ENABLING ACCELERATOR-SOC CO-DESIGN USING RISC-V CHIPYARD FRAMEWORK

by

Shruthi K Muchandi

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2020

Approved by:

____

Dr. Hamed Tabkhi

____

Dr. Fareena Saqib

____

Dr. Ronald Sass

ABSTRACT

SHRUTHI K MUCHANDI. Enabling accelerator-SoC co-design using RISC-V Chipyard framework. (Under the direction of DR. HAMED TABKHI)

The rise in transistor cost in conjunction with the slowdown of Moore's law has increased the demand for scalable SoC (System-on-Chip) based frameworks. The opportunity provided by reconfigurable and extensible full-system frameworks opens up a wide domain in research and academia for architecture exploration and customization. Additionally, with today's enormous increase in applications of machine learning, specifically deep learning at the edge which requires flexible real-time cognitive processing, inclines us to have efficient architectures with conflicting combination of high-performance and low-power utilization. This leads us to have scalable, latency aware domain-specific architectures.

However, accelerator design and optimization are often done in isolation considering the best possible constraints. This results in system integration becoming much more complicated due to the non-ideal performance differences. Therefore, this work focuses on enabling a design space exploration platform for processor accelerator co-design in order to achieve the best possible performance on the targeted systems. This work presents parameterizable system support for any streaming, data-hungry accelerators using Chipyard, an open-sourced, extensible, RISC-V based, agile, full-system hardware design and evaluation framework developed by University of California, Berkeley. With this work's RISC-LCAW (RISC-V Loosely-Coupled Accelerator Wrapper) contribution, we try to ease the manual effort and engineering behind the accelerator system integration by proving an accelerator integration socket in Chipyard framework. This wrapper template is designed with a focus on data-hungry, streaming, loosely-coupled hardware accelerators.

Also, we enable the co-design support for AWARE-DNN accelerator, developed in TeCSAR (Transformative Computer Systems and Architecture Research Lab) at University of North Carolina at Charlotte by using RISC-LCAW to integrate with the RISC-V based Rocket Chip SoC system. This integration benefits the accelerator configurability and optimizability by providing the system integration interpretation. AWARE-DNN accelerator offers an automated, re-configurable workflow for generating application specific architectures based on the inherent data-flow of targeted application and user specified real-time requirements. This combination fuels the expandability to explore the vital domains of power, latency and performance for the targeted system. Furthermore, some of the possible system-level effects on the accelerator latency and throughput are elevated compared to optimizing the accelerator in isolation. We evaluate this integration using Verilator RTL simulator for three sizes of convolution networks and see that the end-to-end latency of RISC-AWARE compared to standalone AWARE-DNN accelerator is 1.7× for 64×64×3, 2.6× for 32×32×3 and 1.2× for 11×11×3 image size.

DEDICATION

I would like to dedicate this work to the open-source community which motivated me and kindled the interest in me to pursue research in this domain and provided support when needed. Also, to my family and friends who showed deep trust and encouragement during all phases of my academic career.

ACKNOWLEDGEMENTS

With an immense pleasure, I would like to express my deep and sincere gratitude to my thesis advisor Dr. Hamed Tabkhi for his encouragement and support throughout my Masters study and thesis. This work would have not been possible without his motivation, patience and knowledge while guiding me in our research and writings. Besides my advisor, I would like to thank the rest of my thesis committee, Dr. Ronald Sass and Dr. Fareena Saqib for their valuable time and advisement. I also want to thank my peers at TeCSAR (Transformative Computer Systems and Architecture Research) for constant support and encouragement. Finally, I appreciate my family for their love which gave me the strength to pursue my Masters.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

An SoC is a single substrate electronic system which integrates all the hardware components and software functionalities achieving high performance and reliability at low power and area. This quality increases the demand for having SoCs in mobile and edge computing markets. Open-sourced, customizable SoC frameworks are favored enormously in the research and academia, as they push the boundaries of customization and experimentation. One of the best available resources is the RISC-V based Chipyard framework[1]. This agile based SoC generator framework offers full-system flexibility to customize hardware and the corresponding software functionalities. It offers a one stop approach for building our own custom chips.

With the slowdown of Moore's law and Dennard scaling, domain-specific architectures are considered the new golden age of computer architecture[2]. This class of architectures enables us to tailor the hardware design with respect to a specific domain of applications. They are not completely similar to ASICs as they are not completely specific for a single application, but they aim for a discipline of applications. It is already a proven fact that the performance benefits for those targeted applications is massive compared to running them on general-purpose computers.

Neural Networks are human-brain inspired algorithms for which development and applications picked up at a rapid pace from the past decade. Availability of huge amounts of data, advances in dedicated, efficient and powerful hardware and vast development in algorithms was the major influence for the AI inspired trend. Recently Deep Neural Networks (DNNs) have found wide spread usage in many emerging smart applications. This has been seen to be especially true for applications involving computer vision, speech recognition, robotics, image processing and so on. As show in

[3, 4], the performance of deep learning based approaches increases significantly with increments in data size. These applications command scalable, real-time processing, which generally comes with high computational complexity cost at lower power and high-performance metrics. Hence, efficient system architecture design for DNNs is an important step towards enabling deployment of these architectures in the domain of real-time edge intelligence. SoCs are one of the most widely used platforms for operating these architectures for providing high-performance and reliability at low power domains[5, 6]. SoC architects manage to tailor the complete system architecture for targeted applications based on the algorithm behavior, core, hardware accelerators and interconnect configurations to produce an efficient and reliable end product.

1.1 Motivation

SoC based designs are one of the best approaches for balancing area, power, and throughput as well as providing an organized platform for hardware-software co-design. With the increasing demand for high-performance and low power computing, the number of hardware accelerators deployed in mobile and server systems is huge. However most accelerators are designed and optimized in isolation as an IP block which gives a deceptive interpretation of the overall performance measure for the complete system. Generally, hardware IPs are interfaced to the system as an input-output device on a system bus which uses a DMA interface for communication while the control is managed by the host core. This leave the integration pretty straightforward, but the estimated performance gain might not be the same as it was with the standalone accelerator optimizations due to overheads from data movement, memory coherency management, and bus contention. To integrate a hardware accelerator in a multi-core heterogeneous system presents even more challenges. Ideally, the system has to be designed in a way that balances the compute throughput with the memory interface bandwidth. Hence, a co-designing approach for Soc based accelerators would provide more authentic view on the performance insights. This thesis is motivated by this idea and puts effort in providing an easier co-design platform for the targeted system and hardware accelerator.

Additionally, having a parameterizable full-system generator and evaluation framework is a huge opportunity for the current academia, industry and research communities due to enabling the exploration of various architecture design possibilities. RISC-V Chipyard is one of the most successful open-source agile full-system design and evaluation frameworks. However, while the hardware accelerator integration support is well-defined, the integration into the system generator platform requires a great manual engineering effort. In order to increase the productivity of custom accelerator integration, having a generic hardware wrapper would be very beneficial and time saving.

1.2 Contributions

1. Through this thesis contribution we provide a better understanding of different accelerator coupling approaches and interfaces supported in RISC-V based Rocket Chip system. Also, we present a system-level integration diagram for Rocket Chip which shows possible types of accelerator coupling.

2. In order to ease the manual effort and engineering behind the accelerator system integration in RISC-V Chipyard framework, we design RISC-LCAW (RISC-V Loosely-Coupled Accelerator Wrapper). This wrapper provides a template to integrate any streaming, data-hungry custom accelerators with the parameterizable and extensible RISC-V hardware ecosystem. We limit our scope of accelerators to streaming loosely coupled accelerators. Along with the accelerator controller template, we also provide support for DMA interface and clock-crossing through the wrapper design.

3. We present a RISC-AWARE design, in which we provide system co-design platform and design-space exploration flexibility for AWARE-DNN accelerator. AWARE-DNN is a latency-aware, real-time configurable deep neural network architecture

framework. We integrate this hardware accelerator to the RISC-V Rocket Chip SoC and elevate some system-level effects on the hardware accelerator performance.

1.3 Thesis Outline

The outline of this thesis document is as follows. The background and related work needed for this study is reviewed in Chapter 2. Along with briefing over deep learning and neural networks, this chapter mainly discusses the open-sourced RISC-V ecosystem and the AWARE-DNN accelerator. We additionally include references to some related works for this domain. Chapter 3 presents our first two contributions, which detail on providing a systematic understanding about the RISC-V hardware system's extensibility and RISC-LCAW design. Chapter 4 details on RISC-AWARE design which is to enable system co-design support for AWARE-DNN accelerator and also elevates some system-level effects on the hardware accelerator performance. In Chapter 5, we present our experimental setup and compare the performance of AWARE-DNN accelerator in system and standalone settings. Next, we conclude the thesis and discuss some interesting future exploration possibilities in Chapter 6.

CHAPTER 2: BACKGROUND AND RELATED WORK

In this chapter, we briefly overview the RISC-V Chipyard framework, deep learning and neural networks, and the AWARE-DNN accelerator. Additionally, we also discuss the related contributions in the research community.

2.1 RISC-V Chipyard Framework

Chipyard is an open-source, RISC-V based, SoC generator system developed and maintained by the Berkeley Architecture Research Group at the University of California, Berkeley. It supports agile development for full-system hardware design and evaluation. Chipyard is based on a huge composition of tools, libraries of generators and supports various flows as shown in Figure2.1. It contains Scala based tools and libraries like FIRRTL[7], Treadle[8], and ChiselTesters to support a generator-based architecture design developed in Chisel using Scala programming language primitives. Due to this, currently it is seen as a one-stop solution for custom architecture design and exploration.

