Deep Neural Networks on FPGA

Deep Neural Networks on FPGA

This project builds a deep neural network accelerator for an embedded Nios II system. It also explores interfacing with an off-chip SDRAM and using a PLL (phase lock loop) to generate clocks with specific properties. The RTL designs written are simple and straightforward, but the challenge comes from the system-on-chip components considering we built an entire system with an embedded softcore CPU. Additionally, this project explores Intel's Avalon interconnect protocol that the system follows.

Deep Neural Network Program Background

The type of neural network used is a multi-layer perceptron (MLP) to classify the MNIST hand-written digit dataset. The MLP will take a 28 x 28 -pixel greyscale image as an input and determine which digit (0 to 9) the image corresponds to.

An MLP consists of several linear layers that first multiply the previous layer's outputs by a weight matrix and add a constant "bias" value to each output, and then apply a non-linera activation function to obtain the current layer's outputs (called activations). Our MLP will have a 784-pixel input, two 1000-neuron hidden layers, and a 10-neuron output layer; the output neuron with the highest value will tell us which digit the network thinks it sees. For the activation function, we will use the rectified linear unit (ReLU) which maps all negative numbers to 0 and all positive numbers to themselves.

Each layer computes the following: a' = ReLU(W*a + b), where W is the weight matrix, a is the vector of the prior layer's activations, b is the bias vector, and a' is the current layer's activation vector.

These networks are pre trained and we have pre-formatted images. The following links shows data and the scripts if you are curious:

Data: https://github.com/ryaanluke/DNN-Hardware-Accelerator/tree/main/data

Python training script: https://github.com/ryaanluke/DNN-Hardware-Accelerator/tree/main/scripts

Q16.16 fixed point

This project uses signed Q16 fixed-point numbers to represent all biases, activations, weights, etc. The format works as follows:

- Complete numbers take 32 bits
- Bit 31 represents the sign
- Bits 30 to 16 represent the integral part
- And bits 15 to 0 are the fractional part

Conversion from Q16.16 to an integer is done by rounding:

Real number *n*	Rounded to
0 <= *n* <= 0.5	0
0.5 <= *n* <= 1	1
-0.5 <= *n* <= 0	0
-1 <= *n* <= -0.5	-1

Intel's Platform Designer Tool

To develop the Nios 11-based SoC system, the following tutorials will explain:

- How to build a Nios 11-based SoC system
- How to build components that hang off the Avalon on-chip interonnect
- How to run program on the Nios II core when you have a physical FPGA

Note: uses Intel FPGA Monitor Program

Links:

Introduction to the Platform Designer Tool:
ftp://ftp.intel.com/Pub/fpgaup/pub/Intel_Material/17.1/Tutorials/Introduction_to_the_Qsys_Tool.pdf

Making Platform Designer Components:
ftp://ftp.intel.com/Pub/fpgaup/pub/Intel_Material/17.1/Tutorials/making_qsys_components.pdf

Intel FPGA Monitor Program Tutorial for Nios II:
ftp://ftp.intel.com/Pub/fpgaup/pub/Intel_Material/17.1/Tutorials/Intel_FPGA_Monitor_Program_NiosII.pdf

Avalon Interface Specifications:
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

Introduction to the Intel Nios II Soft Processor:
ftp://ftp.intel.com/Pub/fpgaup/pub/Intel_Material/17.1/Tutorials/Nios2_introduction.pdf

Nios II Processor Reference Guide:

https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf

Nios II Software Developer Handbook:

https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2sw_nii5v2gen2.pdf

Memory Map

The following memory map shows the components and their memory locations.

Component	*Base*	*End*
Nios II debug mem slave	0x00000800	0x00000fff
JTAG UART	0x00001000	0x00001007
PIO (7-segment display)	0x00001010	0x0000101f
Word-copy accelerator	0x00001040	0x0000107f
DNN accelerator	0x00001080	0x000010bf
Instruction memory	0x00008000	0x0000ffff
SDRAM controller	0x08000000	0x0bffffff

PLL and SDRAM controller

Using Intel Platform Designer, create a Nios II system including a JTAG module as well as 32KB on-chip memory; both should be connected to the instruction and data masters. Reset and interrupt vectors of the on-chip memory should be default offsets at 0x0 and 0x20.

