CONVOLVE

CONVOLVE

Seamless design of smart edge processors

GRANT AGREEMENT NUMBER: 101070374

Deliverable D6.1

Modular Architecture Template Definition

Title of the deliverable	Modular Architecture Template Definition
WP contributing to the deliverable	WP 6
Task contributing to the deliverable	Task 6.1
Dissemination level	PU – Public
Due submission date	30/04/2023
Actual submission date	30/04/2023
Author(s)	Gianna Paulin (ETHZ) Michael Rogenmoser (ETHZ) Tim Fischer (ETHZ) Vikram Jain (KUL) Guilherme Paim (KUL)
Internal reviewers	Bram Verhoef (AXE) Egbert Jaspers (VIN)

Document Version	Date	Change
V0.0	15/03/2023	Initial version with ToC and first content
V1.0	14/04/2023	The finalized draft ready to internal revision
V1.1	28/04/2023	Implemented feedback from reviewers and finalization

Table of Contents

Deliverable Summary

This document describes and defines high level SoC architecture and the interfaces for accelerators.

1. Objectives

This document "D6.1 Modular Architecture Template Definition" is a deliverable of the Work package No.6 "Compositional architecture DSE and SoC generation", task T6.1 "Modular architecture template definition" under the task lead of ETHZ and sets out the modular architecture template definition and defines the performance models required to design modelling and design space exploration frameworks.

1.1. WP6 Objectives

WP6 deals with automated compositional system architecture design space exploration (DSE) and system-on-chip (SoC) generation. This is done by providing a modular architecture template consisting of a RISC-V host with one or multiple machine learning (ML) and security accelerators.

The objectives of WP6 are defined as follows:

1) Provide a secure and modular RISC-V based SoC architecture template that eases the integration of multiple accelerators, managing control, synchronization, data exchange and run-time reconfiguration.
2) Create a SoC-level performance modelling framework for running ML applications on the targeted modular runtime configurable architectures, integrating the component models coming out of WP2.
3) Develop a rapid Design Space Exploration (DSE) framework to cycle quickly over ULP SoC and accelerator constellations, finding the optimal balance between design-time and run-time flexibility.
4) Realize an automated design time instantiation flow for optimal and run-time flexible SoC generation.

1.1.1. Deliverable D6.1 Objectives

The first deliverable D6.1 of WP6 focuses on specifying the modular SoC architecture template and defines a high-level SoC architecture and the interfaces for the accelerators designed in WP2/3.

1.2. WP6 Contribution to CONVOLVE's Objective

WP6 focuses on the modular SoC design and rapid deployment which makes the work package one of the contributors to achieve CONVOLVE's target to **reduce design time of edge AI hardware systems by 10x** by focusing on the faster design time of the SoC architecture and providing a design space exploration tool for rapid software-hardware co-design explorations. At the same time, WP6 is crucial to bring together all developed accelerators which are needed to achieve CONVOLVE's goal to **achieve 100x energy efficiency improvement** by providing an SoC template with standard interfaces to a set of ultra-low-power ML and security acceleration blocks which exploit novel architectures, microarchitectures, circuits and devices.

To achieve these goals, it is necessary to have customizable hardware acceleration blocks that can be parameterized during both design and run time using a standard interface. These blocks should allow for various configurations based on diverse application needs, including adjustments in supply voltage, clock frequency, data representation accuracy levels, parallelization degrees and dimensionality precision values. WP6 focuses on providing a modular and scalable SoC with such standardized interfaces such that the design acceleration blocks can be plugged easily to reduce the overall design time.

In addition to the RTL design itself, performance models and simulators must also be modifiable to enable fast exploration of the design-space without sacrificing compositional flexibility. WP6 focuses also on automated design-space exploration (DSE) and simulators using performance models of the hardware building blocks.

2. State of the Art and Related Work

This chapter gives a background of the projects which are used to specify and design the high-level SoC template. The template makes use of open-source RISC-V-based IPs from the PULP Platform project which is introduced in the following. Additionally, we give an overview of the state-of-the-art of heterogeneous SoC where general purpose compute cores are combined with highly specialized accelerators.

2.1. RISC-V

RISC-V[1] is an open-source instruction set architecture (ISA) that is gaining significant popularity in the embedded systems community. It is a RISC (Reduced Instruction Set Computing) architecture that has a simple, elegant design and is highly configurable, making it ideal for a wide range of applications.