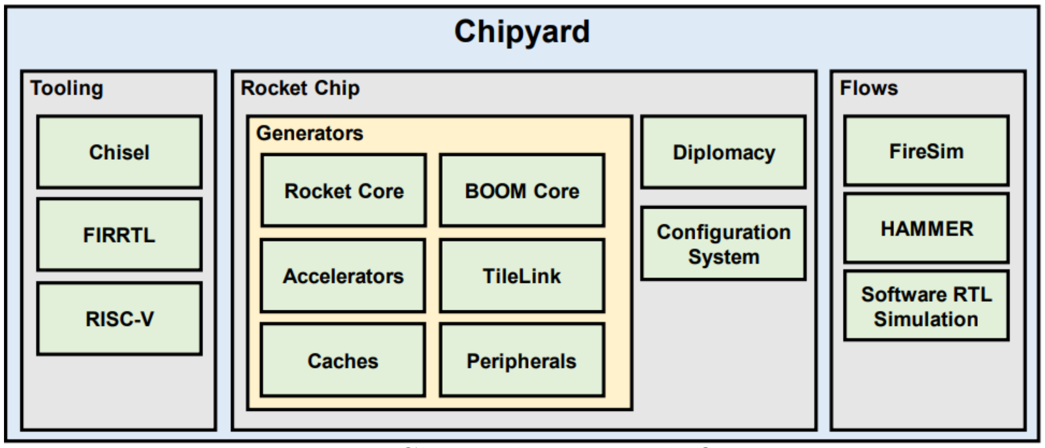


Figure 2.1: Chipyard Framework Overview

Chisel[9] is an open-source hardware construction language developed to support parameterizable generator based hardware designs. It is embedded in Scala which adds support for high level Object-Oriented and functional primitives. When the Chisel code is ready to be compiled, it uses SBT for building the source code. Since Chisel is derived from Scala it is considered to be a library within Scala and hence uses the same build tool for compiling the source code. The next step is hardware design elaboration where Chisel tries to construct a circuit based on the code, supported parameters, library functions and configurations. Once the elaboration is successful, the generator produces a FIRRTL output which is an intermediate representation of the circuit source code. FIRRTL is used to translate Chisel-based source files to Verilog representation. It again goes through various steps which optimize the design and finally outputs the Verilog file which can be used for running a simulation or FPGA implementation. Also, as mentioned in [10] FIRRTL leverages reusabilty by enabling RTL customization for the underlying complexity. Chipyard also supports Treadle, which is a circuit simulator which executes FIRRTL directly. Additionally, for testing Chisel-based designs, Chisel Testers is supported by Chipyard. Chisel Testers makes use of Scala APIs for DUT interaction and works with multiple backends such as Treadle and Verilator.

The RTL generators in Chipyard framework are designed using Chisel which is a mix of meta-programming and standard RTL. Normally, hardware designs using standard RTL are a single instance of the design emitted my generator. However, due to the meta-programming and parameterization support, the generator design is capable of integrating complex hardware in an automated format. It is capable of generating complex RISC-V based SoCs[11], including in-order and out-of-order[12] processors, uncore components, vector co-processors and hardware accelerators. The framework not only includes the initial complex designs but also allows flexible extensions to support user-specific customizations for any system component. Along with this, it leverages agile end-to-end computer architecture research with a single re-usable toolchain by providing a tool-base for automated VLSI flow[13] and also development of custom target software workloads[14] for bare-metal and linux based systems, to run on FPGA accelerated cloud simulators[15, 16, 17] or software RTL simulators like Verilator[18] or VCS. Due to its strong assembly of tools and libraries which ease the developer to customize the architecture design and integration as per the user needs, Chipyard is currently gaining popularity across the globe in industry and academia.

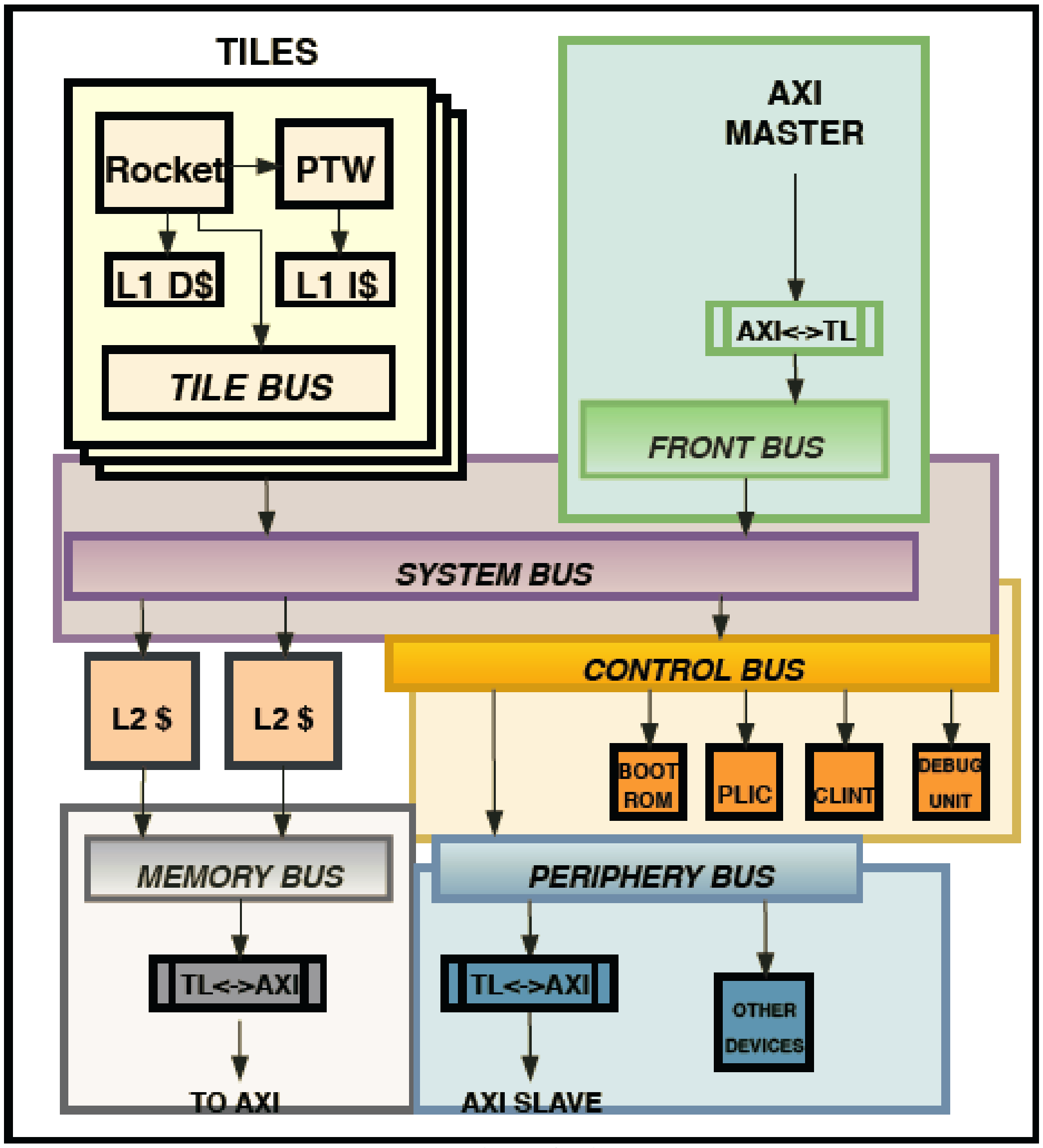


Figure 2.2: Rocket Chip SoC generator system

2.1.1 Rocket Chip generator

The Rocket Chip generator is a RISC-V SoC generator which was developed at University of California, Berkeley and supported by SiFive. Rocket Chip can be configured to have customizable inorder or out-of-order cores, heterogeneous or homogeneous core configurations, and additional extensibility through accelerators. Rocket Chip can also be extended at a software-level with ISA extensions to support co-processor functionality. The SoC generator system supports two types of system interconnect protocols, TileLink and AMBA. This generator system is tile-based and hence each core can be customized individually if needed. The Chipyard documentation covers about configuring the SoC based on custom needs in higher detail.

In the Figure 2.2, each block is developed as a Chisel RTL library which can be automatically imported and customized in our hardware designs. The tile includes the system core, optional L1 caches, page table walker and tile bus which is an internal interconnect. Rocket Chip system supports various configurable interfaces to leverage the SoC design experience. It has a system bus which connects the uncore to the tiles. Front bus acts as a master to the system bus and connects optional master devices to the system. There is a memory bus which provides the memory system interface. It connects the optional L2 banks or the broadcast hub to the main memory or DRAM. The system also supports control and periphery buses which are for memory mapped IO devices. Control bus was designed for devices internal to the Rocket Chip system. Mostly, SiFive developed core-complex devices like programmable interrupt controllers, debug units, bootrom, etc. go here. The periphery bus as the name suggest interfaces the external peripheral devices to the system.

2.2 Deep Learning and Neural Networks

Deep learning is a technique used by machine learning algorithms for computers to learn and predict various application demands such as image classification, action detection, and speech recognition. At a basic level, Deep learning imitates the behavior of neurons in human brain by stacking multiple layers, but with simpler single neuron dynamics. With this technique the machine learns to filter inputs at various layers and finally comes to a conclusion or output about the inputs. Each successive layer is inputted with the output of previous layer which as a whole processes the input and constructs a composite representation of it. Though various factors like activation, weights and transfer function[19] play a role, when considering a simple example of image prediction, the flow is the following: the input pixel matrix, which is provided to the first layer where the construction of pixel happens. The next layers filter for edges with the following layer beginning to represent certain low level shapes. This process continues by increasing the abstractions until a final output is produced.

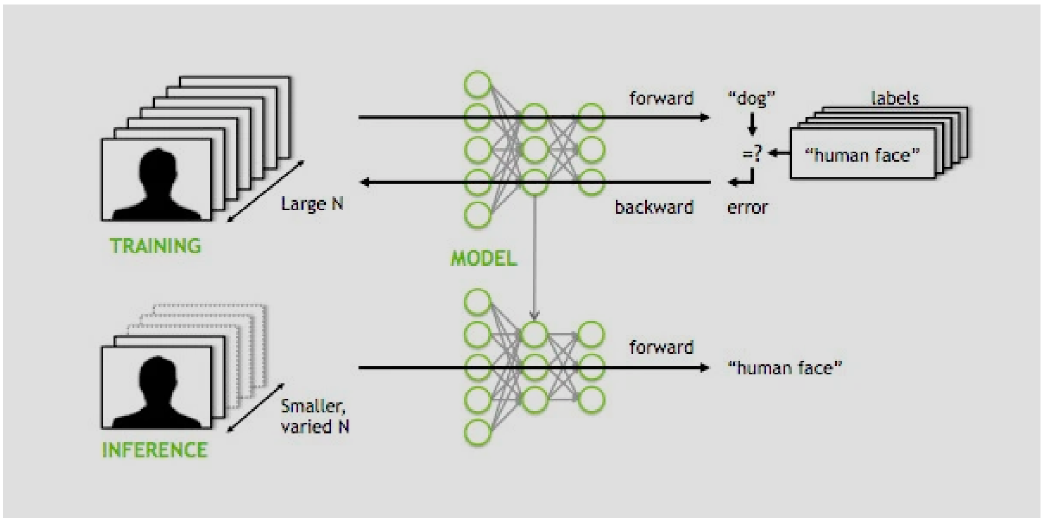


Figure 2.3: Deep learning Training and Inference phases

As shown in Figure2.3, deep learning happens in two phases, training and inference. As the names suggest, the network is trained to achieve a desired task by feeding in loads of data. It is mostly compute-intensive, and the weights are actually adjusted in this phase of learning. Training phase again occurs in two phases, forward-propagation and back-propagation. In forward-propagation, the input is loaded into the initial layer and it is propagated until an output is produced. The output is then compared with the expected output using a cost function and the performance is evaluated. The same process is repeated by back-propagation, where the data in feed back into the initial layer with the aim to reduce the cost function by changing the weights. This way until optimum weights are calculated the training continues and the model is ready for inference or testing. In the inference phase there is only forward propagation of data and the output produced is the final prediction result.