Phase-locked loop (PLL) IP has following specification:

- 50MHz reference clock
- One output clock, outclk0, a 50MHz clock with a phase shift of 0ps used to drive most of the circuit
- Another output clock, outclk1, a 50 MHz clock with a phase shift of -3000ps used for SDRAM chip
- Locked output

SDRAM controller has following specifications:

- Data width of 16 bits
- Row width of 13 bits
- Column width of 10 bits
- Refresh time of 7.8125µs
- t_rp time of 15ns
- t_rcd time of 15ns
- t_ac time of 5.4ns
- base address of 0x08000000

Simulating and debugging Nios II system

The toplevel module ***task2.sv*** instantiates the Nios II system and connects it to the SDRAM and 7-segment display. The provided ***run_nn.c*** file is also used to test the system. Using Intel's FPGA monitor, you can load ***nn.bin*** at address 0x0a000000 and one of the test images at address 0x0a800000. Finally, you can run the file to evaluate the neural network on the input image by running the program.

run_nn.c and ***task2.sv***: https://github.com/ryaanluke/DNN-Hardware-Accelerator/tree/main/task2

Data files: https://github.com/ryaanluke/DNN-Hardware-Accelerator/tree/main/data

To simulate the Nios II system, you need to add files generated from Platform Designer for the Nios system (nios_system/simulation/submodules/*.v, nios_system/simulation/submodules/*.sv, nios_system/simulation/nios_system.v into the Modelsim projects. You also need to copy nios_system/simulation/submodules/*.hex into the same simulation directory. You also need to use the following libraries: altera_mf_ver, cyclone_ver, altera_ver, and altera_lnsim_ver.

For our SDRAM controller, you need to simulate the DRAM module in the testbench and connect it to the device under test. Platform Designer provides a generic SDRAM simulation model. In simulation, the DRAM will be initialized from the file "altera_sdram_partner_module.dat" in the simulation directory. It's just a $readmem readable file that can be overwritten on our own.

A Memory Copy Accelerator

This IP component copies a range of words from one address in SDRAM to another. It has an Avalon memory-mapped slave interface (accept commands from the CPU) and an Avalon memory-mapped master interface (to interact with the SDRAM). Such capabilities are often called DMA copy, the point being to do the memory transfer without involving the CPU.

To start the copying process, the processor will first set up the transfer by writing the byte address of the destination range to word offset 1 in the accelerator's address range, the source byte address to word offset 2, and the number of 32-bit words to copy to word offset 3. Next, the CPU will write any value to word offset 0 to start the copy process. Finally, the CPU will read offset 0 to make sure the copy process has finished.

The CPU ensures the copy process has finished when it reads offset 0 in the modules address range. This means the memory copy accelerator module bust stall until the copy is complete.

Word	*Meaning*
0	When written, starts accelerator; may also be read
1	Destination byte address
2	Source byte address
3	Number of 32-bit words to copy

wordcopy.sv has these following parameters:

Name	*Type*	*Semantics*
clk	Input	clock
Rst_n	Input	Reset
[3:0] slave_address	Input	Address from CPU master
Slave_read	Input	Enabled if CPU master wants to read
Slave_write	Input	Enabled if CPU master wants to write
[31:0] Slave_writedata	Input	Input from CPU master what we want to write
Master_waitrequest	Input	From SDRAM_Controller to see if its busy
[31:0] master_readdata	Input	Input from SDRAM slave once we're done reading
Master_readdatavalid	Input	Input from SDRAM slave if read_Data is valid
Slave_waitrequest	Output	Assert if busy waiting for some internal process to be done
[31:0] slave_readdata	Output	Output to CPU master what we just read
[31:0] master_address	Output	Output to SDRAM salve the address we want to access
Master_Read	Output	Output if we want to read to the SDRAM slave
Master_write	Output	Output if want to write to the SDRAM slave

