Neural Net Hardware Accelerator using Verilog on Altera FPGA

Piyush Agarwal **Raunaq Nandy Majumdar**

Goal:

To accelerate the network inference on MNIST image dataset using techniques learnt during the course.

Problem Statement:

The reference software to run MNIST-NN on ALTERA DE2-115 board is given. There are two different networks architectures used in the reference software:

	Architecture	Layers	Accuracy	Inference Time
MNIST-MLP	Dense	1	97.59%	0.22 sec per image
MNIST-CNN	Conv2D	2	99.13%	27.58 sec per image

We have to study the reference software, profile the software to identify the computation bottlenecks and apply various techniques to reduce the computation time to achieve faster inference on both networks.

Results of Software Profiling

We used performance counters to profile the reference code. The computation time in some of the critical sections of the code is as below:

```
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|inference      | 100|  27.54395|    1377197633|        1|
+---------------+-----+-----------+---------------+-----------+
|conv_2d_loop2  | 86.7|  23.86823|    1193411281|        1|
+---------------+-----+-----------+---------------+-----------+
|kad_eval_marked| 100|  27.54318|    1377159003|        1|
+---------------+-----+-----------+---------------+-----------+
|kad_eval_at    | 100|  27.54320|    1377160016|        1|
+---------------+-----+-----------+---------------+-----------+
|kad_op_list    | 100|  27.54299|    1377149632|       13|
+---------------+-----+-----------+---------------+-----------+
|conv2d_1st     | 3.36|   0.92636|      46318156|        1|
+---------------+-----+-----------+---------------+-----------+
|conv2d_2nd     | 86.9|  23.93935|    1196967532|        1|
+---------------+-----+-----------+---------------+-----------+
```

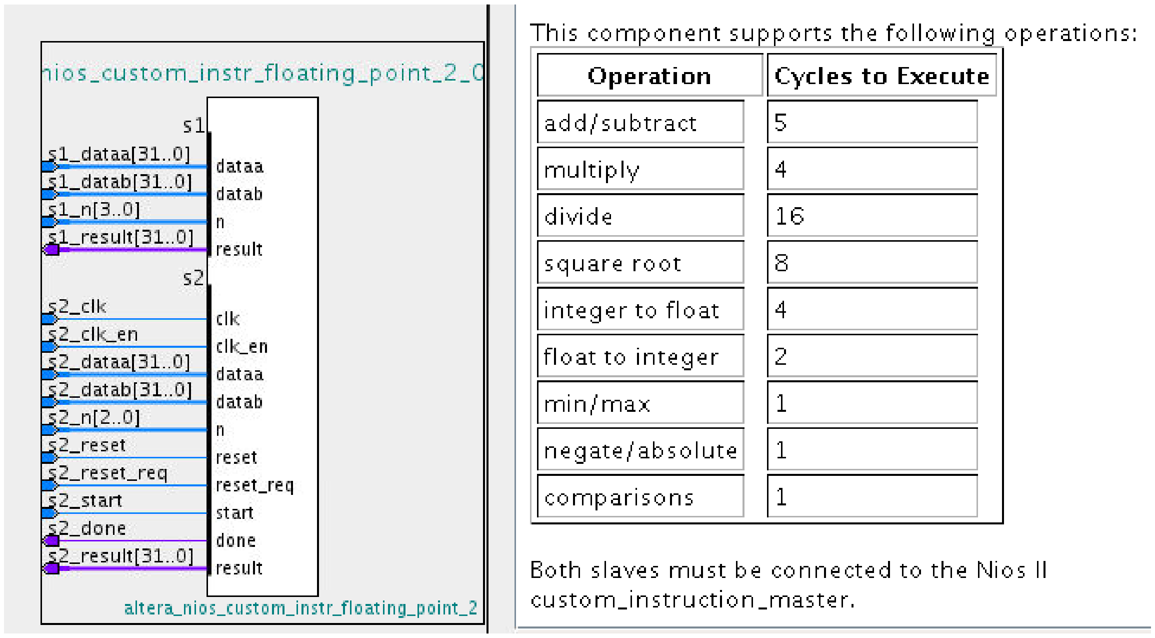
The conv_2d_loop2 is a performance counters added to the conv2d_loop2 function in kautodiff.c file. This functions calls kad_sdot function repeatedly which is performing the MAC operation. So we decided to optimize the kad_sdot function.

