FPGA-based Convolutional Neural Network Accelerator

Andrew Pan
EE 538
University of Washington
Seattle, USA
andrep24@uw.edu

***Abstract*—This project explores two of the main concepts covered in the paper *Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks* [1] - loop unrolling and memory access optimization. The accelerator is described entirely using SystemVerilog HDL. A python testing framework was developed.to verify the accuracy of the convolution operation. The testing framework uses SciPy and NumPy to compute the convolution and then format the data in a way that could be read by a SystemVerilog testbench.**

I. Introduction

The primary motivation for this project is to understand the parallelism techniques utilized in the convolutional neural network (CNN) accelerator [1]. The two main techniques used in the accelerator are loop unrolling and memory access optimization. Loop unrolling takes advantage of computing operations that have independent data relationships. For memory access optimization, a ”ping-pong” buffer method is used. In this architecture, there are multiple buffer channels so that the memory loading and computation can be time-multiplexed.

To implement these techniques, SystemVerilog was chosen because of its common usage with FPGAs. The goal of the module is to perform a two-dimensional convolution on an arbitrarily sized input, output, and kernel. Furthermore, the module supports an arbitrarily sized tile, which is the dimension which loop unrolling has been performed.

II. Implementation - Modules

A. input_loop

This is the module that contains the arithmetic logic. It performs a multiplication between the given input_fm and weights and sums them all together to create a partial sum for one element in output_fm. Each module performs T_n of these calculations in parallel, where T_n is the input feature map tile size.

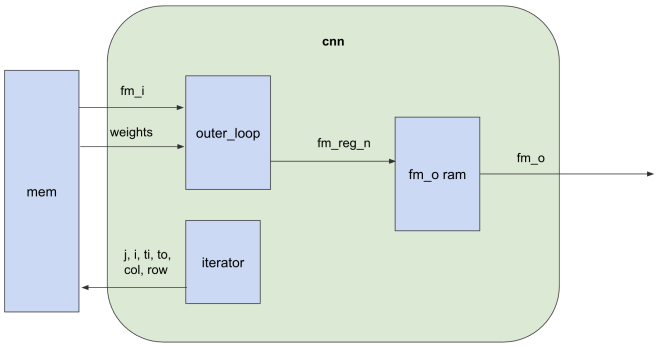

Fig. 1. CNN module datapath

B. output_loop

This module is the entirety of the computation engine. It instantiates T_m number of input_loop instances, where T_m is the output feature map tile size. By combining output_loop and input_loop, the computation is unrolled along the input and output feature map dimensions.

C. cnn_counter

This is a simple synchronous parameterized counter that is used to track the index of different arrays. The parameter max_p indicates the maximum count of the counter, while stride_p indicates how much the counter increases by when the en_i signal is asserted.

D. iterator

The iterator module manages all of the different iterators for the cnn module. It creates six instances of cnn_counter to control six iterator signals: j, i, ti, to, col, and row.

E. cnn

For complete datapath flow, see "Fig. **??**". Cnn is the top level module for this design. It takes a feature map and weights as inputs and outputs the computed output feature map. It uses a valid/ready handshake to initiate computation. The control logic is primarily controlled by the iterator module in addition to the FSM described in Fig X.

The datapath consists of an instance of output_loop which computes the value to be written to the output feature map fm_o. A synchronous RAM is used to store the values computed by the computation engine. Every cycle the machine is in the eBUSY state, the values computed will be accumulated in the fm_o RAM. The address is controlled by the value of the iterator signals from the iterator module.

III. Testing Framework

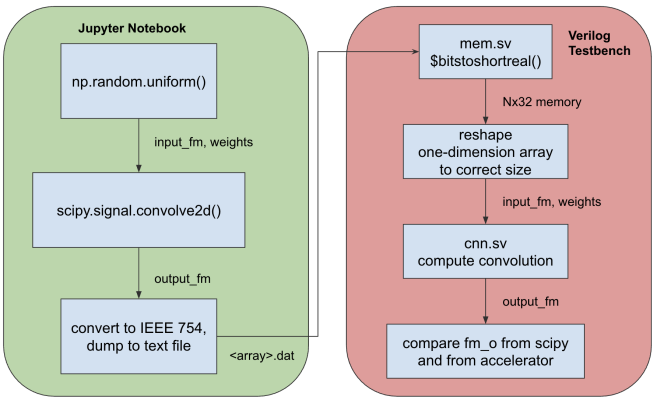

Fig. 2. Verification Flow

A. Jupyter Notebook

The goal of the Python Jupyter notebook is to provide a source of truth to compare against the output of the accelerator. Similar to the accelerator, the notebook accepts parameters to modify the size of the input_fm, weights, and output_fm matrices. NumPy is used to generate random values to populate the input_fm and weights matrices. The SciPy convolve2d() function is used to perform the convolution. The edges are padded with zeros and the "mode='same'" option is used.

After the convolution is calculated, the notebook writes the values of input_fm, weights, and output_fm to external files. The values are then converted to IEEE754 fp32 binary format. Next, they are written one per line so that they can be read by the SystemVerilog testbench.

B. SystemVerilog testbench

First, the output from the Jupyter notebook is read using the built-in $readmemb() function. The values are stored in a Nx32 memory, where N is the total number of values stored within a given array. After the values are read into the memory, the original arrays are reconstructed using a series of for loops. The binary values are converted to the shortreal type using $bitstoshortreal().

IV. Results

The CNN accelerator produces functionally correct results for any values for the given parameters: row size (R), column size (C), input map dimensions (N), and output dimension size (M). These results were verified through the testing framework described in Section III. The performance was difficult to analyze, because the design cannot currently be mapped to an FPGA.

V. Future Work

A. Parameterized kernel size

Currently, the module is only functionally correct for kernel sizes of K=1. This is because there needs to be additional logic to compute the values at the edges of the convolution. I believe this can be done in the iterator module by using some control logic to prevent the iterator from overflowing the bounds.

B. Memory access optimization

I would like to add a module to implement the "ping-pong" buffers. I have drafted a schematic for the module, but it has not yet been implemented.

C. Hardened floating point adders and multipliers

The design is currently using the shortreal data type, which follows the IEEE 754 fp32 standard. However, this data type is typically not synthesizable on FPGAs. I think this can be solved by treating the data as 32 bit numbers and then utilizing hardened fp32 IP cores.

D. Performance comparison with CPU implementation

Testing the CNN accelerator compared with a CPU implementation would be valuable to find out the speedup that is achieved by using an accelerator. The total compute time for given inputs can be calculated by multiplying the clock period by the number of clock cycles.

REFERENCES

[1] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," FPGA'15, February 22–24, 2015, Monterey, California, USA.