[31:0] master_writedata	Output	Output to SDRAM slave of what we want to write

The module has these following design ideas:

Block	*Functionality*	*Details*
Address Counter/Assign Block	Triggered at the event of an asynchronous reset or positive edge of the clock.	Controls the assignment of source and destination address, as well as incrementing addresses we are accessing.
Word Counter Block	Triggered at the event of an asynchronous reset or positive edge of the clock.	Controls the number of words being copied and sent.
Word assignment block	Triggered at the event of an asynchronous reset or positive edge of the clock.	Controls the word data received from slave.
State Machine Flow D-FlipFlop	D-flipflop controlling the flow of the state machine upon the trigger of asynchronous reset or positive edge of the clock.	Upon every clock cycle, the current state is moved to the next state. Upon an asynchronous reset, the current state is reset.
Next State Case Block	Determines next state by checking current state and other signals determining the path.	7 defined states (RESET, OFFSET_IDLE, READ_P1, READ_P2, WRITE_P1, WRITE_P2, PRE_RESET_IDLE) to indicate 7 primary behaviours. Initial state reset (RESET). Idles until CPU writes offset 0 to start module (OFFSET_IDLE). Read word from slave by providing address and additional state to hold signals steady for Avalon interconnect protocol (READ_P1, READ_P2). Write word to slave by providing address and additional state to hold signals steady for Avalon interconnect protocol (WRITE_P1, WRITE_P2). End state to tell CPU module is done (PRE_RESET_IDLE).
Output Case Block	Supplies module outputs defined by the current state as well as controls other signals governing the flow of the circuit.	Properly supplies the signals for each state. Controls other block signals.

Modules and testbenches: https://github.com/ryaanluke/DNN-Hardware-Accelerator/tree/main/task4

A Basic Deep Neural Network Accelerator

A Vector-vector dot product accelerator that computes the inner product *w*a*, adds a bias *b*, and optionally applies the ReLU activation function to the result. The CPU will provide address that this module needs to access through SDRAM. To set up the computation, the CPU will write addresses of the bias vector, weight matrix, input and output activations, and the input vector length to the following word offsets in the component's address range:

Word	*Meaning*
0	When written, starts accelerator; may also be read
1	Bias vector byte address
2	Weight matrix byte address
3	Input activations vector byte address
4	Output activations vector byte address
5	Input activations vector length
6	Reserved
7	Activation function: 1 if ReLU, 0 if identity

The component is also able to handle multiple requests following this design flow: if offsets 1-7 are not changed between two writes to offset 0, they keep their previous values.

dnn.sv has these following parameters:

Name	*Type*	*Semantics*
clk	Input	clock
Rst_n	Input	Reset
[3:0] slave_address	Input	Address from CPU master
Slave_read	Input	Enabled if CPU master wants to read
Slave_write	Input	Enabled if CPU master wants to write
[31:0] Slave_writedata	Input	Input from CPU master what we want to write
Master_waitrequest	Input	From SDRAM_Controller to see if its busy
[31:0] master_readdata	Input	Input from SDRAM slave once we're done reading
Master_readdatavalid	Input	Input from SDRAM slave if read_Data is valid
Slave_waitrequest	Output	Assert if busy waiting for some internal process to be done
[31:0] slave_readdata	Output	Output to CPU master what we just read
[31:0] master_address	Output	Output to SDRAM salve the address we want to access
Master_Read	Output	Output if we want to read to the SDRAM slave
Master_write	Output	Output if want to write to the SDRAM slave

[31:0] master_writedata	Output	Output to SDRAM slave of what we want to write

The module has these following design ideas:

