Convolutional Neural Network

1. Introduction

Convolutional Neural Network (CNN) is widely used in the machine learning task in the computer vision and neural language processing area. In this project, we implement the convolutional neural network algorithm on the DE-1 SOC FPGA + HPS to run a pre-trained CNN-based network: VGG-11.

2. Data Flow

The figure below shows the structure of the VGG11, which contains these types of operators: conv2d, Relu, max pooling 2d, adaptive average pooling, linear(fully connection).

To limit the volume of data transmitted between the HPS and the FPGA, we did the following modification compared with the original VGG:

1. Instead of the famous VGG16, we apply VGG11, which is lighter with less parameters compared with VGG16.

2. The input image shape is set to 32 by 32 by 3 rather than 224 by 224 by 3, and the output class number is changed from 1000 to 10. We use an open source 32 by 32 RGB dataset called CIFAR-10 to train the VGG11 network.

3. Float point 16 is used instead of float point 32. We use fp16 to make the data width shorter. The fp16 follows the IEEE 754 with one bit sign, five bit exponent and ten bit mantissa.
3. Hardware System Architecture

The system is built with two main components, HPS and FPGA that work with each other. From high level, the HPS is in charge of interfacing with the user and obtains the initial data, whereas the FPGA is in charge of low level computations.

The system is first configured and generated using qsys (platform designer). The qsys configuration is as follows:
The detailed configuration of each block is as follows:
The HPS system is configured with two DDR3 SDRAM interfaces, connected through the Avalon Memory Mapped interface as shown below. One is configured to read-only and the other as write-only.

A list of PIO IP cores are added for HPS to send or receive control signals from / to FPGA. As seen below, the h2f_finish signal has one bit and is configured as an input, with conduit “h2f_finish” and base address 0x00000080.
This signal can be used in the top level module as follows, as part of the generated qsyr module.

```
// PIO to start the image reading from OCM
.image_sent_ocm_export = (start_ocm_wire),
.fpga_stat_export = (fpga_stat_wire),
h2f_start_export = (h2f_start),
h2f_read_length_export = (h2f_read_length),
feat_map_dim_export = (feat_map_dim),
h2f_buf_offset_export = (h2f_buf_offset),
.f2h_start_export = (f2h_start),
f2h_write_length_export = (f2h_write_length),
f2h_buf_offset_export = (f2h_buf_offset),
h2f_finish_export = (h2f_finish),
f2h_finish_export = (f2h_finish),
```

In the HPS code, we first take advantage of the "sopc-create-header-files" tool from quartus to generate a header file that has the configured base addresses defined using the following command:

```
sopc-create-header-files "path/to/qsys.sopcinfo" --single hps_0.h --module hps_0
```
The generated header file includes the defines above, and we can use the PIO in the HPS C code as follows:

```c
// lightweight HPS-to-FPGA bridge
void *virtual_base;
virtual_base = mmap(NULL, HW_REGS_SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, HW_REGS_BASE);

uint32_t *h2f_finish = virtual_base + ((unsigned long)(ALT_LWFPGASLVS_OFST + H2F_FINISH_BASE) & (unsigned long)(HW_REGS_MASK));

*h2f_finish = 0;
```

Similarly, as the DMA controller is also on the lightweight HPS-to-FPGA bridge, we are able to command the DMA controller similarly:

```c
// ========= HPS ----DMA-----> FPGA ==========
void *h2p_lw_dma_addr0 = NULL;
h2p_lw_dma_addr0 = virtual_base + ((unsigned long)(ALT_LWFPGASLVS_OFST + DMA_0_BASE) & (unsigned long)(HW_REGS_MASK));

#define _DMA_REG_STATUS(BASE_ADDR) *((uint32_t *)BASE_ADDR+0)
#define _DMA_REG_READ_ADDR(BASE_ADDR) *((uint32_t *)BASE_ADDR+1)
#define _DMA_REG_WRITE_ADDR(BASE_ADDR) *((uint32_t *)BASE_ADDR+2)
#define _DMA_REG_LENGTH(BASE_ADDR) *((uint32_t *)BASE_ADDR+3)
#define _DMA_REG_CONTROL(BASE_ADDR) *((uint32_t *)BASE_ADDR+6)

_DMA_REG_STATUS(h2p_lw_dma_addr0) = 0;
```
4. Main Modules

4.1 readOCM.sv

This module handles reading from the OCM, including both reading weight and bias, and loading the feature map.

