HW5 Domain-SpecificAccelerator

郭宗信, 111550105

***Abstract*—This report presents the integration of a domain-specific accelerator (DSA) into the Aquila RISC-V platform to enhance the computation speed of a Convolutional Neural Network (CNN). We implement a hardware acceleration solution using a Xilinx floating-point IP core, focusing on optimizing the convolution and fully-connected layer operations. The system interfaces with memory-mapped I/O (MMIO) and employs AXI-Stream protocol for efficient data transfer. Our implementation evaluates non-blocking mode of the floating-point IP, demonstrating performance differences in the MNIST handwritten digit recognition task. Experimental results show notable speed improvements in CNN operations (about 6 time faster), while highlighting the impact of data transfer overhead between the processor and accelerator.**

Keywords—RISC-V, Aquila SoC, Domain-Specific Accelerator (DSA), Convolutional Neural Network (CNN), AXI-Stream, FPGA, Hardware Acceleration, Floating-Point IP, MNIST, Neural Network Inference

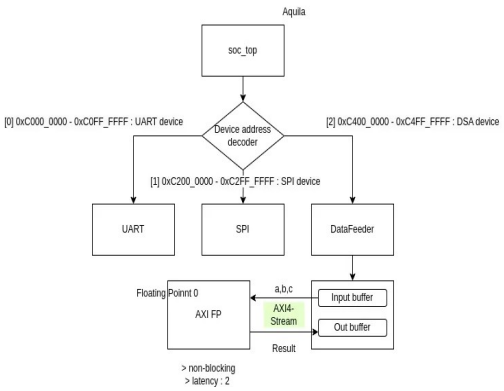
I. INTRODUCTION

Domain-Specific Accelerators (DSAs) are hardware accelerators designed to optimize specific computational tasks, offering better performance efficiency compared to general-purpose processors. In this project, we implement a DSA within the Aquila RISC-V SoC to accelerate Convolutional Neural Network (CNN) computations, specifically focusing on floating-point operations. Our implementation integrates a Xilinx floating-point IP core through the AXI-Stream interface, with data and model weights loaded from an SD card. The system includes both hardware and software modifications: hardware integration through memory-mapped I/O (MMIO) and software adaptations to utilize the acceleration. We evaluate the implementation using the MNIST dataset for handwritten digit recognition, comparing performance between the original software implementation and our hardware-accelerated version.

II. SYSETM ARCHITECTURE

A. System Overview

The acceleration system is implemented on the Aquila RISC-V platform, integrating a Xilinx floating-point IP core specifically for accelerating CNN operations. The system architecture comprises three main components, each serving a distinct role in the acceleration process. Below is the simple figure demonstrating the architecture.



1. Core Components:

- Aquila RISC-V Core: Functions as the main controller
- Data Feeder Module: Manages data flow and buffering
- Xilinx Floating-Point IP: Executes accelerated computations

2. Memory Organization:

- **DSA Device**: 0xC4000000 - 0xC4FFFFFF
- **SD Card Controller**: 0xC2000000 - 0xC2FFFFFF
- **UART Device** :0xC0000000 - 0xC0FFFFFF

B. Interface Design

The system employs two primary interface protocols. For Memory-Mapped I/O (MMIO), it enables Aquila core to control the accelerator, manages device addressing and control signals, and handles read/write operations between components.

The AXI4-Stream Interface provides high-throughput data transfer, connects data feeder to floating-point IP, and supports both blocking and non-blocking modes.

For signals related to I/Os, it includes strobe as the activation signal to trigger device operations, addr for target address for device access, and we (write enable) which controls read/write direction where 1 is for write and 0 is for read. The byte_enable specifies which bytes in a 32-bit word are active, while ready serves as a device status signal indicating it can accept new requests. Finally, data_in represents data to be written to the device, and data_out represents data read from the device.

C. Data Path Organization

The data path is organized to optimize CNN computations. The Input Path begins with SD card loading model weights and test data, followed by the data feeder buffering incoming data, and AXI4-Stream transferring data to floating-point IP.

The Processing Path involves floating-point IP performing multiply-accumulate operations, supporting vector operations for CNN computations, and returning results through AXI4-Stream interface.