2.3 AWARE-DNN Accelerator

AWARE-DNN[20]is an architecture compiler framework developed in TeCSAR, at University of North Caroline at Charlotte. It is an application-specific hardware accelerator which is developed using a manually optimized architecture template. This framework is built using Chisel hardware construction language. The Chisel hardware construction language can leverage modularity and utilize libraries of Composite Blocks. The accelerator is built on the following function blocks:

- **Convolution Processing Engine:** It is the computation unit of AWARE DNN. It consist of two Sub modules:

  MAC-Engine: It handles computation and driven by 2D line buffer with extension to 1D line buffer [20] and accesses weigh according to the line Buffer's control signal.

  Tensor Buffer: It is Fused-Computational Unit and has an extension to 2D line Buffer. It can be thought of fused convolutional processing Unit. The layer's degree of channel parallelism indicates the number of tensor buffer required.

- **Aggregation Units:** It handles the aggregation of multiple CPE. Every CPE is constructed at the granularity of single channel. The results will be accumulated and then passes to the next processing element.

- **RELUs:** It handles the non-linear activation found in DNNs. It compares input to zero and outputs to max. This required for summation of every channel for individual convolution window.

- **Pooling Processing Units:** It is a final atomic function unit. It accelerates the max pool operation on streaming feature map.

- **Multi-Dimensional Aggregation Processing Engine (MDAPE):** The fusion of tensor buffer results in channel aggregation units to work at the granularity of single row instead of single pixel.

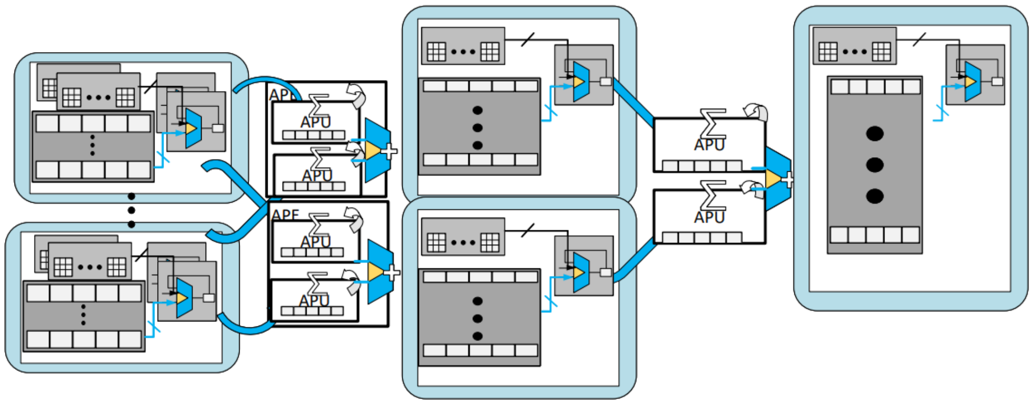


Figure 2.4: An abstraction of a full architecture Instance

Figure 2.4 shows an abstract view of a full architecture instance. Each layer in this architecture is mapped to an individual pipeline and each stage buffer sized for small section of FM tile. After enough data is accumulated, the current stage will begin feeding to the next stage in a producer consumer fashion. This is called as temporal layer parallelism (layer pipelining). This minimizes inter-layer data movement but puts memory hierarchy for weight storage. Due to which a special memory hierarchy is required to exploit the parallelism. The available parallelism shows which can be parallelism can be exploited to get the design needs are given below:

- **Convolution parallelism** : The convolution operation is done by single MAC unit. If the FM source is extended it becomes a sliding window operation. This creates data dependencies which must be maintained. AWARE DNN benefit from this data extension through buffer pipelining.

- **Kernel parallelism** : AWARE-DNN uses array of convolutional processing elements in repeated fashion to compute all the kernels of the particular layer simultaneously. This operation need weights and feature map.

- **Channel parallelism:** Due to the nature of DNNs, the channel parallelism of the layer is determined by kernel parallelism of the previous layer. The AWARE-DNN takes advantage of this in the form of channel buffering.

2.4 Related Work

The concept of specialized hardware accelerators being integrated to the processor was prominent from 1980 when Intel announced the release of Intel C8087 FPU[21]. However, it resumed interest around 2002 when the frequency scaling for smaller integrated chip sizes became an issue causing a stall in single-thread processor performance. From then on till today, special-purpose accelerators are still in use with massive demand. One of the early accelerator for graphics is Sony Emotion Engine[22], which was designed for gaming and graphics workloads. Then emerged GPGPUs[23] which were capable of high memory-intensive and graphics and non-graphics workloads. This flexibility increased the demand for these processing units to the most. FPGAs also made their entry in the early 2000s and showed massive performance increase compared to CPUs on some workloads.

Currently most studies on accelerators focuses on the performance and energy efficiency that accelerators can provide in comparison to general purpose processors [24, 25, 26]. But, interesting examples of the studies which consider the system-level implications are shown in [27, 28], which focuses on software level support for loosely-coupled accelerators. But this study limits by not considering a wider range of accelerator models and applications, while also neglecting the memory-hierarchy interference effects. Also this work makes effort using a coherent cache to attach accelerators to the PCIe bus and removing the need for device drivers. This is a could be a suitable approach for workloads which are accelerated off-chip due to area, off-chip communication needs or even to support reconfigurability (FPGA based accelerator). However, though current works on accelerators are mainly focused on optimizing design for performance and energy efficiency considering the accelerator standalone environment, this work[29] nicely showcases the need for co-designing the accelerator in system environment. In the work, the authors analyze the effects of on-chip communication and accelerator invocation overheads on different accelerators based on processor coupling and communication with memory. They present a quantitative comparison of high-throughput accelerator designs like Fast Fourier Transforms, AES, etc. following various coupling designs like tight coupling behind a CPU, loose out-of-core coupling with DMA to the Last Level Cache, and loose out-of-core coupling with DMA to DRAM and conclude that for workloads with non-trivial data sizes are best served by loosely-coupled accelerators with customized, private memory blocks.

One other interesting work which elevates the system-level effects on the accelerator is [30]. This work presents system-level analysis of the effects of cache-coherence models, accelerator-accelerator interference and processor-accelerator interference for loosely coupled accelerators. They use Embedded Scalable Platform for designing and programming the SoC with different configurations and run it on FPGA based platform. They conclude non-coherent memory model is the most effective approach for accelerators with large workloads. Also they conclude that that the accelerator speedup must be evaluated in a full-system context considering the interaction with all system components.

Though the above mentioned works are similar to the path of our work which is to provide a co-design environment for hardware accelerators and system cores, however, the previous work[29] is conducted using functional and cycle-accurate simulation environment[31, 32] to elevate the effects of system parameters on accelerator performance and not on providing a customizable processor accelerator co-design environment. Closely, in this [30] study, they used Embedded Scalable Platform for design and programming but framework but is not as extensible as RISC-V Chipyard agile framwork. In this thesis, we try to build a system interface wrapper for hardware accelerators in a well-built, extensible environment.

RISC-V Chipyard environment provides access to pre-built parameterizable system RTL blocks which are ready to use but lacks deep enough usage documentation for beginners to begin their integrations. However, because of the super-flexible full-system agile framework, there are many contributions which mainly utilize the open-source ISA extensibilty. Works like [33, 34, 35] are accelerators which are targeted for tightly-coupled integration design and hence work by ISA level extensions since they are either vector, systolic array based or encryption accelerators which require close interface with the processor design. One recent related work is FireSim-NVDLA[36], in which they integrated Nvidia Deep Learning Accelerator into FireSim[15]. FireSim is an FPGA-accelerated full-system simulator, which runs on the Amazon cloud FPGAs developed in the Berkeley Architecture Research Group at University of California at Berkeley. FireSim works on target design based cycle-simulation which is derived from the open-source RISC-V based Rocket Chip SoC [4]. In this integration, the NVDLA accelerator is decoupled from the DRAM controller of the host FPGA and to add a realistic memory model with last level cache and DRAM to the simulation environment for performance analysis. Also, through this integration they focused on analyzing the memory interference between the core and accelerator and performance benefits of having a shared last level cache.

CHAPTER 3: UNDERSTANDING ACCELERATORS COUPLING IN RISC-V ECOSYSTEM

RISC-V is an open-source development platform which has been gaining high-demand because of its open instruction set and the ability to customize a product that could be tailored to the needs of targeted workload. However, due the lack of proper documentation and the report of updates, keeping up with development environment is a huge challenge. This could possibly hinder the process of agile hardware development. Hence, this contribution is dedicated for the understanding and categorizing of different accelerator coupling types supported in RISC-V ecosystem.

The flow of this chapter is as follows, initially different system interfaces supported in Rocket Chip are overviewed. Then, detailed description of accelerator coupling types supported for Rocket Chip system is provided.

3.1 Rocket Chip interfaces overview

In the Figure 3.1, the interfaces inside the tiles, processor core to L1 caches is connected using the tile bus, which is the local bus. And the tiles are connected to the rest of the system via system bus, which is again interconnected to front bus, peripheral bus and control bus. Front bus, which masters the system bus is used for devices which would want to access the memory directly as a master without the processor core interference. Periphery and control buses are used for memory-mapped devices which are mastered by system bus. Optionally, these bus frequencies and widths can be configured. Rocket Chip SoC generator system libraries have flexibility to also integrate the AXI base devices through special converters. The system memory is connected to the on chip using the memory bus which is again connected to the system bus. The loosely-coupled accelerators integrated into this

SoC system have flexibility to configure the DMA to last level cache or DRAM if there is no last level cache in the design. Control bus is specified for memory-mapped devices internal to the Rocket Chip SoC system whereas Periphery bus is defined for external device integration into the SoC system as a peripheral or input-output device.