After HPS finishes asking the DMA controller to copy data into OCM, it sends a signal to the “start” wire below, after which the readOCM module will load the weight and bias from OCM into “weight_bias” and raise “in_data_ready” when finished.
The module also reads feature maps from OCM at index “conv_idx”, after receiving a rising edge on “start_fm”. It will read the feature map into “feat_map_in” and raise “finish_fm” when done.

The detailed interface is as follows.

```verilog
module readOCM(
    input logic clk, reset,
    // ------------ Read weight and bias ------------
    // Input from HPS
    input logic start, 
    input logic [15: 0] read_length, 
    input logic [5: 0] read_data_dim,
    // Output to pipeline
    output logic in_data_ready,
    output logic [(3*3+1)*16-1: 0] weight_bias,
    // ----------------------------------------------
    // ------------ Read feature map 3x3 ------------
    // Input from pipeline
    input logic start_fm, 
    input logic [15: 0] conv_idx,
    // Output to pipeline
    output logic finish_fm,
    output logic [3*3*16-1: 0] feat_map_in,
    // ----------------------------------------------
    // On-Chip RAM 0 s2 (read)
    input logic [ 7: 0] ocm0_readdata, 
    output logic [16: 0] ocm0_addr, 
    output logic ocm0_chip, 
    output logic ocm0_clk_enab,
    // Debug
    output logic [ 2: 0] debug_state
);
```

The module is implemented using a finite state machine structure with the following states. Please refer to the code for detailed implementation.

S0: Reset
S1: Prepare to read weight bias
S2: Read weight bias
S3: Finished reading weight bias. Wait for feature map request
S4: Received feature map request, prepare to read
S5: Read feature map 8 LSB
S6: Read feature map 8 MSB
S7: Finished feature map reading

4.2 convOpt.sv

This module implements convolution. Input “in_data_ready” signals that weight and bias is ready, and “in_data_dim” is the dimension of the data. "Weight_bias" is the weight and bias read by readOCM module. The module uses a finite state machine to manage states. Please see the source code for detailed implementation.

module convOpt(
    input logic clk, reset,

    // Input from pipeline
    input logic in_data_ready,
    input logic [ 5: 0] in_data_dim,
    input logic [(3*3+1)*16-1: 0] weight_bias,

    // Request FM from pipeline
    output logic start_fm,
    output logic [15: 0] conv_idx,
    input logic finish_fm,
    input logic [3*3*16-1: 0] feat_map_in,

    // Output to pipeline
    output logic start_out,
    output logic [16: 0] out_idx,
    output logic [3*3*16-1: 0] feat_map_out,

    // Output to HPS
    output logic finish,

    // Debug
    output logic [ 2: 0] debug_state
);

4.3 writeOCM.sv

This module handles writing data to OCM that will be DMA’d back to HPS. The module uses a finite state machine to manage states. Please see the source code for detailed implementation.

module writeOCM(
    input logic clk, reset,
    input logic start_out,
    input logic [16: 0] out_idx,
    input logic [3*3*16-1: 0] feat_map_out,
    output logic finish_out,
    // On-Chip RAM 1 s1 (write)
    output logic [ 7: 0] ocm1_writedata,
    output logic [16: 0] ocm1_addr,
    output logic ocm1_chip,
    output logic ocm1_clk_enab,
    output logic ocm1_write,
    output logic [15: 0] count,
    // Debug
    output logic [ 2: 0] debug_state
);

5. Testbench

Version 1:
For one 32x32 feature map, total time is 472us
Loading 32*32: 40us
Computation: 390us
Sending 32*32: 40us

