Puspal Chatterjee
ECEN Grid username: 695r43
Abhishek Nayak
ECEN Grid username: 695r48

SoC Design Course Project report: Neural Network Inference

Fall 2021

ABSTRACT

Neural Network inference is one of the commonly accelerated functions. Application-specific hardware is more efficient since it involves minimum complexity and needs minimal work to implement a given function.

In this project, we explored various architectural/system optimizations and designed a simple neural network accelerator to accelerate the inference on a database of 10000 images. The optimizations explored are HW Accelerators and Custom Instructions. We observed improved performance (speedup of 1.54x) using Custom Instructions compared to HW Accelerator implementation.

Additionally, we studied the variation of performance with different cache sizes. The results were according to our expectations, with the larger cache size delivering higher performance.

Finally, we tried implementing Direct Memory Access (DMA) to allow our HW subsystems to access the memory independently of the NIOS processor. Although we see a much-improved performance, we are getting unexpected inference results.

Platforms & tools used: DE2 FPGA board, Qsys SoC design tool, Nios II Configurable Soft processor, various peripherals from IP library, Nios II Software Integrated Development Env, VGA Display.

INTRODUCTION

The breakthrough of deep learning has started a technological revolution in various areas such as object identification, image/video recognition, and semantic segmentation. Neural networks, one of the representative applications of deep learning, have been widely used and developed many efficient models. But the high computation and storage complexity of neural network inference poses great difficulty in its application. CPU platforms are hard to offer enough computation capacity.

The usage of the FPGA (Field Programmable Gate Array) for neural network implementation provides flexibility in programmable systems. For the neural network-based instrument prototype in a real-time application, conventional VLSI neural chip design is limited in time and cost. With low precision artificial

neural network design, FPGAs have higher speed and smaller sizes for real-time applications than the VLSI design. The programmability of reconfigurable FPGAs yields the availability of fast special-purpose hardware for wide applications. Its programmability could set the conditions to explore new neural network algorithms and problems of a scale that would not be feasible with the conventional processor.

In this course project, we designed an SoC that implements a neural network inference (which is an application of moderate complexity) by applying the concepts of HW/SW partitioning. The working reference software on the DE2 platform and the trained networks was provided to us. The trained model was made using KANN C libraries [1].

HW/SW partitioning is one of the key topics in the System-on-chip design. It is a form of customization or specialization where we have to choose the best combination of (specialized) hardware and software to implement a given function. In designing embedded systems, design space exploration is performed to satisfy the application requirements. This can be achieved either by different architectural choices or by appropriate task partitioning. In the end, the synthesis process generates the final solution with a proper combination of Software, Hardware, and Communication Structures. The software part may consist of Operating System, Application code, and Drivers for peripherals. Similarly, the hardware consists of appropriate one or more processor cores with dedicated IP cores and communication buses. Design space exploration, from a system point of view, design space exploration can be performed by partitioning the application functionality into hardware and software components.

The architectural templates for HW/SW partitioning - *HW Accelerators* and *Custom Instruction* differ primarily in integrating the custom hardware with the processor. HW Accelerators/ Co-processors are connected to the system bus or a dedicated co-processor interface, whereas Custom Instruction is a fine-grained integration into the processor.

DMA does data transfer between memory and hardware accelerators. It acts as a master between two slaves. It shows superior performance over custom instruction and hardware accelerator because DMA completely bypasses CPU for data transfer and reduces overhead from CPU. Moreover, huge amounts of data can be transferred on a single go from memory and accelerator using DMA, resulting in a reduction of communication/transportation overhead.

INITIAL CONFIGURATION

We are provided with the MNIST-NN reference SW on Altera DE2-115 boards. The SW has been derived from KANN, a lightweight neural network library written in pure C. Two different types of neural networks are given:

1. MNIST-MLP, provided as an example network written with KANN APIs, consists of 1 hidden layer with 64 neurons and yields a 97.59% accuracy on 10000 MNIST test images.
2. MNIST-CNN, which incorporates two convolution layers (32 3x3 filters, ReLU) followed by a fully-connected layer (128 neurons), yields a 99.13% accuracy on the same test image set.

