have (0: Disabled
                                                             1: Only number of channels will be dynamic
                                                             2: All the dimensions will be dynamic)
 """
```

- Can control the dynamic tiling level and whether it should be run as a multi-threaded version or not
  - By default, it will search for the best size of the systolic array for all cases
- It will also output the best size for systolic array, and the maximum tiling factor used
  - In “opt_params.json”
  - Follow the instructions in “Build the HLS kernel” section of README to build the kernel using these values

DSE – New Module

- When defining a new module
  - Count the number of cycles for the computation of one tile
  - If it is more than the following, it should be added to the DSE code

$$\frac{Tn * (Th + Fh - 1) * (Tw + Fw - 1)}{SIMD\_LANE}$$

  - To add the latency:
    - 1. Define a function like the following computing the latency
    - 2. Add the function instance to “layer_latency_est” function

```
def relu_est(in_num, in_num_t, out_num_t, out_h_t, out_w_t, lane):
  return out_num_t * out_h_t * out_w_t / lane / np.ceil(in_num / in_num_t)
```

Instruction Generator

```
"""
   Pass the following command line arguments or change the default value

    -t : The name of the json file containing the maximum tiling factors and the systolic array size
    -m : The generated file from DSE
    -i : The name of the json file containing format of the image
    -o : The name of the output tensors
 """
```

- Will produce two files
  - "params.h"
    - The HW config for building the hardware and the parameters needed to test the result
    - Copy it to the HLS_Codes folder and SDx_project/src
  - "network.insts"
    - Contains a set of instructions for each of the layers
- If you have defined a new module, update the instruction generator

TensorFlow Integration

Integration

- After you have followed the instructions in "Build the SDx Project" to create the bitstream, you can start doing the integration
  - Copy the bitstream (.xclbin file), host executable (.exe file) and instructions (.insts file) to libsacc/config
  - Follow the instructions in README of libsacc directory to install the library
    - This library can get as many as number of images in a batch
    - You just need to make your TensorFlow code recognize this library
  - Replace your TensorFlow code with tf_DSA directory

Rewriting TensorFlow Code for Integration

- 1. Copy sacc_utils.py to your project
  - Modify the path to the libsacc library
- 2. Import libsacc library in your code
  - import sacc_utils
- 3. Add the following to offload the CNN computation to FPGA
  - self.constants = sacc_utils.Constants()
  - self.sacc_module = tf.load_op_library(self.constants.custom_lib_path)
  - result = self.sacc_module.sacc([self.tensor_image])

Takes a list of N images as input,
sends them to FPGA,
and returns a list of size N as the output

Check out this file
as an example

Pipelining at the TensorFlow Level

- To enable pipelining at the TensoFlow level
  - You can divide your tasks into several functions
    - Use Python's Process to mimic the stages of the pipeline
    - Use Python's Queue to connect the stages to each other
    - Example:

```
input_q = Queue()
output_q = Queue()
t = Process(target=task1, args=(input_q, output_q))
t.start()
```

Check out this file as an example