Methodology

- The kad_sdot function performs MAC operation and it is called in a for loop within conv2d_loop2 function in kautodiff.c. This is part of the convolution operation.
- The arguments for MAC operation are floating point numbers.
- To accelerate the MAC operation FP hardware needs to be available.
- We explored Nios II FP custom instruction hardware as well as added our own hardware for FP addition and multiplication.
- These modules are common for both custom instruction and hardware accelerator.
- The FP adder and multiplier is implemented in Verilog and tested in Questasim for functionality.
- Then we added a custom instruction to Qsys and measured performance.
- After then we added hardware accelerator and measured its performance.

FP support in Nios II

Nios II provides a custom instruction for floating point computation. The baseline implementation for the CNN took 27.54 sec to classify one image which with NIOS II FP custom instruction got reduced to 8.52 sec.



This component supports the following operations:

Operation	Cycles to Execute
add/subtract	5
multiply	4
divide	16
square root	8
integer to float	4
float to integer	2
min/max	1
negate/absolute	1
comparisons	1

Both slaves must be connected to the Nios II custom_instruction_master.

As can be seen in the above figure, the NiosII FP custom instruction takes 5 cycles for an addition and 4 cycles for a multiplication which we feel can be improved. Moreover, there are several other functions provided which we do not need so we thought of simplifying the floating point hardware by writing our own adder and multiplier logic.

Custom FP Arithmetic unit

We have used IEEE 754 32-bit protocol. Also, we only need addition and multiplication support. Separate modules for Addition and Multiplication have been created.

Multiplication:

- Mantissas are multiplied.
- Exponents are added.
- Sign & overflow are taken care of.

Addition:

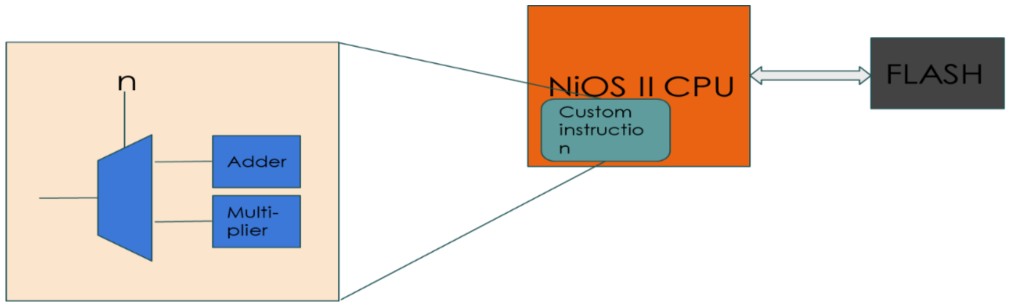
- First exponents are made equal.
- Mantissas are added. If mantissa[27:26] > 2 increase exponent by 1.
- Again, Sign & overflow are taken care of.

Custom Instruction

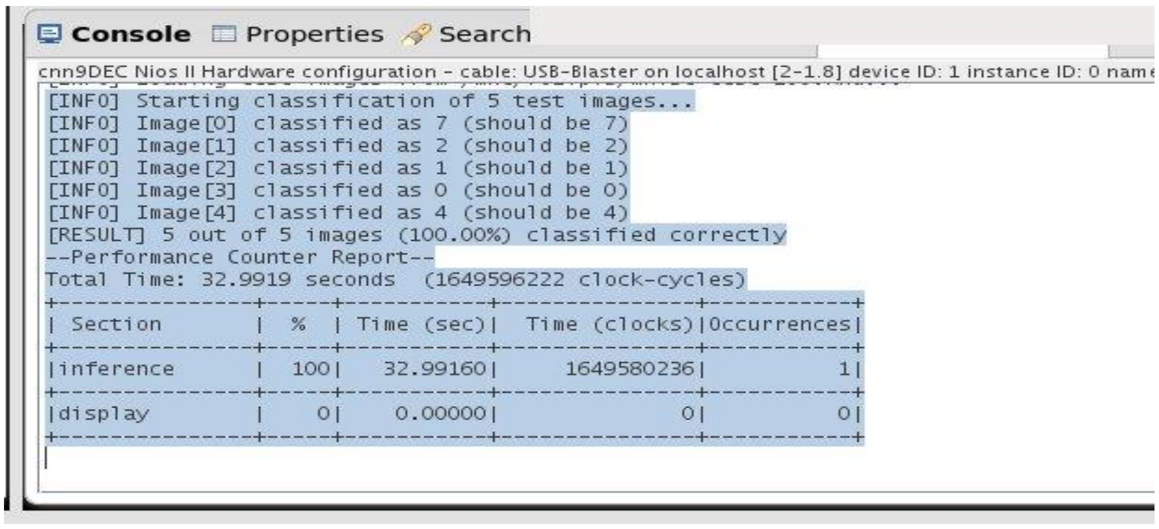
First, we tried custom instruction using the Nios II FP hardware, but this degraded the performance to 16.73 sec/image from 8.52sec/image which we were getting when we were just using the Nios II FP hardware. Thus, we are effectively losing on performance if we just add custom instruction without custom FP hardware.