Version 2:
For one 32x32 feature map, total time is 840us
6. Synthesis result

<table>
<thead>
<tr>
<th>Resource</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Estimate of Logic utilization (ALMs needed)</td>
<td>5455</td>
</tr>
<tr>
<td>3. Combinational ALUT usage for logic</td>
<td></td>
</tr>
<tr>
<td>1. -- 7 input functions</td>
<td>156</td>
</tr>
<tr>
<td>2. -- 6 input functions</td>
<td>1540</td>
</tr>
<tr>
<td>3. -- 5 input functions</td>
<td>1262</td>
</tr>
<tr>
<td>4. -- 4 input functions</td>
<td>2119</td>
</tr>
<tr>
<td>5. -- &lt;=3 input functions</td>
<td>3861</td>
</tr>
<tr>
<td>5. Dedicated logic registers</td>
<td>3688</td>
</tr>
<tr>
<td>7. I/O pins</td>
<td>97</td>
</tr>
<tr>
<td>8. I/O registers</td>
<td>80</td>
</tr>
<tr>
<td>9. Total MLAB memory bits</td>
<td>0</td>
</tr>
<tr>
<td>10. Total block memory bits</td>
<td>73728</td>
</tr>
<tr>
<td>12. Total DSP Blocks</td>
<td>0</td>
</tr>
<tr>
<td>14. Total DLLs</td>
<td>1</td>
</tr>
<tr>
<td>15. Maximum fan-out node</td>
<td>CLOCK_50~Input</td>
</tr>
<tr>
<td>16. Maximum fan-out</td>
<td>3763</td>
</tr>
<tr>
<td>17. Total fan-out</td>
<td>52077</td>
</tr>
<tr>
<td>18. Average fan-out</td>
<td>3.95</td>
</tr>
<tr>
<td>Compilation Hierarchy Node</td>
<td>Combinational ALUTs</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1</td>
<td><code>main</code></td>
</tr>
<tr>
<td>2</td>
<td><code>conv_hps_system/The System</code></td>
</tr>
<tr>
<td>1</td>
<td><code>convOptolv</code></td>
</tr>
<tr>
<td>1</td>
<td><code>float_adder_fp_add</code></td>
</tr>
<tr>
<td>1</td>
<td><code>float_multigen_f_kernel[0]_fp_mult</code></td>
</tr>
<tr>
<td>1</td>
<td><code>float_multigen_f_kernel[1]_fp_mult</code></td>
</tr>
<tr>
<td>1</td>
<td><code>float_multigen_f_kernel[2]_fp_mult</code></td>
</tr>
<tr>
<td>1</td>
<td><code>float_multigen_f_kernel[3]_fp_mult</code></td>
</tr>
<tr>
<td>1</td>
<td><code>float_multigen_f_kernel[4]_fp_mult</code></td>
</tr>
<tr>
<td>1</td>
<td><code>float_multigen_f_kernel[5]_fp_mult</code></td>
</tr>
<tr>
<td>1</td>
<td><code>float_multigen_f_kernel[6]_fp_mult</code></td>
</tr>
<tr>
<td>1</td>
<td><code>float_multigen_f_kernel[7]_fp_mult</code></td>
</tr>
<tr>
<td>1</td>
<td><code>float_multigen_f_kernel[8]_fp_mult</code></td>
</tr>
<tr>
<td>1</td>
<td><code>pm_mult_Mult10</code></td>
</tr>
<tr>
<td>1</td>
<td><code>pm_mult_Mult1</code></td>
</tr>
<tr>
<td>1</td>
<td><code>pm_mult_Mult1[0]</code></td>
</tr>
<tr>
<td>1</td>
<td><code>pm_mult_Mult1[1]</code></td>
</tr>
<tr>
<td>1</td>
<td><code>pm_mult_Mult1[2]</code></td>
</tr>
<tr>
<td>1</td>
<td><code>writeOCM_wr</code></td>
</tr>
</tbody>
</table>
6. Other Modules & Test Files Developed

Main.sv: Top module for synthesis

readFMPipeline.sv: Version 1’s read from OCM

readImg.sv: Test module

tbEchoWrite.sv: Test bench top module for writing to OCM and echo back

tbFPU.sv: Test bench top module for the float16 module used

tbOpt.sv: Test bench top module for version 2

tbPipeline.sv: Test bench top module for version 1

tbRWFeatureMap.sv: Test bench top module for reading and writing feature map

testEcho.sv: Test bench top module for echoing data from input OCM to output OCM

writeFeatMap.sv: Write feature map to output OCM initial test

writeFMPipeline.sv: Write feature map to output OCM version 1

writeOCM.sv: Write feature map to output OCM version 2

writeOCM8.sv: Initial test to write to OCM 8-bit data