One of the key advantages of RISC-V is its open-source nature, which allows anyone to contribute to the design and development of the ISA. This means that RISC-V processors can be customized with ISA extensions and optimized for specific applications, making them more efficient and cost-effective. Additionally, the open-source nature of RISC-V promotes innovation and collaboration among developers, which can lead to faster and more efficient development of new technologies.

RISC-V is designed to be vendor-neutral, which means that it can be implemented by any processor manufacturer without the need to pay royalties or license fees. This makes RISC-V an attractive option for companies that want to avoid the high licensing fees associated with proprietary ISAs.

RISC-V is also designed with modularity and extensibility in mind. The ISA is divided into standard modules which support a specific set of features such as atomic, single and double precision floating-point operation, which can be combined to create custom configurations. Overall, RISC-V's modular and extensible design enables a wide architectural and microarchitectural freedom such as VLIW (Very Long Instruction Word) or vector architectures. For example, RISC-V also specifies the RISC-V Vector Extension (RVV) which can be used to accelerate a wide range of HPC workloads, such as matrix multiplication, signal processing, and image processing. The RVV is highly configurable and can be customized to suit specific application needs. In addition, it has reserved encoding space for custom extensions. The deliberate design of the modular construct aims to facilitate easy extensibility, allowing developers to customize the inclusion or exclusion of features based on their specific requirements. This makes RISC-V processors highly adaptable and suitable for various application needs. All these properties of RISC-V make it the ideal candidate to design open-source, re-usable SoC architectures. The CONVOLVE initiative aims to achieve this objective by leveraging these properties.

[1] https://riscv.org/

2.2. PULP Platform

The PULP (Parallel Ultra Low Power) Project[2], a collaboration between ETH Zurich and University of Bologna, aims to produce open-source hardware and software based on RISC-V architecture that is scalable and energy efficient. The project has created multiple open-source RISC-V processor cores, peripherals, and other intellectual properties required for developing comprehensive System-on-Chips (SoCs). As part of the Convolve scheme, diverse collaborators will utilize several PULP IPs in constructing an innovative heterogeneous SoC featuring novel specialized accelerators.

The open-source project currently provides numerous relevant hardware and software elements, utilizing the open-source instruction set architecture (ISA) RISC-V extensively. The collection of intellectual properties includes several types of RISC-V cores ranging from those that are fully Linux-compatible to low-power microcontroller cores. As part of their research initiatives, they have integrated various extensions into the RISC-V ISA, such as `XpulpV1`, `XpulpV2`, and even ones tailored for neural networks called `XpulpNN` to decrease overall program cycle count while improving energy efficiency. These extended instructions comprise features like hardware loops, post-increment loads and stores as well as packed-SIMD dot-product operations amongst others. Their custom LLVM and GCC compilers support these extensions through built-in functions and partial automatic optimization processes.

PULP also has various open-source simulators to aid in developing software on these platforms. The C++-based GVSoC simulator [1], [2] supports running semi-cycle accurate simulations (cycle-count up until 90% accurate) of PULP-based systems at a much faster rate than typical RTL-level simulations. The (non-cycle-accurate but instruction-accurate) Rust-based Banshee simulator (publication) was developed to make quick software verification possible for scaled-up manycore systems.

For the CONVOLVE project, we make use of the extensive infrastructure developed within the PULP project and extend them with further IPs (e.g., a scalable interconnect).

[2] https://pulp-platform.org/

2.3. Heterogeneous System-on-Chips (SoCs)

Many recent industrial and research systems on chips for ML inference targeted towards (extreme) edge devices exist in the literature. In this survey, the performance and power information of the chips is gathered and plotted to evaluate and compare the existing state-of-the-art. The herein surveyed inference platforms are limited to the constraints imposed by energy-constrained devices on power (≤100 mW) and a circuit area smaller than 20mm^2. This review also limits to recent chips presented in and after 2019. The platforms are divided into four categories and shown in Table 1:

TABLE 1: SURVEYED HARDWARE PLATFORMS.