3.2 Reformalizing existing system concepts in RISC-V ecosystem

Accelerators are dedicated hardware units used to improve the performance of the targeted application. However, the choice of processor integration type has a fundamental effect on the accelerator design considerations and vice-versa. The accelerators are widely differentiated into tightly and loosely coupled based on their integration with processor. Below is the detailed description about them.

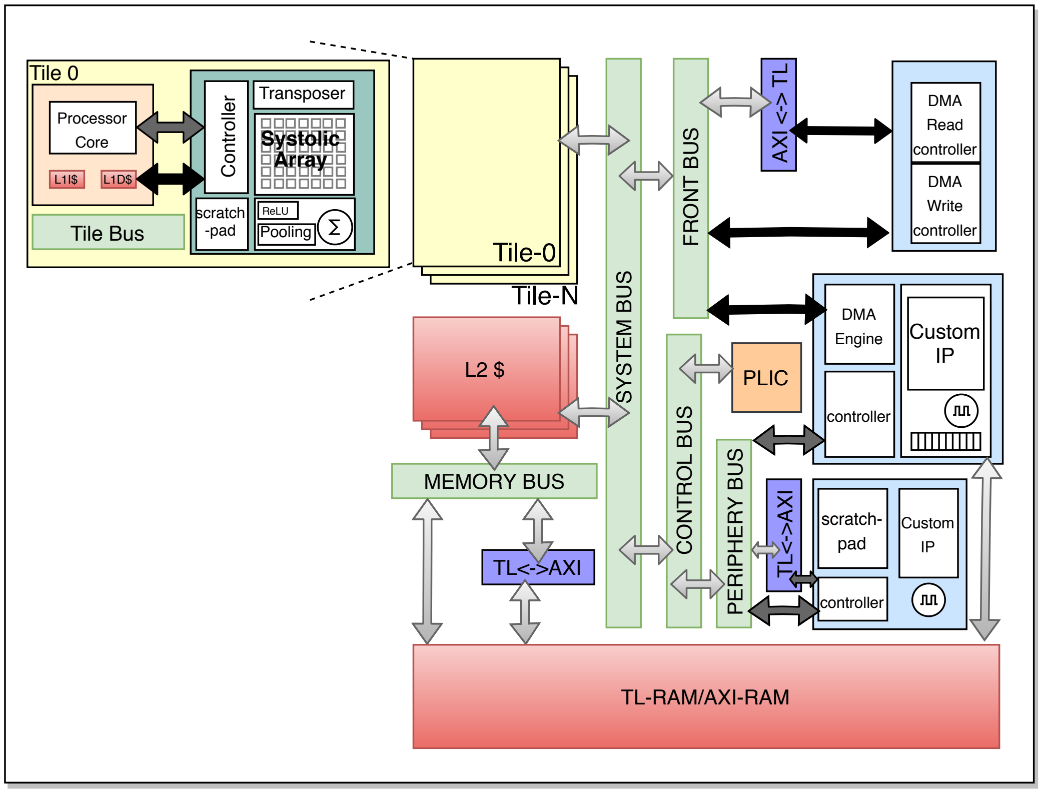


Figure 3.1: Rocket Chip accelerator coupling types

- **Tightly-coupled accelerators:** Tightly-coupled accelerators are closely controlled by the processor, meaning they are dedicated to the targeted processor are cannot be portable across various system cores. With the core directly controlling the accelerator, they have either low or no invocation overheads. However, they an be further classified based on the processor's local resources being shared like private caches, register file and pipeline.Also for these accelerators, the local memory is either transparent to the core or the accelerator shares the host memory and register file. Two major subdivisions in this are type of coupling are:

ISA Extension: This type of integration basically extends the processor pipeline with special functional units which share almost all the resources with the processors. Generally, these specialized units execute applications which follow single instruction multiple data like workloads. Some examples are vector and encryption based units which work using instructions which are not a part of host processor ISA. This type of coupling could basically limits the accelerator design because of the area and limited amount of storage elements. In Rocket Chip, these accelerator units are integrated inside the host processor core. The support for this type of accelerator extension in RISC-V ecosystem is easy because of its open ISA which could be tailored to the custom needs. However, for the scope of industry, this type of extension would not be preferable as the customized vendor specific devices have a closed ISA restrictions.

Co-processor: These accelerators optionally share some resources with the host processor but communicates using a dedicated interface. These accelerators perform complex tasks compared to ISA extended units and hence cannot be integrated on the processor pipeline. They use a special interface for communication and optionally for data transfer which is activated through special instructions. For data transfer, these accelerators can use a DMA or load-store bus transactions.

In context of Rocket Chip system, the co-processor is integrated inside the processor tile, shown at top left portion in Figure 3.1. This type of integration uses a special interface called Rocket Custom Co-processor (ROCC) interface. Further, this interface has two dedicated sub-interfaces, one for core communication and the other for memory communication. The ROCC ‘command’ interface is used to communicate and also request data from the accelerator. And the ‘mem’ interface for accessing the memory. However, the ROCC interface is provided with other sub-interface dedicated for data cache access. Each of these sub-interfaces follow ready-valid interface protocol internally which assure the hand-shaking mechanism while communication. This ready-valid interface is implemented using Chisel ‘Decoupled’ interface.

- **Loosely-coupled accelerators:** Loosely-coupled accelerators have their local memory ranges mapped in the system memory and they communicate with the core using these mapped registers. For applications with huge workload size, the processor might limit the accelerator’s achievable speed when integrated tightly. Hence, by decoupling the high-throughput accelerators from processor cores, the accelerator can be customized to include specialized multi-port datapaths, private-local memories and gain most of the speedup. These accelerators are configured using software device-drivers similar to a peripheral or device on system bus in a SoC. In Rocket Chip system, these accelerators are coupled as a slave device on system bus. For integrating as a device, they have a special bus which is mastered by system bus called the peripheral bus. All the external IPs or devices are attached to processor core via this bus interface.

  However, these accelerators differ based on the memory communications. Different types of these are described below:

  Memory copy based accelerators: Memory copy based accelerators transfer data by performing a memory to memory copies. This type of data transfer is similar to the memcpy() function provided by glibc. Basically, this function moves data using SIMD registers. However, in Rocket Chip system, there is support for memory copy co-processor which works in the similar way as load-store in vector or SIMD functional units. This co-processor has access to the CPU's page table walker and thus can perform its own virtual memory translations.

DMA based accelerators: For this type of coupling, data transfer can be done using DMA mechanism. Typically DMA interfaces can either access last level cache or DRAM. In this type of communication, CPU has to flushes all data from private caches and invalidate the region used to store the accelerator output data. Then it sends a DMA requests to the DMA engine and begins the transfer. Consequently, the accelerator starts execution either when it receives all the data or starts execution and streams in the data. After computing is done it outputs the data back to DRAM via DMA. The CPU can access the accelerator output in DRAM correctly as it invalidated that device memory region from its private caches. In the context of implementing this in Rocket Chip system, they can have the DMA interface nodes attached onto the front bus, which masters the system bus and provides direct access to the memory. In Figure 3.1, we show this type of coupling in context of isolated master device and also a slave peripheral with DMA engine for data communication.

CHAPTER 4: RISC-V LOOSELY-COUPLED ACCELERATOR WRAPPER (RISC-LCAW) DESIGN

This chapter provides a detailed design implementation of RISC-LCAW (RISC-V Loosely-Coupled Accelerator Wrapper). The flow of the chapter is as follows, first, the high level design of RISC-LCAW is over-viewed then the design of internal modules in the wrapper are explained.

4.1 RISC-LCAW Overview

The expanding RISC-V ecosystem provides support for various processor implementations like Rocket and BOOM and SoC generation frameworks like Rocket Chip and Embedded Scalable Platform. However, though accelerator interfaces or sockets are designed and are currently in use, to understand and utilize the benefits of such a huge framework on its own is painful and time taking. Ideally, having a defined accelerator wrapper for easy integration would reduced the time and manual effort for engineering the interface wrapper. In this thesis, we design a RISC-V Rocket Chip SoC system interface wrapper for loosely-coupled accelerators. We are limiting this work to target the data hungry, streaming accelerators which are currently in wide demand. In this section, we will be going through the design implementation details of the wrapper.

The Figure 4.1 shows the block diagram of the RISC-LCAW design. The main modules of the wrapper are Controller, DMA Engine and Input Buffer. Controller manages the accelerator interface and DMA and communicates to the processor core using the memory-mapped registers. The DMA Engine is responsible for reading and writing data to the memory (last-level cache or main memory) directly using the front bus. Input Buffer is an on-chip buffer used to store the data streamed by the DMA.