D. FLOATING-POINT IP

The Xilinx floating-point IP core in our implementation supports both blocking and non-blocking operational modes, each offering distinct characteristics for CNN computations. However, in this report, we primarily focus on non-blocking mode.

The Non-Blocking Mode operates without a back-pressure mechanism and uses a simplified AXI4-Stream interface that

only requires TVALID and TDATA signals, with no TREADY signal needed. For computation triggers, it executes when all input channels show valid data, with operations occurring every enabled clock cycle and output generating regardless of downstream status.

The Blocking Mode implements a full flow control mechanism and uses a complete AXI4-Stream interface with TVALID, TDATA, and TREADY signals, including back-pressure support. Its data flow management includes an internal buffer system for input/output, where operations only start with fresh data on all channels and prevent data loss through flow control.

III. Implementation

A. Hardware Design

The DataFeeder Module serves as the bridge between Aquila core and floating-point IP, with its base address starting at 0xC4000000. It implements control and data paths using memory-mapped registers, handles data synchronization between software and hardware, and manages data flow for weight and input vectors.

As for the Floating-Point IP Integration, it is configured in non-blocking mode for maximum throughput. It features three input channels for data streams (a_data, b_data, c_data) and a single output channel for results. The integration uses direct AXI-Stream interface connection and is optimized for CNN's multiply-accumulate operations.

B. Software Design

The Memory Management system employs direct memory mapping for hardware control registers, with four primary control registers: the Trigger register at 0xC4000000, Weight data register at 0xC4000004, Input data register at 0xC4000008, and Result register at 0xC400000C.

For CNN Operation Implementation, it features a modified convolution function for hardware acceleration, utilizing direct register access for data transfer. The implementation includes synchronization through hardware trigger mechanism, along with result accumulation and management, and error handling for computation validation.

The Control Flow encompasses the hardware initialization sequence, data transfer coordination, operation triggering and monitoring, and result collection and processing.

C. Additional - Fully-Connected Layer Acceleration

The fully-connected layer acceleration implementation uses a straightforward state machine approach with two states (IDLE and CALC) for computation control. The hardware interface is organized around three memory-mapped addresses: 0xC4100000 for weights, 0xC4200000 for input data, and 0xC4300000 for results. The data feeder module manages data flow between memory and the floating-point IP while the software layer maintains the neural network structure and handles bias addition.

D. Profiler

The Profiler tracks 3 cycle counts:

-transfer_cycles: Time spent on data input (A, B, C)

-computation_cycles: Time used for computation

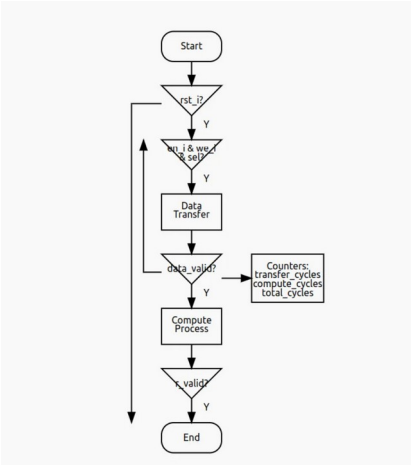
-total_cycles: Total time including all operations

Operation flow:

1.transfer_cycles starts when writing A/B/C data (data_feeding=1), stops when data_valid=1

2.computation_cycles starts when data_valid=1, stops when r_valid=1

3.total_cycles increments during any active operation (en_i || compute_flag || data_feeding)



IV.
IV.
IV.
IV.
IV.
IV.
IV.
IV.
IV.
IV.
IV.
IV.
IV.

Result

-Achieved 6.17× acceleration using Conv3D with FC layer

-Reduced processing time by 84% (21,445 → 3,473)

-Maintained 95% accuracy across optimizations

A. Performance Enhancement Analysis

First, we evaluate the acceleration performance of different implementations:

TABLE I. TABLE

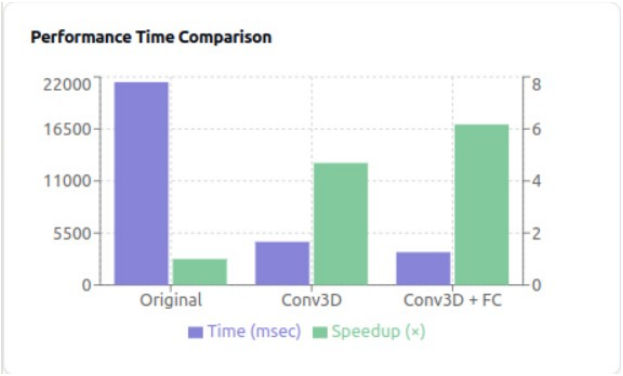


Fig. 1.