To add custom instruction with our custom FP hardware, below steps were performed.

- Used extended combinational custom instruction wrapper.
- Wrapped FP multiplier and adder module in an extended combinational custom instruction.
- The CI is called from the for loop of kad_sdot function in kautodiff.c.
- For n = 0: The multiplier output is selected.
- For n =1: The adder output is selected.



Performance comparison with custom instruction		
Inference Time per image	**Baseline**	**Custom Instruction**
MLP	0.22 sec	0.04 sec
CNN	27.54 sec	6.4 sec



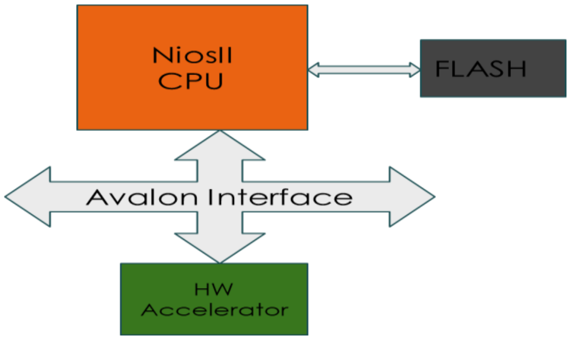
```
cnn9DEC Nios II Hardware configuration - cable: USB-Blaster on localhost [2-1.8] device ID: 1 instance ID: 0 name
[INFO] Starting classification of 5 test images...
[INFO] Image[0] classified as 7 (should be 7)
[INFO] Image[1] classified as 2 (should be 2)
[INFO] Image[2] classified as 1 (should be 1)
[INFO] Image[3] classified as 0 (should be 0)
[INFO] Image[4] classified as 4 (should be 4)
[RESULT] 5 out of 5 images (100.00%) classified correctly
--Performance Counter Report--
Total Time: 32.9919 seconds  (1649596222 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|inference      |  100|   32.99160|     1649580236|          1|
+---------------+-----+-----------+---------------+-----------+
|display        |    0|    0.00000|              0|          0|
+---------------+-----+-----------+---------------+-----------+
```

Result 1: Performance improvement due to the Custom instruction in CNN

Hardware Accelerator

For our hardware accelerator we used the same FP adder and multiplier and built a Verilog wrapper. The main steps in creating accelerator were.

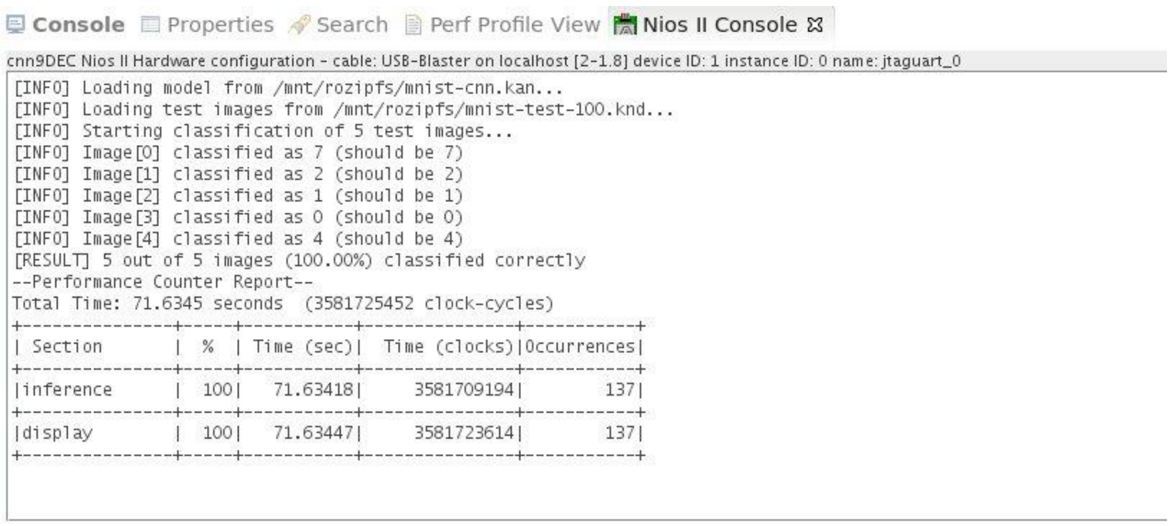
- Used Avalon-MM Simple slave for connecting to CPU.
- Transfer 96 values of X and Y using for loop in kautodiff.c using two different for loops.
- The input data is stored in register bank within accelerator.
- After X & Y are written signal end of transfer by writing to flag register.
- When flag register is written start MAC operation in hardware and waitrequest is asserted.
- The final sum is stored in a register and returned.