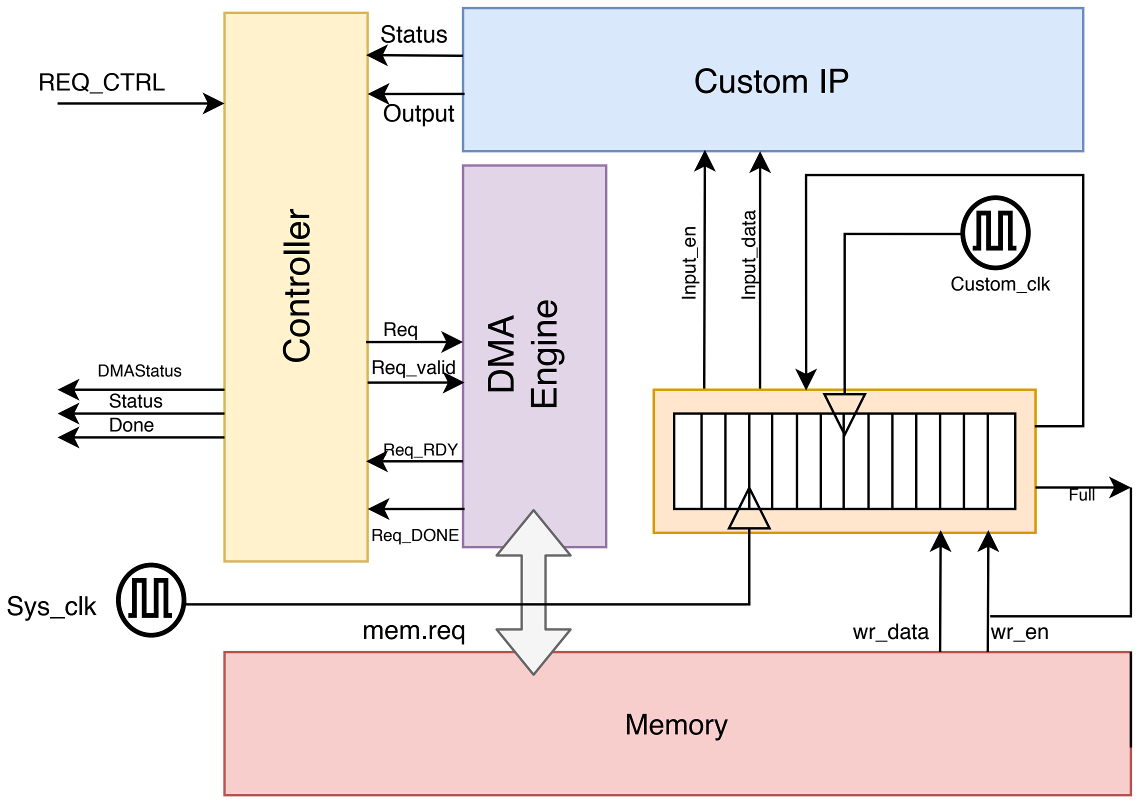


Figure 4.1: RISC-LCAW Design

Additionally, it is designed to support the clock domain crossing from system clock to accelerator clock. The accelerator eventually reads the data from the Input Buffer either in a streaming way, where the buffer size can be optimized but has to manage the data rate, or it can wait until the buffer gets full. In the next sub-sections, each module is described in detailed with the interfaces and design flow.

4.2 Controller Design

The wrapper contains a Controller module which manages the complete accelerator interface. The Figure 4.2 describes the controller flow. The module initiates the DMA transfer request to the DMA Engine when the processor core updates the DMAREQ register with the appropriate transaction start address, size and type. This module is also responsible to update the status of the accelerator to the processor core. When the DMA request type is read, it is serviced using the appropriate DMA

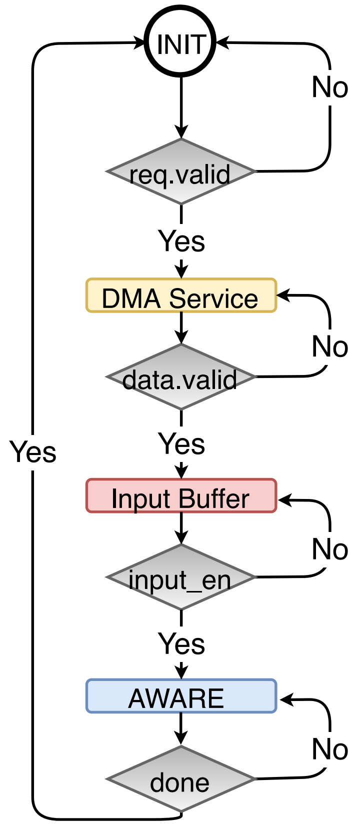


Figure 4.2: Controller Flow

Read Controller, and when the data.valid is high, the controller enables the Input Buffer write signal. When the input buffer is not empty, the accelerator input_en signal is configured to start reading the data either once all the data is loaded by the DMA into the buffer or can be customized to read streaming from the buffer. Additionally, this buffer implements an asynchronous queue which can help in crossing clock domains from the system clock to customized accelerator clock. However, once the accelerator finishes its computation, the output and done signals are outputted to the Controller which configures a DMA Write request by requesting the processor for bus grant. Once the processor updates the corresponding control registers, the Controller initiates the DMA Write to begin.

4.3 Direct Memory Access Interface Design

DMA as the name suggest is a feature of computer systems which facilitates direct memory access to certain hardware subsystems, without the involvement of the processor core. With DMA, the processor can continue with its operations without the need to wait for the transaction to complete. It has to just initiates the transfer and can continue with its execution and can update all the registers accordingly when it receives an interrupt from the DMA controller that the operation is done. This feature would be beneficial any time that the CPU cannot keep up with the data transfer rate, or when the CPU needs to overlap the computation and access while waiting for a slow data transfer. Without DMA, the CPU generally uses programmed input/output, where it is typically waiting for the entire duration of the data transfer operation, and is thus wasting all the cycles to perform other work. DMA transactions can be streaming or memory-to-memory. In the latter, the DMA is responsible for moving data within memory. DMA operates as a bus master, meaning it doesnot require the processor involvement for accesing system memory. However, correct measures must be taken to put the processor into a hold condition to avoid bus contention. There are three modes of DMA operations, burst mode, cycle stealing mode and transparent mode. We limit our design to support burst mode.

In our design, the main controller module intiates and enables the DMA request. We use Chisel HDL to design and program this module. We use Chisel's 'Decoupled' interface for maintaining the handshake protocal with different modules. Considering the DMA read operation, as seen in Figure 4.3, the interface has request valid (dmaReq.valid), transaction address (addr), transaction size (blockSize) which provide information about the DMA transaction and, request done ready (dmaReqDone.rdy) which is the ready signal from controller on finishing DMA request as inputs from the controller side to DMA Read controller. And, the request done (dmaReqDone) and data as outputs. From the memory side, the outputs from the

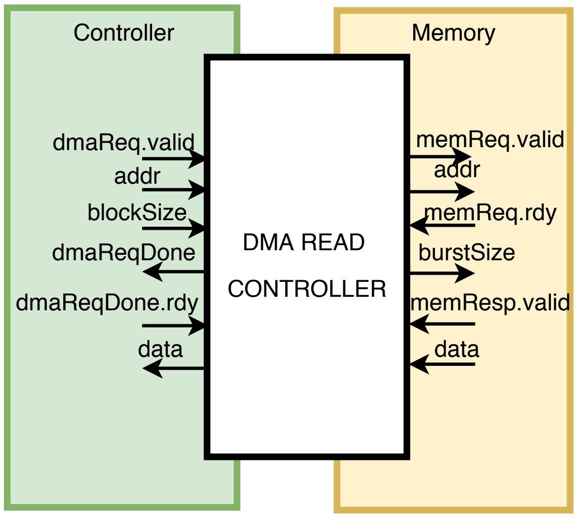


Figure 4.3: DMA Read Interfaces

Read controller are memory request valid (memReq.valid), transaction address (addr) and burst size (burstSize). The memory request ready (memReq.rdy), memory response valid (memResp.valid) and data are inputs to the DMA Read controller from the memory end.

In the similar way, the DMA write operation, as seen in Figure 4.4, the interface has request valid (dmaReq.valid), transaction address (addr), transaction size (blockSize) and data to be written into memory which provide information about the DMA transaction and, request done ready (dmaReqDone.rdy) which is the ready signal from controller on finishing DMA request as inputs from the controller side to DMA Write controller. And, the request done (dmaReqDone) and data as outputs. From the memory side, the outputs from the Read controller are memory request valid (memReq.valid), transaction address (addr) and burst size (burstSize). The memory request ready (memReq.rdy), memory response valid (memResp.valid) are inputs to the DMA Write controller from the memory end.

The flow of DMA Read controller is shown in the Figure 4.5. This design is managed using four states. First, 'INIT' state at which the DMA is idle waiting for a valid request. When dmaReq.valid is enabled, the state transitions to 'REQ' where, this controller sends a read request to memory. If the memory request is accepted, the

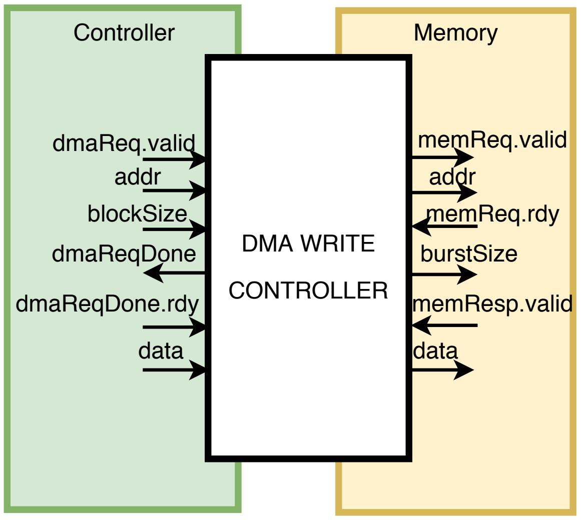


Figure 4.4: DMA Write Interface

controller transitions to ‘READ’ state. In this state, the data is read and if no more transaction bytes are left, the state is moved to ‘DONE’ state orelse, the state is moved back to ‘REQ’ state to request another burst of data from memory. The burst size, transaction size are configurable. But the burst size has to be in powers of two.

Similarly, the flow of DMA Write controller is shown in the Figure 4.6. This design is managed using four states. First, ‘INIT’ state at which the DMA is idle waiting for a valid request. When dmaReq.valid is enabled, the state transitions to ‘WRITE’ where, this controller sends a write request to memory. If the memory request is accepted, the controller transitions to ‘RESP’ state. In this state, the write done completion is received and if no more transaction bytes are left, the state is moved to ‘DONE’ state orelse, the state is moved back to ‘WRITE’ state to request write for another burst of data in memory. The burst size, transaction size are configurable. But the burst size has to be in powers of two.