B. *Size checking*

Using `isnan()` for Number Validation

This implementation uses the `isnan()` function to validate numerical values, which requires including the math library (`#include <math.h>`). When invalid values are detected, the system outputs a warning message.

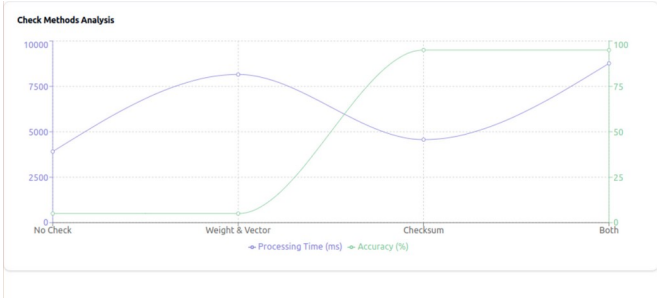
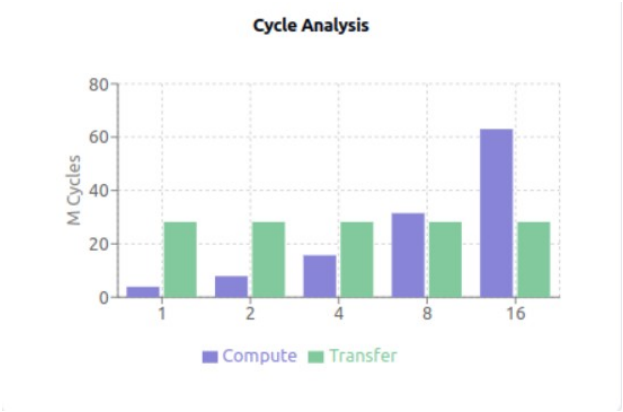


Fig. 2.

C. *Latency analysis*

In this analysis, I measured the non-blocking floating-point IP latency and analyzed both computation and transfer cycles. A key observation is that accuracy remains stable at 95% across different thread configurations, with one exception: when IP latency equals 1, accuracy drops significantly to 5%.



V. Discussion

A. Performance Analysis:

The Conv3D implementation achieves 4.69× speedup, while Conv3D with Fully-Connected Layer reaches 6.17× speedup. This improvement likely stems from better parallel processing and optimized data flow. The maintenance of 95% accuracy suggests effective error handling and computational precision. IP Latency Impact: The interesting relationship between IP latency and accuracy reveals critical system behavior:

Latency=1 shows poor accuracy (5%), suggesting insufficient processing time for stable computations Latency≥2 maintains 95% accuracy, indicating a minimum threshold for reliable floating-point operations Transfer cycles remain stable (~28.2M), implying memory bandwidth isn't a bottleneck Computation cycles scale with latency, showing direct correlation with processing depth

B. Validation Method Analysis:

When olny handle Conv3D, no validation trades reliability for speed (3,913 msec, 5% accuracy) Input validation adds overhead without accuracy benefit Checksum validation provides optimal balance (4,569 msec, 95% accuracy) Combined validation shows similar accuracy but higher overhead.

However, an intriguing phenomenon emerged: While validation was designed to signal illegal data, these signals weren't present in output. However, removing validation caused accuracy to drop from 95% to 5%, and the process could only be modified by adding checking instructions.

We hypothesize this behavior relates to data transfer latency, where validation instructions inadvertently compensate for transfer delays. This effect remains consistent regardless of IP latency settings.

This suggests validation serves dual purposes: data verification and timing compensation, indicating system synchronization is as crucial as data validation for computational accuracy.