The number of cycles required for computation in HW accelerator once the data has been transferred is 96. We assert the waitrequest signal for this duration so that the sum cannot be transferred from HW accelerator to CPU. Once the computation is done, waitrequest is deasserted and CPU receives the final sum.

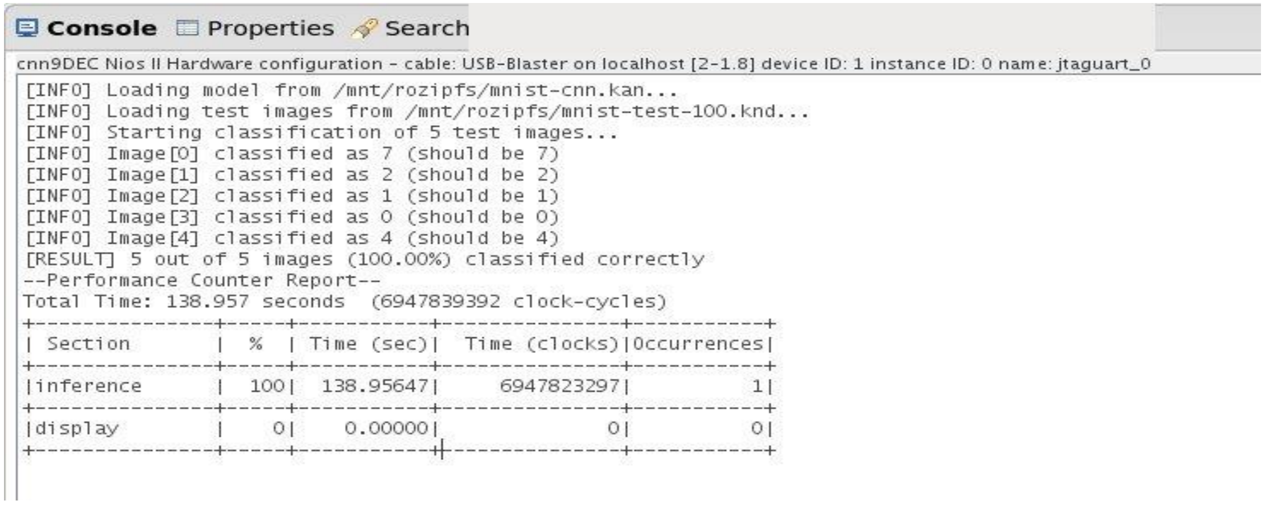
To transfer data values from CPU to accelerator, we have to bypass the cache as it doesn't assert the write signal on Avalon Interface until cache is bypassed.