The CNN and MLP programs use 32-bit floating-point arithmetic since it offers the greatest dynamic range, making them suitable for any application. It would be the ideal number format to use.

This project's scope was limited to using the pre-trained model and accelerating the inference process.

Next, for further configuration, the steps involved were:

1. adding the necessary peripherals to the SoC from HW2 so that the SW running on the NiosII processor can access the on-board flash memory (where the pre-trained model and the test images will be stored)
2. creating a new NiosII application project with the reference SW and configuring the BSP project to support ROZIPFS (Read-Only ZIP File System)
3. creating a ROZIPFS image from the input files and programming the file system onto the flash memory, and
4. building and running the application on a DE2-115 board.

SOFTWARE PROFILING

In order to achieve a highly cost-effective system solution, it is very essential to perform performance estimation of Software and Hardware components. To increase design space exploration and estimate the software performance, it seems mandatory to use one or the other software profiling tool.

We have run the software profiling on CNN and MLP programs using the built-in **gprof** functionality in Nios II IDE.

For quick reference - the below data trace the functions in the CNN program taking up the majority of the time during the execution of the software program -

Index	%time	self	children	called	name
1	100.0	0.00	0.17		main[1]
2	98.1	0.00	0.17	100	kann_apply1[2]
3	97.1	0.00	0.17	100	kad_eval_at[3]
4	97.1	0.00	0.17	100	kad_eval_marked[4]
5	**94.1**	**0.15**	**0.01**	**206**	**kad_op_conv2d[5]**

Pointer to the complete profile data -
https://drive.google.com/file/d/10qj530iJNkCfdxCnwPSOxis4IS7FzUEn/view?usp=sharing

We observe that the function **kad_op_conv2d** accounts for 94.1% of the total execution time.

Similarly, from the profiling data of the MLP program, we found that the function **kad_sgemm_simple** accounts for 98.1% of the total execution time.

Pointer to the complete profile data -
https://drive.google.com/file/d/1nsVkw4l6WYh8h-UTzBaKvXTkbhzcpkoT/view?usp=sharing

Both of the above functions perform a huge amount of floating-point additions and multiplications using the loop functions of **kad_sdot** and **kad_saxpy_inlined**.

ARCHITECTURAL TEMPLATES FOR HW/SW PARTITIONING

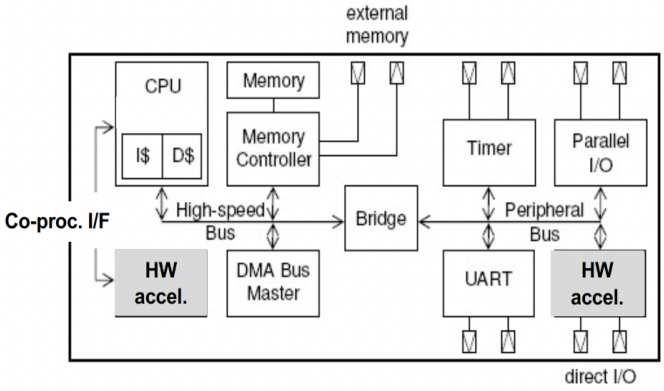
This section discusses the two primary architectural templates for HW/SW partitioning - HW accelerators and Custom Instructions.