4.4 Input Buffer Design

Input Buffer module is mainly designed to buffer the accelerator input data and enable support for safe clock domain crossing between the system clock and accelerator clock. As seen in the Figure 4.7, the Input Buffer acts as a barrier between the two

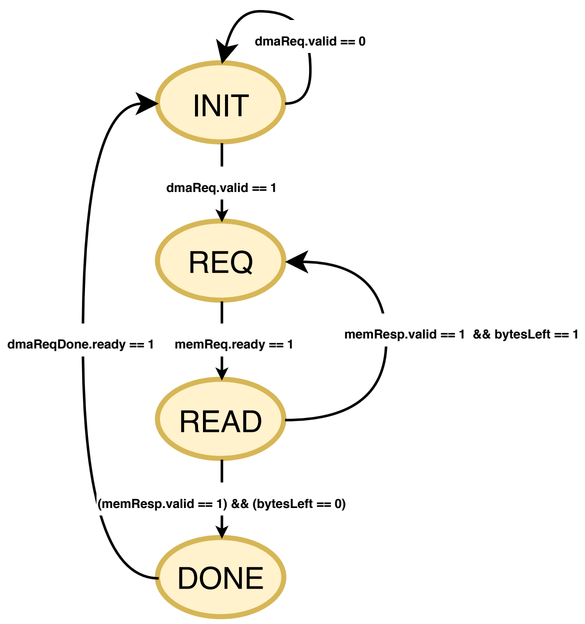


Figure 4.5: DMA Read Functional Flow

clock domains, write and read. The necessary signals from the write domain are write enable (wr_en), write data (wr_data) which is the data to cross the domain and the write clock (wr_clk) to which is the FIFO write operation synchronized. From the read domain, we have a read enable (rd_en) similar to write enable which is used to choose the right data. Then we have read clock (rd_clk) and read data (rd_data). We also have two very important control signals full and empty managed by write and read clock domains respectively. In general, clock domain crossing is a design optimization which gets complication in larger systems where the correct timing has to be maintained. In Chisel environment, due to the implicit clock, implementing and testing clock domain crossing is a challenge.

Clock domain refers to any sequential logic which is synchronous to a specific clock. When there is a need for data transfer between two different clock domains as shown in Figure 4.8, specific conditions have to be maintained. Often in larger systems, we tends to have different hardware subsystems running at different clocks for power and efficiency. In those cases, where there are multiple clock domains involved, the boundary logic for each clock domain has to be defined correctly to

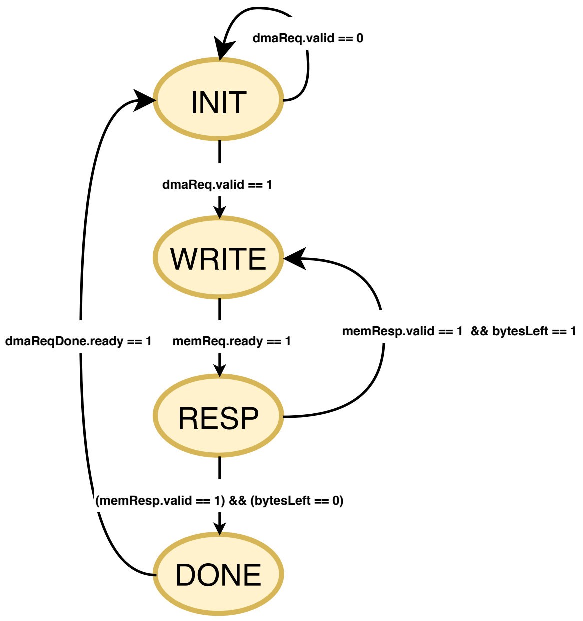


Figure 4.6: DMA Write Functional Flow

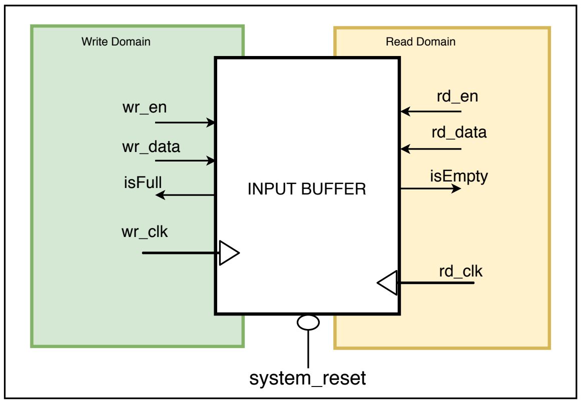


Figure 4.7: Input buffer interfaces for domain crossing

avoid the condition of meta-stability. Meta-stability is the condition in hardware circuits when the setup time and hold time of a particular signal are violated causing instability in the resulting output. This leads to data corruption or timing issues which would ultimately break the complete system logic making it difficult to debug and verify. Generally double-flopping technique is used to crossing from slow to fast, fast to slow in case of single data bit. But for complex cases, asynchronous FIFOs or queues are the best approach.

In the Figure 4.9, a detailed clock domain crossing implementation is presented. We have read and write controllers for controlling the read and write pointers and

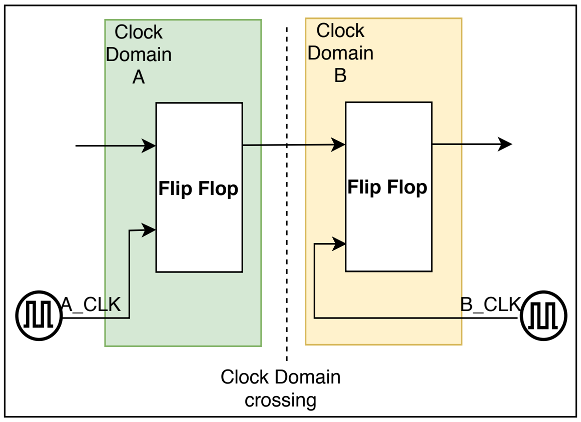


Figure 4.8: Clock Domain Crossing

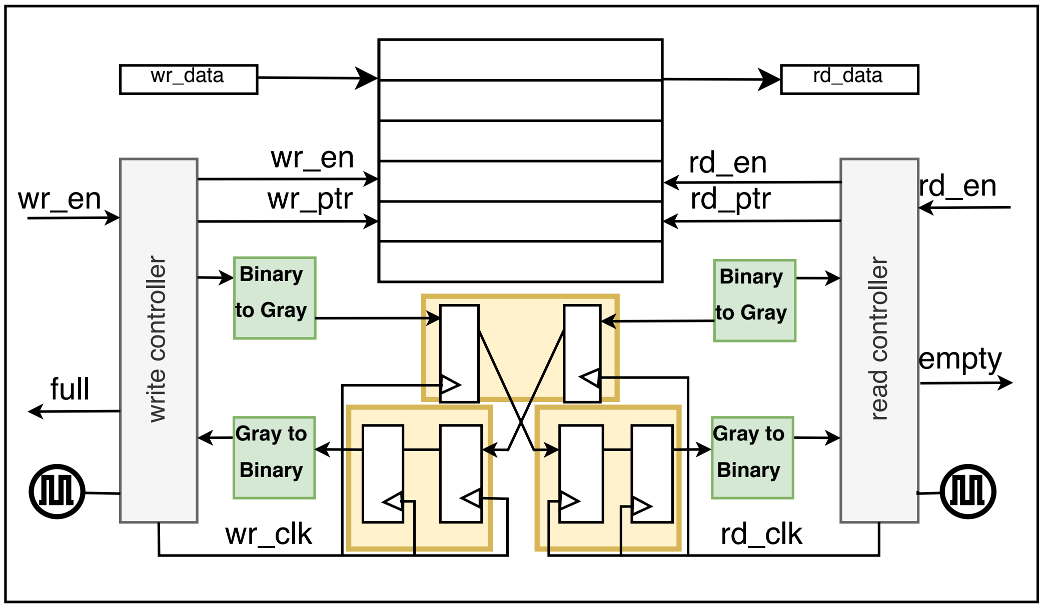


Figure 4.9: Clock Domain Crossing detailed implementation

data enables. These pointers are synchrnoized between the two domains using the double flop synchronizers. we also have gray to binary and binary to gray converters to avoid false full condition with binary pointers. Also, the full and empty condition signals are generated based on these pointers. In Chisel environment, due to the support of implicit clock, we need not supply a write clock as the system clock is the write clock which is supplied implicitly in Rocket Chip SoC system. Al the domain related signals have to be defined and updated in their respective domains. The FIFO has to be traced as a barrier by just using the pointers satisfying the correct boundary conditions, if full do not write to FIFO and if empty do not read from the FIFO.

CHAPTER 5: RISC-AWARE (RISC-V AWARE-DNN INTEGRATION)

In this chapter, we present one example of leveraging RISC-LCAW design for enabling a co-design system framework. AWARE-DNN is an application-specific hardware architecture compiler framework developed using Chisel HDL. It takes advantage of the modular design strategies inherent in Chisel and presents a parameterizable generator based hardware accelerator design. This accelerator is a data-flow streaming accelerator made to work near sensor, on the edge. By integrating it into RISC-V Chipyard, it can get the benefits of agile development design framework which also provides wrappers for various back-end design flows like software-based simulation, FPGA-accelerated system and VLSI flows.

Also, with the Rocket Chip support, many accelerator design optimizations can be explored which would allow it to work intelligently (for example adding support for leveraging multi-resolution neural network design) and possibly improve the design to support interface to multiple cameras. Additionally, this integration could enhance the application performance by having it optimized in system environment.

The Figure 5.1 presents the high-level integration block diagram for RISC-AWARE. This diagram can be explained efficiently by sectioning it as various steps involved in hardware-level integration with Rocket Chip. The following sections describe RISC-AWARE integration as a bottom-up approach or like a cake-pattern, starting with plugging accelerator into the RISC-LCAW template.

5.1 Customizing RISC-LCAW with AWARE-DNN

As discussed previously in Section 4.1 about the Figure 4.1, any custom device can be plugged into as a black box by port-mapping the accelerator device ports to the respective signals inside the wrapper template. Additionally, one important system-

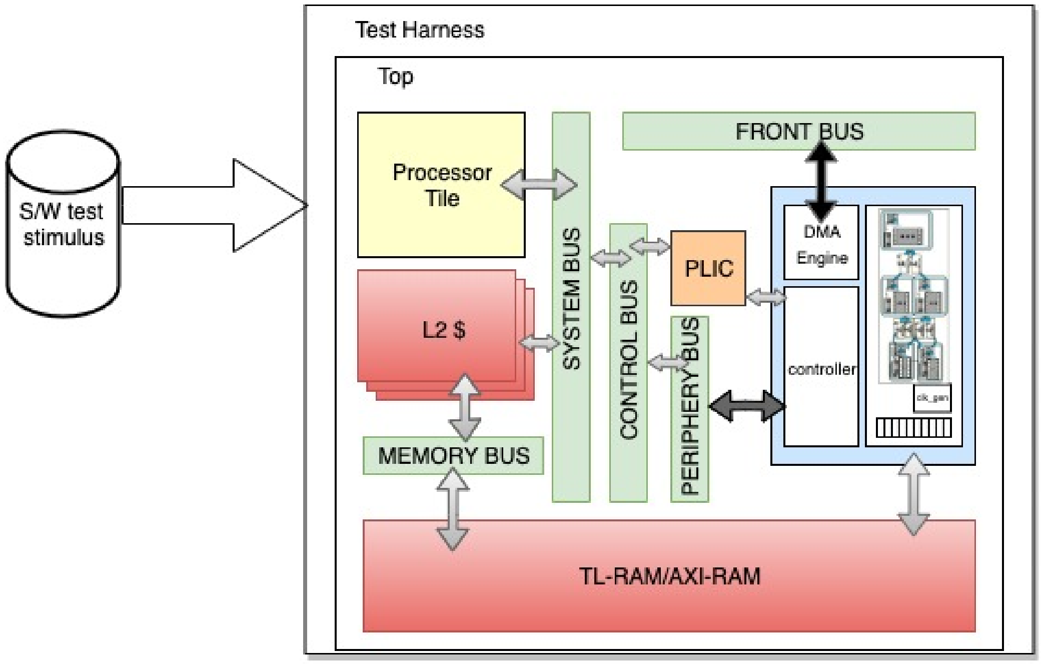


Figure 5.1: AWARE Integration

level customization we implemented for AWARE-DNN integration is to enable the support for a deterministic data access rate for

it needs to access the data at a specific frequency to maintain the flops per second between each layer. To implement this functionality, we used counters to begin the data transfer into the accelerator at a specific frequency. We could customize the DMA burst size and the targeted memory model for DMA (last-level cache or DRAM) to check the system-level parameterization effects on accelerator performance.