Performance comparison with hardware accelerator		
Inference Time per image	**Baseline**	**Custom Instruction**
CNN	27.54 sec	14.32 sec



```
cnn9DEC Nios II Hardware configuration - cable: USB-Blaster on localhost [2-1.8] device ID: 1 instance ID: 0 name: jtaguart_0
[INFO] Loading model from /mnt/rozipfs/mnist-cnn.kan...
[INFO] Loading test images from /mnt/rozipfs/mnist-test-100.knd...
[INFO] Starting classification of 5 test images...
[INFO] Image[0] classified as 7 (should be 7)
[INFO] Image[1] classified as 2 (should be 2)
[INFO] Image[2] classified as 1 (should be 1)
[INFO] Image[3] classified as 0 (should be 0)
[INFO] Image[4] classified as 4 (should be 4)
[RESULT] 5 out of 5 images (100.00%) classified correctly
--Performance Counter Report--
Total Time: 71.6345 seconds  (3581725452 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|inference      |  100|   71.63418|     3581709194|        137|
+---------------+-----+-----------+---------------+-----------+
|display        |  100|   71.63447|     3581723614|        137|
+---------------+-----+-----------+---------------+-----------+
```

Result 2: Performance of the HW accelerator on the CNN algorithm



```
cnn9DEC Nios II Hardware configuration - cable: USB-Blaster on localhost [2-1.8] device ID: 1 instance ID: 0 name: jtaguart_0
[INFO] Loading model from /mnt/rozipfs/mnist-cnn.kan...
[INFO] Loading test images from /mnt/rozipfs/mnist-test-100.knd...
[INFO] Starting classification of 5 test images...
[INFO] Image[0] classified as 7 (should be 7)
[INFO] Image[1] classified as 2 (should be 2)
[INFO] Image[2] classified as 1 (should be 1)
[INFO] Image[3] classified as 0 (should be 0)
[INFO] Image[4] classified as 4 (should be 4)
[RESULT] 5 out of 5 images (100.00%) classified correctly
--Performance Counter Report--
Total Time: 138.957 seconds  (6947839392 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|inference      |  100|  138.95647|     6947823297|          1|
+---------------+-----+-----------+---------------+-----------+
|display        |    0|    0.00000|              0|          0|
+---------------+-----+-----------+---------------+-----------+
```

Result 3: Baseline performance of the CNN algorithm

Results

The combined results with all the implementations are given in below table.

Performance comparison with different approach		
Inference Time per image	**MLP**	**CNN**
Baseline	0.22 sec	27.54 sec
Nios II FP hardware	NA	8.52 sec
CI with Nios II FP hardware	NA	16.73 sec
Custom Instruction	0.04 sec	6.4 sec
Hardware Accelerator	NA	14.32 sec

Conclusion

From the results obtained and our work during the course of this project, we draw the following conclusions.

- While hardware implementation of any specific function gives good performance improvement, care has to be taken to use proper interface and consider communication overheads.
- By properly understanding the requirements, we can create efficient hardware as is evident by the difference in performance of the general purpose Nios II FP hardware and our custom FP hardware.
- If DMA is not used and there is a lot of data transfer from memory to hardware accelerator, the custom instruction is usually the better choice.

Challenges

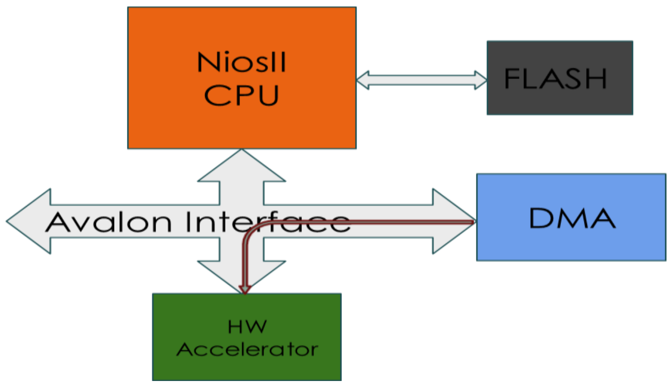
The following are the challenges we faced while implementing our ideas during the project.

- Debugging the Avalon interface connection for the hardware accelerator. We were not getting the write signal correctly so junk values were being loaded
- Understanding the functioning of the DMA controller and most importantly modifications needed to be made in the software for the correct implementation
- Understanding the reason of cache bypassing and cache flushing.

Further improvements

The performance of Hardware Accelerator can be improved if we use DMA to transfer data from memory to accelerator. The implementation with DMA performs as below.

- DMA directly sends the block of data to the accelerator reducing communication overhead and freeing up the CPU.
- The DMA slave gets signal from CPU to start transfer.
- The DMA master takes over bus and sends the data by the tx channel.
- DMA interrupts CPU on completion of transfer.
- HW Accelerator does its job and returns final sum to CPU.


NiosII CPU
FLASH
Avalon Interface
DMA
HW Accelerator