Consideration	HW accelerator	Custom Instruction
How is custom HW interfaced?	System bus, Dedicated co-proc. I/F	Custom instruction interface Connects directly into processor pipeline
How does SW access the HW	Memory-mapped I/O Driver abstracts the operation performed by the accelerator as API for SW	Compiler maps operations to custom instructions. Intrinsics (macros)
Ability to access memory hierarchy	Direct access to main memory, cannot access cache/registers	Access typically limited to register file, the same view of memory as processor
Parallel execution with the processor	Yes, assuming SW does not block	Pipelining allows other instructions to execute concurrently
Granularity of computations targeted	Coarse (100s-millions of cycles)	Fine (few cycles - tens of cycles)

HARDWARE ACCELERATORS

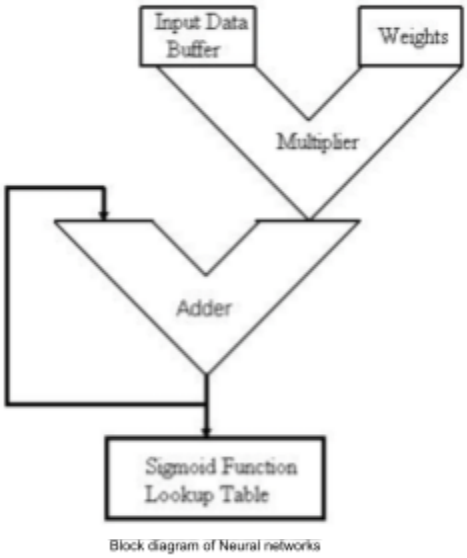
Accelerator is connected to the system bus or a dedicated coprocessor interface. This type of implementation requires minimal or no change to the processor.

The memory-mapped IO views the accelerator as memory, where a unique portion of physical memory address space is assigned to the accelerator. Loads/Stores to these addresses turn into communication transactions, which are conveyed from/to the accelerator by the communication architecture. The software running on the processor copies data from memory and writes results back (accelerator only needs to be in bus slave).



Key steps in designing accelerators

The neural network arithmetic architecture was developed using Verilog with 32-bit floating-point arithmetic.

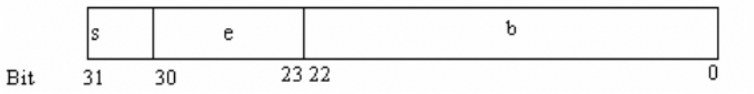


Block diagram of Neural networks

The floating-point adder and multiplier blocks were combined into a single HW MAC unit.

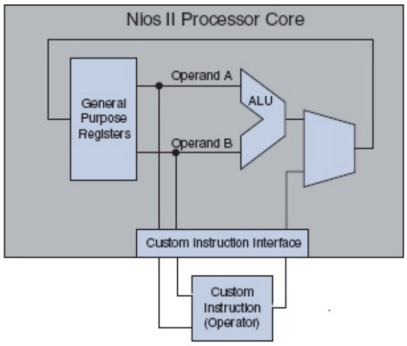
- Cycle 1 - 1st input is fed
- Cycle 2 - 2nd input is provided, and the multiplier result is computed
- Cycle 3 - 3rd input is fed, previously calculated multiplier result is used, and the addition result is computed.

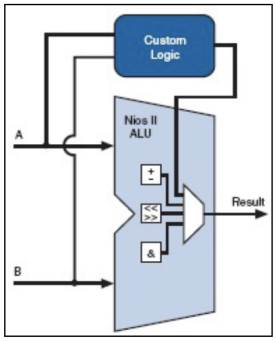
The FP adder and multiplier are according to the IEEE-754 standards for single-precision (32-bit) floating-point arithmetic. The single-precision floating-point numeric representation supports to IEEE-754 standard shown in the below figure.



The floating-point number (n) is computed by: $-1^{s}\,2^{|e-127|}(1.b)$. In the formula, the sign field referred to as 's' is bit 31 and is used to specify the sign of the number. Exponent field is referred to as 'e' is bits 30 down to 23 are the exponent field. The bias of 127 is used. Because the 8-bit quantity is a signed number representation. To store binary representation (b) of floating-point number bits 22 down to 0 are used. The leading one in the mantissa is implicit. So the mantissa is (1.b).

CUSTOM INSTRUCTION





Custom Instruction is a fine-grained integration into the extensible processor of NiosII. Custom instructions specify hardware added to the EX stage of the processor (parallel with the ALU).

This sort of implementation provides limited automation.