As an overall description about AWARE-DNN integration into the RISC-LCAW, the AWARE-DNN wrapper has two nodes for system interaction, one is the periphery bus node for communicating with the processor and the other is the front bus node, which allows AWARE-DNN to bus master for memory accesses. The functionalities of Controller, DMA Engine and Input Buffer are previously discussed in sub-sections 4.2, 4.3, 4.4 respectively.

We use the tabulated memory-mapped control and status registers for processor AWARE-DNN accelerator communication. We have a 64-bit DMAREQ register

Table 5.1: Control and Status memory-mapped registers.

CONTROL AND STATUS REGS	**SIZE (bits)**	**ADDRESS**
DMAREQ (Size, Address, Type)	64 (63:50,49:2,1:0)	0x2000
STATUS	2	0x2008
INTR/DONE	1	0x200C

which is managed by the processor. The processor initializes it with the appropriate data which includes the DMA request type, memory address and data size or transaction size. Also there is a 'STATUS' register which is updated by the accelerator controller. It is currently designed to shows four states, namely DMA Request, DMA Request done, accelerator startup and accelerator done. Additionally, it also has an interrupt register 'INTR' which is updated by the accelerator controller when the accelerator is done computing. And this interrupt the simulation.

5.2 Integrating AWARE-DNN Wrapper into Rocket Chip SoC system

As shown in Figure 5.1, the AWARE-DNN wrapper is integrated into the Rocket Chip SoC system via different bus interfaces. As mentioned in the previous section, there are two nodes for AWARE-DNN to communicate with the SoC, front node and periphery node. These nodes are interfaced to the respective buses. In Chipyard environment this can be done as follows using the Chisel based RTL generators in the Rocket Chip SoC system.

Listing 5.1: Plugging the wrapper to SoC

```
trait HasPeripheryAWARE {this: BaseSubsystem =>
  implicit val p: Parameters
  private val address = 0x2000 //base address of accelerator
  private val portName = "aware"

  val aware = LazyModule(new aware(ControlParams(address,
      pbus.beatBytes))(p))
  //hook aware controller to periphery bus
  pbus.toVariableWidthSlave(Some(portName)) { aware.control.node }
  //hook dma nodes (reader and writer) to front bus
  fbus.fromPort(Some("DMA"))() :=* aware.dmanode
}
```

Once the wrapper is completely designed, we need to attach it up to our SoC. Rocket Chip accomplishes this using the cake pattern. This basically involves placing code inside traits and plugging the accelerator nodes to the interface buses. The Chisel code for plugging the AWARE DNN to RISC-V Rocket Chip is shown in Listing 5.1. The nodes aware and DMA are hooked to periphery bus and front bus respectively. The base address and the port width have to included as parameters to the aware controller node.

After interfacing the accelerator to the SoC system, we need to create a top file with the bus interfaces for the accelerator to mix our traits into the system as a whole. Listing 5.2 shows the source code for creating a top file for AWARE DNN accelerator. Here we basically instantiate the accelerator and wrap it into a top module.

Listing 5.2: Adding the design into Top file

```
// SK: AWARE-Integration: Start: TopWithAWARE
class TopWithAWARE(implicit p: Parameters) extends Top
  with HasPeripheryAWARE {
  override lazy val module = new TopWithAWAREModule(this)
}
class TopWithAWAREModule(l: TopWithAWARE) extends TopModule(l)
  with HasPeripheryAWAREModuleImp
//SK: End: TopWithAWARE
```

After wrapping the accelerator into a top file, we need to create a configuration class for our accelerator in order to be able to include it in any system configurations. For that we need to extend the main ‘Config’ class and create our accelerator configuration as shown in Listing 5.3.

Listing 5.3: Creating AWARE DNN accelerator configuration

```
//SK: AWARE-Integration: Start Default: WithAWARETop
class WithAWARETop extends Config((site, here, up) => {
  case BuildTop => (clock: Clock, reset: Bool, p: Parameters) =>
          Module(LazyModule(new TopWithAWARE()(p)).module)
})
//End Default: WithAWARETop
```

Once we create the configuration class for our accelerator, we can customize our SoC design by using different Chisel RTL design libraries for different system units as shown in Listing 5.4. The complete SoC configuration considered for this study is tabulated in 6.1. However, with the parameterization support of Rocket Chip, we have the flexibility to customize our SoC to a high-end server system or resource-efficient edge system.

Listing 5.4: Configuring the SoC

```
//SK: AWARE-Integration: Start: AWARERocketConfig
class AWARERocketConfig extends Config(
  // use aware top
  new WithAWARETop ++
  // use default bootrom
  new WithBootROM ++
  // use Sifive L2 cache
  new freechips.rocketchip.subsystem.WithInclusiveCache ++
  // single rocket-core
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  // "base" rocketchip system
  new freechips.rocketchip.system.BaseConfig)
//End: AWARERocketConfig
```

CHAPTER 6: SYSTEM SIMULATION AND PERFORMANCE ANALYSIS

6.1 Simulation platform setup

For conducting our experiments we are using verilator software RTL simulator[18]. Verilator is an open-sourced LGPL-Licensed simulator maintained by Veripool. Chipyard framework has support to this simulator. It works by simulating the verilog RTL. For our simulation, the top level system design verilog file generated from the Chisel source code is simulated. The complete system under simulation is run at 500MHz which is managed by the verilator testbench file. The testbench file is by default designed in the Chipyard platform.

Table 6.1: System characteristics used for study.

System Parameter	**Details**
Processor Core	64bit- Rocket core (Inorder 5-stage)
Instruction Caches	16KB, 4-way, 64-sets, block size 64B
Data Cache	16kB, 4-way, 64-sets, block size 64B
Shared L2	512 kB, 8-way, 1024-sets, block size 64B
External Memory	256 MB, single-ported
Simulator Frequency	500MHz
Accelerator Frequency	125MHz

We use the SoC system setup tabulated in 6.1 as a baseline system for evaluating the performance differences with AWARE-DNN accelerator integrated into the system. We use a 64-bit, 5-stage inorder core, defaulted as Rocket core in Rocket Chip SoC generator system. The L1 instruction and data caches are both 4-way associate and 16-kilobytes sized. The cacheline size is by default 64-bytes. Additionally, we configured to include a L2 inclusive cache which is 8-way associative and 512-kilobytes sized. For the external memory support we use AXI-based DRAM of 256-megabytes size. The accelerator is configured to work at 125MHz.

Table 6.2: AWARE-DNN Configuration (parameters are defined for each layer

Parameter	11×11×**3**	32×32×**3**	64×64×**3**
RowSize	11,9,7,3	32,10,8,2	64,12,4,2
FilterSize	3,3,3,3	5,3,5,2	9,3,3,2
Stride	1,1,2,3	3,1,3,1	5,3,1,1
Out_size	9,7,3,1	10,8,2,1	12,4,2,1
KernNum	4,1,4,4	1,1,4,4	1,1,1,4
KernelPar	1,4,1,1	4,4,1,1	4,4,4,1
ChanPar	1,3,4,1	1,4,4,1	1,4,4,4
ChanBuffer	3,4,1,4	3,1,1,4	3,1,1,1
ConvPar	1,1,1,1	5,1,1,1	9,1,1,1
WaitTime	31,10,40,11	2,17,10,34	4,29,11,12

Along with the system simulation setup, we also configured the AWARE-DNN architecture compiler framework to generate accelerators with different network parameters. As tabulated in 6.2, we consider three different accelerator networks with 11×11×3 as baseline model, 64×64×3, modeled with Tiny-ImageNet configuration and 32×32×3, modeled with Mist network configuration for evaluating RISC-AWARE integration.

6.2 System Performance Analysis

In order to test the RISC-LCAW template and evaluate RISC-AWARE integration, we conducted tests with different image resolutions and compared the performance with the standalone accelerator performance. The main performance metric considered in this analysis is end-to-end latency which is the time taken to load the entire data set into the accelerator. Generally, this is equal to the latency required to load the first image as latencies for loading the remaining images is overlapped with computation.

As the baseline setup, accelerator network with image resolution 11×11×3 is considered and the DMA burst size is made to be cacheline size which is 64bytes. The parameters for accelerator setup are listed in 6.2 and for system setup, as mentioned in 6.1. For the initial evaluation, the baseline setup of system and accelerator is considered and as seen in Figure 6.1, the end-to-end latency for 11×11×3 resolution is increased by 1.2×.