Platform Type	Surveyed Hardware
General Purpose	[3]–[5]
Digital Accelerators	[6]–[14]
Mixed Precision and In-Memory Computing (IMC)	[15]–[22]
Heterogeneous Multi-Core SoCs	[23]–[31]

General-purpose systems: These are commercial and research chips based on general-purpose CPU architecture and consist of more traditional cores like RISC-V and ARM with no specialized accelerators for AI. Some CPU cores are extended to support quantized matrix computations standard in ML workloads. Such systems have been traditionally used in IoT devices as they are flexible and easily programmable. Several commercial vendors provide microcontroller units used for ML in IoT applications. Specifically, all the boards that reported their performance for the MLPerf benchmarks have been included in this survey.

Digital accelerators: Specialized accelerators have been used extensively for ML workloads as they provide high performance and energy efficiency. This category consists of hardware blocks designed to accelerate a few ML layers efficiently, with extra support for precision scalability, sparsity, etc. They are based on the typical domain-specific hardware accelerator architecture [32] with decoupled computational units and memory.

Mixed precision and In-Memory Computing (IMC): The high data communication cost between the memory and cores is a significant bottleneck with the traditional decoupled compute and memory architectures. In most ML digital accelerators, the memory accesses energy dominates the overall consumption of the system. This has led to the design of new architectures based on near-memory computing and, more recently, IMC. The idea behind IMCs is to compute inside the memory bit cells, which avoids any data movement during the compute cycles. IMC is comparatively much more energy efficient and has gained a lot of interest in recent years. Two flavors of IMC, analog and digital IMC, have been proposed. Analog IMC provides very high energy efficiency but suffers from constraints in scaling to higher precision, lack of dataflow flexibility, and is prone to accuracy degradation due to noise. These constraints have led to the design of digital IMCs, which are memory blocks tightly integrated with digital MAC computation instead of analog.

Heterogeneous multi-core System-on-Chips: Heterogeneous systems are complete standalone system-on-chips that also provide several interfaces to connect to external world. They typically consist of a heterogeneous combination of accelerators and general-purpose cores. The trend of heterogeneous multi-core is driven by the premise that a single accelerator does not scale well and cannot provide flexibility and efficiency for mapping evolving ML models. Therefore, several energy-efficient simple cores can be connected in a system, and all network layers can be mapped on the most efficient core for the given layer. Several SotA heterogeneous SoCs from the convolve partners are included in the plot below. These include BrainTTA, DIANA, Kraken, TinyVers, and Vega.