Block	*Functionality*	*Details*
Address Counter/Assign Block	Triggered at the event of an asynchronous reset or positive edge of the clock.	Controls the assignment of activation vector, bias vector, output activation vector, weight vector addresses, as well as incrementing addresses we are accessing.
Vector length control block	Triggered at the event of an asynchronous reset or positive edge of the clock.	Assigns value of vector length.
Element counter blocks	Triggered at the event of an asynchronous reset or positive edge of the clock.	Controls counter for number of elements processed, both inner and outer product.
ReLU control block	Triggered at the event of an asynchronous reset or positive edge of the clock.	Controls whether ReLU will be applied or not.
State Machine Flow D-FlipFlop	D-flipflop controlling the flow of the state machine upon the trigger of asynchronous reset or positive edge of the clock.	Upon every clock cycle, the current state is moved to the next state. Upon an asynchronous reset, the current state is reset.
Next State Case Block	Determines next state by checking current state and other signals determining the path.	14 defined states (RESET, OFFSET_IDLE, READ_BIAS_ADDR READ_BIAS_ADDR_P2, READ_WEIGHT_ADDR READ_WEIGHT_ADDR_P2, READ_INPUT_ACTIVATION_ADDR READ_INPUT_ACTIVATION_ADDR_P2, A_MULT_B, APPLY_Q1616_PLUS_B, APPLY_RELU, WRITE_OUT_ACTIVATION_ADDR, CHECK_VECTOR_LENGTH_DONE, PRE_CPU_READ_IDLE) to indicate 14 primary behaviours. Initial state reset (RESET). Idles until CPU writes offset 0 to start module (OFFSET_IDLE). Read bias vector element, weight vector element, and input activation vector element from slave by providing address and additional state to hold signals steady for Avalon interconnect protocol (READ_BIAS_ADDR

		READ_BIAS_ADDR_P2, READ_WEIGHT_ADDR READ_WEIGHT_ADDR_P2, READ_INPUT_ACTIVATION_ADDR READ_INPUT_ACTIVATION_ADDR_P2). Apply a' = ReLU(W*a + b) in three steps (A_MULT_B, APPLY_Q1616_PLUS_B, APPLY_RELU). Write element into output activation vector (WRITE_OUT_ACTIVATION_ADDR). Check if all elements in vector are finished (CHECK_VECTOR_LENGTH_DONE). End state to tell CPU module is done (PRE_CPU_READ_IDLE).
Output Case Block	Supplies module outputs defined by the current state as well as controls other signals governing the flow of the circuit.	Properly supplies the signals for each state. Controls other block signals.

Modules and test benches: https://github.com/ryaanluke/DNN-Hardware-Accelerator/tree/main/task5

A Slightly Optimized Deep Neural Network Accelerator

To optimize the previous DNN accelerator, we add reuse functionality to reuse fetched input activations. A reuse key is important because these accelerators are connected to an external SDRAM, and SDRAMS are slow and costly to access in terms of energy in comparison to on-chip SRAM. Therefore, it is more energy efficient and faster to identify data that can be reused, copy it from DRAM to on-chip SRAM, and read it from SRAM many times. This additional memory, or "cache" can be generated using Intel Quartus Prime's Megafunction memory generator.

dnn.sv has these following parameters:

Name	*Type*	*Semantics*
clk	Input	clock
Rst_n	Input	Reset
[3:0] slave_address	Input	Address from CPU master
Slave_read	Input	Enabled if CPU master wants to read
Slave_write	Input	Enabled if CPU master wants to write
[31:0] Slave_writedata	Input	Input from CPU master what we want to write
Master_waitrequest	Input	From SDRAM_Controller to see if its busy
[31:0] master_readdata	Input	Input from SDRAM slave once we're done reading
Master_readdatavalid	Input	Input from SDRAM slave if read_Data is valid
Slave_waitrequest	Output	Assert if busy waiting for some internal process to be done
[31:0] slave_readdata	Output	Output to CPU master what we just read
[31:0] master_address	Output	Output to SDRAM salve the address we want to access
Master_Read	Output	Output if we want to read to the SDRAM slave
Master_write	Output	Output if want to write to the SDRAM slave
[31:0] master_writedata	Output	Output to SDRAM slave of what we want to write

The module has these following design ideas:

Block	*Functionality*	*Details*
Address Counter/Assign Block	Triggered at the event of an asynchronous reset or positive edge of the clock.	Controls the assignment of activation vector, bias vector, output activation vector, weight vector addresses, as well as incrementing addresses we are accessing.