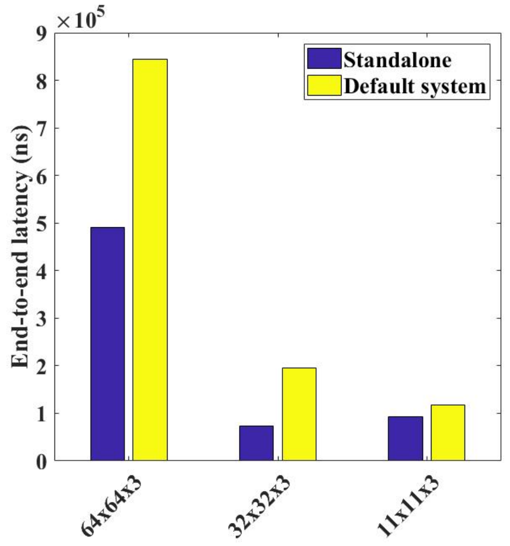
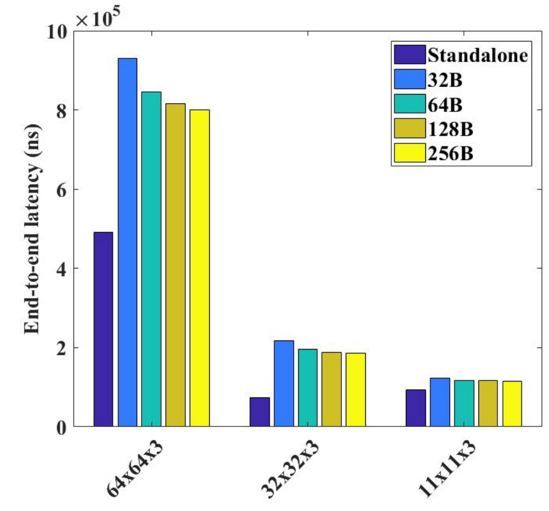


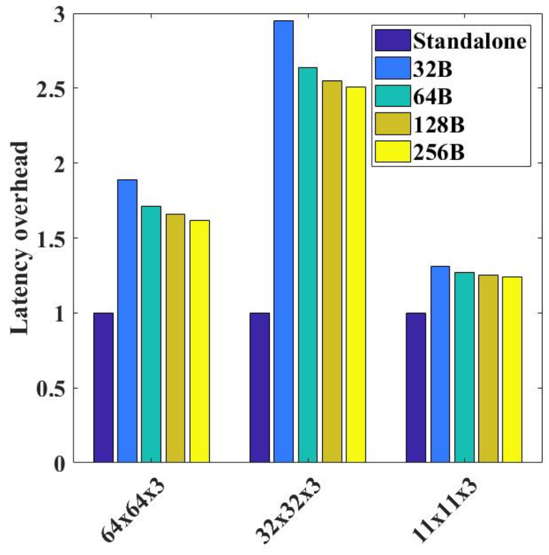
Figure 6.1: Accelerator end-to-end latency for different image resolutions in standalone and co-design environment

Then the test using baseline system configuration with varying accelerator networks with different image resolutions is conducted. The results show that with increase in data sizes and frames per second, we see an increase in accelerator end-to-end latency. For 64×64×3 resolution the accelerator end-to-end latency is increased by 1.7× and for 32×32×3 it is increased by 2.6×.

Additionally, tests are conducted with different burst sizes to explore the system-level effect on the accelerator. As presented in Figure 6.2a, the end-to-end latency seems to decrease with increasing DMA burst sizes. Furthermore, we present latency overhead for varying burst sizes and observe that for all networks, the system with 256bytes burst size performed better compared to the default burst size (64bytes).



(a) Comparison of accelerator end-to-end latency with varying burst sizes



(b) Latency overhead with varying burst sizes with respect to standalone accelerator

Figure 6.2: Comparison of accelerator end-to-end latencies for different image resolutions

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

This research provides a better understanding of different accelerator coupling approaches and interfaces by re-formalizing the existing system concepts in RISC-V ecosystem. Also a system-level integration diagram for Rocket Chip which shows possible types of accelerator coupling is presented. Additionally, we provide RISC-LCAW(RISC-V Loosely-Coupled Accelerator Wrapper) template as an accelerator socket for integrating loosely-coupled accelerator into Rocket Chip system. However, the scope of the wrapper support is limited for streaming, data-hungry accelerators. With this wrapper design we reduce the manual effort and engineering behind the processor accelerator integration for Rocket Chip. Along with the accelerator wrapper controller, the support for DMA and clock-crossing through the wrapper design is provided.

Furthermore, this work presents RISC-AWARE design, through which we provide system co-design platform and design-space exploration flexibility for AWARE-DNN accelerator. AWARE-DNN is a latency-aware, real-time configurable deep neural network architecture framework. We integrate this hardware accelerator to the Rocket Chip and elevate some system-level effects on the hardware accelerator performance. Experiments were conducted to compare the end-to-end latency of the accelerator with system and standalone. Considering the baseline setup (with image resolution $11\times11\times3$ and default system parameters), the results show that there is an increase in the end-to-end latency by $1.2\times$ the standalone accelerator latency. And for the networks with image size $64\times64\times3$, the accelerator end-to-end latency is increased by $1.7\times$ and for $32\times32\times3$ it is increased by $2.6\times$. Additionally, in this work we explore the effects of DMA burst size on RISCV-AWARE and see a decrease in the end-to-end latency when compare to the default burst size of 64 bytes.

7.1 Future Work

Currently, the RISC-LCAW is designed targeting streaming, data-hungry based architectures. As a future implementation, this work can be extended to include support for other loosely-coupled accelerators with private, high-bandwidth, multi-ported memories. Also, the co-designing platform currently supports single-core processor system, the wrapper framework could be extended to provide support for heterogeneous multi-core system platforms and various other possible system-level effects on accelerators can be explored. In addition, implementing full-system accelerator integration simulation can be explored which requires development of software-based device-drivers.

REFERENCES

[1] "Chipyard Documentation Guide." `https://readthedocs.org/projects/chipyard/downloads/pdf/dev/`, 2020. [Online].

[2] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, p. 48â60, Jan. 2019.

[3] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[4] M. Z. Alom, T. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. Nasrin, M. Hasan, B. Essen, A. Awwal, and V. Asari, "A state-of-the-art survey on deep learning theory and architectures," *Electronics*, vol. 8, p. 292, 03 2019.

[5] H. Yoo, "Mobile/embedded dnn and ai socs," in *2018 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, pp. 1–1, April 2018.

[6] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, "Gap-8: A risc-v soc for ai at the edge of the iot," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 1–4, July 2018.

[7] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the firrtl language," Tech. Rep. UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016.

[8] "Treadle: FIRRTL Execution Engine." `https://github.com/freechipsproject/treadle`. [Online].

[9] J. Bachrach, H. Vo, B. C. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing hardware in a scala embedded language," *DAC Design Automation Conference 2012*, pp. 1212–1221, 2012.

[10] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 209–216, Nov 2017.

[11] K. AsanoviÄ, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[12] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. AsanoviÄ, "Boom v2: an open-source out-of-order risc-v core," Tech. Rep. UCB/EECS-2017-157, EECS Department, University of California, Berkeley, Sep 2017.

[13] "Hammer Documentation Guide." `https://readthedocs.org/projects/hammer-vlsi/downloads/pdf/latest/`, 2020. [Online].

[14] "Firemarshal Documentation Guide." `https://firemarshal.readthedocs.io/en/latest/`, 2020. [Online].

[15] "Firesim Documentation Guide." `https://readthedocs.org/projects/firesim/downloads/pdf/latest/`, 2020. [Online].

[16] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 29–42, June 2018.

[17] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. NikoliÄ, R. H. Katz, J. Bachrach, and K. AsanoviÄ, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," *IEEE Micro*, vol. 39, pp. 56–65, May 2019.

[18] "Verilator Manual." `https://www.veripool.org/wiki/verilator/Manual-verilator`. [Online].

[19] "Whitepaper gpu-based deep learning inference : A performance and power analysis," 2015.

[20] J. Sanchez, N. Soltani, R. Chamarthi, A. Sawant, and H. Tabkhi, "A novel 1d-convolution accelerator for low-power real-time cnn processing on the edge," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–8, IEEE, 2018.

[21] J. F. Palmer, "The intel 8087 numeric data processor," in *Managing Requirements Knowledge, International Workshop on*, (Los Alamitos, CA, USA), p. 887, IEEE Computer Society, may 1980.

[22] M. Oka and M. Suzuoki, "Designing and programming the emotion engine," *IEEE Micro*, vol. 19, p. 20â28, Nov. 1999.

[23] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, "Gpgpu: General-purpose computation on graphics hardware," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC â06, (New York, NY, USA), p. 208âes, Association for Computing Machinery, 2006.

[24] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation cores: Reducing the energy of mature computations,” *SIGPLAN Not.*, vol. 45, p. 205â218, Mar. 2010.

[25] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” *SIGPLAN Not.*, vol. 49, p. 269â284, Feb. 2014.

[26] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, “Q100: The architecture and design of a database processing unit,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS â14, (New York, NY, USA), p. 255â268, Association for Computing Machinery, 2014.

[27] J. H. Kelm and S. S. Lumetta, “Hybridos: runtime support for reconfigurable accelerators,” in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pp. 212–221, 2008.

[28] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, “Architecture support for accelerator-rich cmps,” in *Proceedings of the 49th Annual Design Automation Conference*, DAC â12, (New York, NY, USA), p. 843â849, Association for Computing Machinery, 2012.

[29] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, “An analysis of accelerator coupling in heterogeneous architectures,” in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2015.

[30] D. Giri,, P. Mantovani, and L. P. Carloni, “Accelerators and coherence: An soc perspective,” *IEEE Micro*, vol. 38, pp. 36–45, Nov 2018.

[31] Y. Luo, Y. Li, X. Yuan, and R. Yin, “Qsim: Framework for cycle-accurate simulation on out-of-order processors based on qemu,” in *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pp. 1010–1015, Dec 2012.

[32] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “Dramsim2: A cycle accurate memory system simulator,” *IEEE Computer Architecture Letters*, vol. 10, pp. 16–19, Jan 2011.

[33] Y. Lee, C. Schmidt, A. Ou, A. Waterman, and K. AsanoviÄ, “The hwacha vector-fetch architecture manual, version 3.8.1,” Tech. Rep. UCB/EECS-2015-262, EECS Department, University of California, Berkeley, Dec 2015.

[34] C. Schmidt and A. Izraelevitz, “A fast parameterized sha3 accelerator,” Tech. Rep. UCB/EECS-2015-204, EECS Department, University of California, Berkeley, Oct 2015.

[35] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, Y. S. Shao, B. Nikolic, I. Stoica, and K. Asanovic, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," *ArXiv*, vol. abs/1911.09925, 2019.

[36] F. Farshchi, Q. Huang, and H. Yun, "Integrating nvidia deep learning accelerator (nvdla) with risc-v soc on firesim," *ArXiv*, vol. abs/1903.06495, 2019.