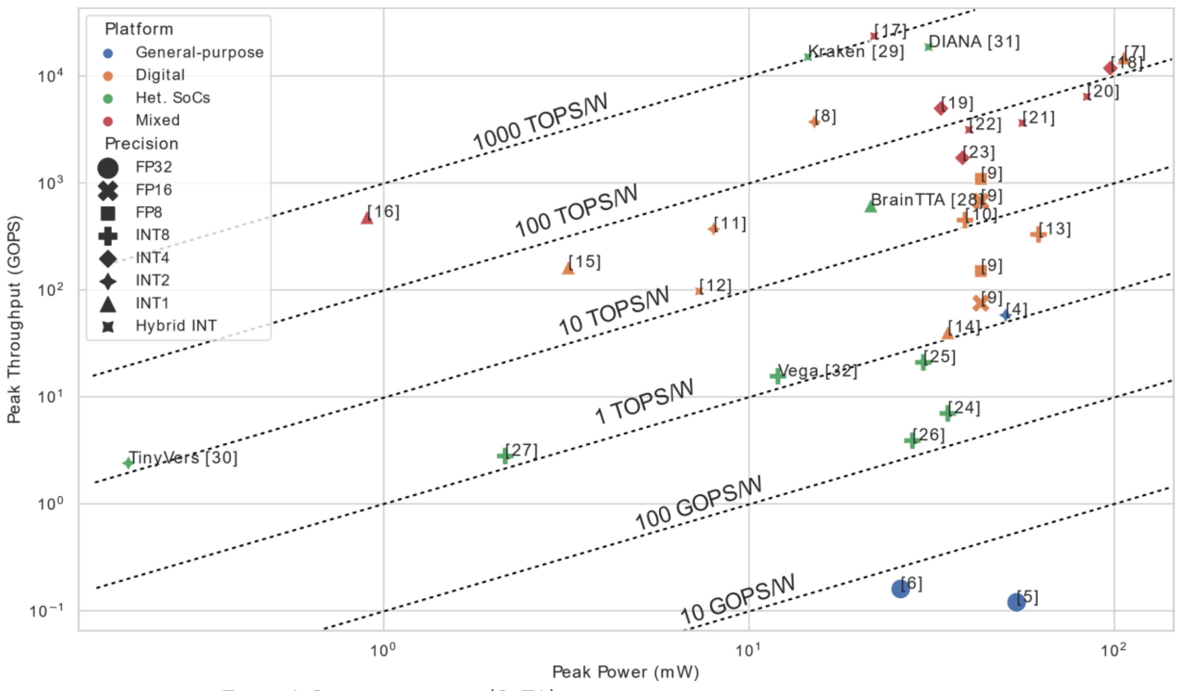


FIGURE 1: STATE-OF-THE-ART (SOTA) CHIP DESIGNS FOR EXTREME-EDGE COMPUTING.

Heterogeneous Chips from CONVOLVE Partners:

BrainTTA [28] is able to efficiently map various typical AI workloads, because of its inherent flexible datapath from the Transport-Triggered Architecture (TTA). The SoC consists of a RISC-V processor and a TTA-based accelerator. The accelerator is fully programmable and is supported by a C-compiler, which greatly simplifies mapping various AI (and other) workloads. BrainTTA, fabricated in 22nm FDX, has a peak energy efficiency of 29/15/2 TOPS/W (binary, ternary, and 8-bit precision) and a throughput of 614/307/77 GOPS.

Kraken, SoC with SNN and ANN accelerators: Kraken [29] is an example for an ultra-low-power heterogeneous SoC fabricated in 22 nm and combines a 32-bit RISC-V host core, 1 MiB of scratchpad L2 SRAM memory, and an autonomous I/O subsystem with three programmable, power-gateable accelerators: (1) A 1.8 TOp/s/W parallel general-purpose compute cluster with 8 RISC-V cores sharing 128 KiB of L1 scratchpad memory. The RISC-V cores support hardware loops, SIMD sub-byte dot-product integer operations with mixed-precision capabilities, MAC with concurrent data load (MAC-LD), and floating-point capabilities for energy-efficient digital signal processing. (2) 1.1 TSyOp/s/W accelerator called Sparse Neural Engine (SNE) targets spiking convolutional layers with 4-bit 3×3 filter and 8-bit leaky-integrate and fire (LIF) neuron states. (3) Completely Unrolled Ternary Inference Engine (CUTIE) is a 1036 TOp/s/W Ternary Neural Network (TNN) accelerator.

TinyVers - embedding MRAM: TinyVers [30] integrates a highly flexible-precision scalable digital accelerator, with a RISC-V core, a power management unit and an eMRAM, to provide a complete standalone edge-AI solution. The accelerator supports diverse AI layer types from Deep neural networks (NNs) (CNN, FC, TCN, GAN, AE) to traditional ML models like SVM at INT2/4/8 precisions. Fabricated in 22nm FDX, it provides 0.8-17 TOPS/W with power consumption ranging from 1.7 μW in deep sleep to sub-mW when running real AI workloads.

DIANA - mixed-signal, mixed-precision: DIANA [31] extends the idea of heterogeneity by combining an ULP analog in-memory core (AIMC) with a precision scalable digital NN accelerator, an optimized shared-memory subsystem, and a RISC-V host processor to achieve SotA end-to-end inference at the edge. The SoC achieves peak energy efficiencies of 600 TOPS/W (7bit I, ternary W, 6bit O) for the AIMC and 14 TOPS/W (8bit I/W/O) for the digital accelerator. When end-to-end ResNet20/CIFAR-10 and ResNet18/ImageNet classification workloads are mapped on the chip, 7 TOPS/W and 5.5 TOPS/W efficiencies are reported at system level respectively.

Insights and Trends

Based on the survey of SotA ML processors for the (extreme) edge, qualitative and quantitative analysis of the different key metrics, several insights and trends for future design can be extracted. The remaining section discusses these in detail.