- HW designer manually designs the HW to adhere to the specified custom instruction interface
- Programmer manually changes the SW to use the custom instructions

We have implemented a combinational custom instruction, which takes one cycle in the EX stage.

The processor presents operands (dataa, datab) at one clock edge, and samples result at the next clock edge. An important point is that the longest path through the custom instruction HW directly impacts the clock period.

This project has implemented separate floating-point adder and multiplier units according to the IEEE-754 standards for single-precision (32-bit) floating-point arithmetic.

DIRECT MEMORY ACCESS

Direct memory access (DMA) is a method that allows an input/output (I/O) device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory operations.

We have used DMA to transfer image buffer and weight buffer from memory to accelerator directly on a single go. This will minimize the communication overhead and also reduce the workload from the processor. We have programmed proper memory-mapped addresses and control registers. DMA will act as a master and be connected to the accelerator slave. The accelerator will perform the MAC operations and return the result to the desired address.

RESULTS

We have achieved experiments to compare the performance of the HW accelerator and Custom instruction. Through our analysis, we can see that Custom instruction shows superior performance over HW accelerators.

PERFORMANCE OF HW ACCELERATOR

In this experiment, we have demonstrated the performance improvement of the HW accelerator over SW for MLP and CNN programs. HW accelerator shows **3.06x** improvement over SW for MLP and **3.12x** for CNN.

Program	Optimization	Runtime (secs)	SpeedUp
MLP	SW	22.38	3.06
	HW Acc	7.32	
CNN	SW	2714.1	3.12
	HW Acc	869.21	

PERFORMANCE OF CUSTOM INSTRUCTION

In this experiment, we have demonstrated the performance improvement of the Custom Instruction implementation over SW for MLP and CNN programs. HW accelerator shows **4.73x** improvement over SW for MLP and **4.81x** for CNN.

Program	Optimization	Runtime (secs)	SpeedUp
MLP	SW	22.38	4.73
	Custom Inst	4.73	
CNN	SW	2714.10	4.81
	Custom Inst	563.80	

HW ACCELERATORS VS. CUSTOM INSTRUCTION

In this experiment, we have demonstrated the performance improvement of the HW accelerator over Custom Instruction for MLP and CNN programs. HW accelerator shows **1.547x** improvement for MLP and **1.542x** for CNN.

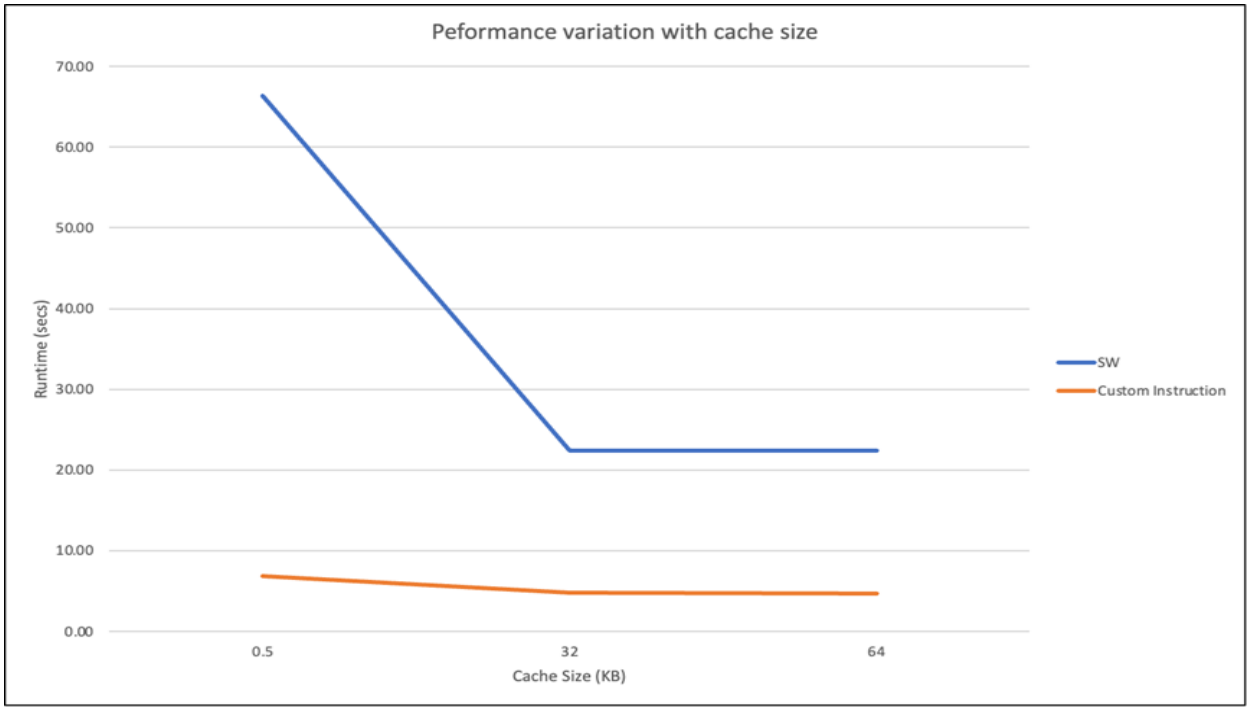
Program	Optimization	Runtime (secs)	Speedup
MLP	HW Acc	7.32	1.5470
	Custom Inst	4.73	
CNN	HW Acc	869.21	1.5417
	Custom Inst	563.80	