Vector length control block	Triggered at the event of an asynchronous reset or positive edge of the clock.	Assigns value of vector length.
Element counter blocks	Triggered at the event of an asynchronous reset or positive edge of the clock.	Controls counter for number of elements processed, both inner and outer product.
ReLU control block	Triggered at the event of an asynchronous reset or positive edge of the clock.	Controls whether ReLU will be applied or not.
State Machine Flow D-FlipFlop	D-flipflop controlling the flow of the state machine upon the trigger of asynchronous reset or positive edge of the clock.	Upon every clock cycle, the current state is moved to the next state. Upon an asynchronous reset, the current state is reset.
Next State Case Block	Determines next state by checking current state and other signals determining the path.	14 defined states (RESET, OFFSET_IDLE, READ_BIAS_ADDR READ_BIAS_ADDR_P2, READ_WEIGHT_ADDR READ_WEIGHT_ADDR_P2, READ_INPUT_ACTIVATION_ADDR READ_INPUT_ACTIVATION_ADDR_P2, A_MULT_B, APPLY_Q1616_PLUS_B, APPLY_RELU, WRITE_OUT_ACTIVATION_ADDR, CHECK_VECTOR_LENGTH_DONE, PRE_CPU_READ_IDLE) to indicate 14 primary behaviours. Initial state reset (RESET). Idles until CPU writes offset 0 to start module (OFFSET_IDLE). Read bias vector element, weight vector element, and input activation vector element from slave by providing address and additional state to hold signals steady for Avalon interconnect protocol (READ_BIAS_ADDR READ_BIAS_ADDR_P2, READ_WEIGHT_ADDR READ_WEIGHT_ADDR_P2, READ_INPUT_ACTIVATION_ADDR READ_INPUT_ACTIVATION_ADDR_P2). Apply a’ = ReLU(W*a + b) in three steps (A_MULT_B, APPLY_Q1616_PLUS_B, APPLY_RELU).

		Write element into output activation vector (WRITE_OUT_ACTIVATION_ADDR). Check if all elements in vector are finished (CHECK_VECTOR_LENGTH_DONE). End state to tell CPU module is done (PRE_CPU_READ_IDLE).
Output Case Block	Supplies module outputs defined by the current state as well as controls other signals governing the flow of the circuit.	Properly supplies the signals for each state. Controls other block signals.
Cache memory instantiation	Instantiated an on-chip SRAM	Providing a local memory / cache system to store and retrieve reusable data
Cache valid control	Control signals and control block	Controls to determine whether data in local cache is valid or not. Cache is valid if the CPU provides a vector address to access and the address matches a previous search.

Modules and test benches: https://github.com/ryaanluke/DNN-Hardware-Accelerator/blob/main/task6/dnn.sv

Summary

This project explored building a deep neural network accelerator for an embedded Nios II system. After building the Nios II system, we built an additional wordcopy module that takes words from one place in memory and places them in another place. Additionally, we built a sophisticated DNN vector-vector dot product accelerator with a local cache reuse feature to reduce the energy and latency cost of fetching data from DRAM.

Programming / Lessons takeaways:

1. Using Intel Quartus Prime's Platform Designer to build a SoC
   - Used Intel documents to build an embedded softcore Nios II system.
   - Avalon Interconnect wiring.
   - Memory Mapped addresses.
   - PLL and SDRAM controller

2. Avalon Interconnect Protocol
   - Built modules that follow Avalon Interconnect's master-slave protocol

3. Testing with C program and Intel FPGA Monitor
   - Used given run_dnn.c program to test image processing deep neural network pretrained with a python script
   - Used Intel FPGA monitor to examine registers

4. Interfacing with different memories
   - Wrote modules that interfaced with local cache memories internally accessible as well as external memories that were needed to be accessed through master-slave protocol.

5. Writing state machines that had to iterate through memory locations
   - Multiple modules had to iterate and fetch content through memory byte by byte.
   - Complex state machine that had to account for memory timing will iterating through the memory.

6. Q16.16 fixed point arithmetic