The performance improvement using Custom instruction is more than that of the HW accelerator. This is because Custom instruction is tightly coupled with the processor pipeline and communication/transport overhead of data is less in the case of custom instruction. HW accelerator requires multiple iterations of communication overhead between processor, memory, and accelerator, which is reduced in the case of custom instruction.

Moreover, in the HW Accelerator, the cache is bypassed using the bit-31 cache bypassing technique to store the computed results into the memory. There will be a guaranteed performance hit due to cache bypassing, as we studied from the below experiment.

PERFORMANCE VARIATION WITH CACHE SIZE

In this experiment, we have explored the performance variation of Custom Instruction optimization with respect to the Cache size. Both data cache and instruction cache sizes were varied as 512 B, 32 KB, and 64 KB.



As we can see in the above graph, SW performance improves significantly when the cache size is changed from 512 B to 32 KB. The performance further increases slightly from 32 KB to 64 KB cache size variation. For Custom instruction, performance improvement is there when the cache size is increased from 512 B to 32 KB and further to 64 KB. However, the rate of improvement is less as Custom instruction already exploits the scope of improvement. Overall, the custom instruction shows superior performance with a 64 KB cache size.

Moreover, the size of datasets is less than 32 KB. Therefore, most performance improvement is exploited using 32 KB cache size. However, 64 KB gives slightly better results but not that much.

PERFORMANCE OF DMA

Cache coherence problems may occur between the memory and data cache of the processor. As DMA bypasses the processor for its operation, the latest data may be present in the cache, whereas memory holds old data. This condition may lead to wrong/garbage data being used in hardware acceleration. Our DMA design and simulation observed **10x** performance gain over custom instruction. However, inferred data of MLP is coming incorrectly, most probably because of the above reason.

To fix the issue, one solution can be to flush the processor cache, which will result in updating the latest data from the cache to memory. Therefore, it will resolve the problem. However, frequent flushing of cache can result in degradation of overall performance. The other solution can be to bypass cache by setting the MSB of address as 1 in the design.

CONCLUSION

We have performed profiling to identify which all functions to accelerate to improve performance. Then, we performed HW acceleration using custom instruction and the HW accelerator. Both of them show performance improvement over given SW. However, Custom instructions show better gain over HW acceleration.

For further performance improvement, we can try unrolling the loop (where MAC operations happen) in the software code. We can also achieve less execution time by reusing the data.

REFERENCE

[1] KANN - a lightweight C library for artificial neural networks. GitHub - attractivechaos/kann: A lightweight C library for artificial neural networks