Accelerator Dataflow: As data movement remains the primary bottleneck in ML workloads, finding the most optimal dataflow remains the biggest challenge. The selection of dataflow also dictates the efficient support of different ML layers on these hardware accelerators. A single dataflow cannot efficiently support the different types of layers; therefore, reconfigurable architectures such as DIANA and TinyVers that support multiple dataflows should be the ideal choice. However, careful trade-off analysis should be undertaken for reconfigurability vs. hardware overhead and power consumption. This requires a design space exploration for the co-design of hardware and software.

Arithmetic Precision: From the several performance plots, a strong correlation between precision and energy or power consumption can be observed. Therefore, finding the most optimal precision and training the models with a quantization-aware methodology to achieve good accuracy metrics remains a primary target. Support for variable precision computation can be beneficial to provide flexibility to support different workloads. Moreover, depending on the models, hybrid quantization, i.e., different precision of weights and activation might be more efficient; however, it can increase the complexity of the hardware.

Sparsity: Model compression through pruning can considerably improve performance. However, random sparse computation can be complex to handle in hardware but has the advantage of being relatively more accurate than structured sparsity. Therefore, a careful trade-off analysis between the accuracy of model deployment and hardware complexity should be undertaken. Structured sparsity can help with a known sparsity pattern making the hardware design easier. Though the accuracy of the models with structured sparsity needs to be evaluated, they can benefit from sparsity-aware training. Some surveyed platforms have used sparsity (unstructured and structured), showing improved performance and should be a future trend for hardware and software design.

Circuit Area: On-chip memory is the main driver for the overall chip area in most platforms. Large on-chip memory helps keep the model data local to the chip with reduced access (time and number) to external memory. This can decrease the system's overall energy consumption if the entire memory space is utilized. However, in most cases, large memory reduces the area efficiency as more storage does not always correspond to high bandwidth data movement, meaning that the throughput remains the same but the area increases. Analog and digital IMC alleviate this issue by having a coupled access and compute design, shown in several performance and area efficiency plots. For all hardware platforms, memory hierarchy design should be evaluated through an exploration framework to find an optimal design choice with maximum data reuse. Future designs could integrate efficient IMCs into SoCs to find a middle ground.

Analog vs. Digital IMC: Analog IMCs are significantly more area and energy efficient but suffer from analog noise and cannot scale to higher precision which affects the accuracy of models. Digital IMCs are more scalable and easier to train models for; thus, many recent works have focussed on these.

Heterogeneity: Heterogeneous SoCs tend to be more flexible and can cater to evolving ML applications. Efficient analog/digital accelerators and IMCs can be integrated into heterogeneous SoCs (e.g., DIANA) to take advantage of both, bringing flexibility with a moderate reduction in energy efficiency. Recent works have shown this is a major trend in research and academia. Multiple chips which combine ML-optimized RISC-V ISA extension cores with multiple specialized accelerators have been introduced. Thus, this trend is here to stay, and future designs should take advantage of this. However, challenges like compilers which can schedule workloads efficiently on the different resources, remain a bottleneck. Fast design exploration and full-stack hardware-software generation methodologies should be adopted to address these challenges of heterogeneous systems.

Power Management: Finally, power management brings the required system-level flexibility to save considerable power when mapping duty-cycled workloads. Power management should also be used to fully exploit dark silicon in heterogeneous systems where parts of the chip are not utilized for specific computational workloads. Future IoT devices must combine heterogeneous systems with power management to build genuinely efficient and flexible systems.

3. Definition of high-level SoC Template and Accelerator Interface

In this chapter the high-level SoC template and the different accelerator integration levels are described. For each integration level we define the supported interfaces for the *control* and for the *data plane*. The described SoC template is inspired by the open-source IP landscape of the PULP Platform project [26,29]. CONVOLVE partners are free to re-use the open-source IPs[3] to design their accelerators.

3.1. SoC Template Overview

Figure 2 gives a high-level overview of the SoC template. The SoC template is split into a support infrastructure domain which comes with a RISC-V host, a main memory, and some peripherals. This domain is attached over a high-speed on-chip interconnect (e.g., network-on-chip (NoC), AMBA AXI) to a set of L2-accelerators. These L2-accelerators can be the same type of accelerator or a combination of different accelerators. An example L2-accelerator is provided with the compute cluster (shown on the right side of Figure 2). Inside this Accelerator a set of general-purpose RISC-V cores can be extended with L0-accelerators or can share the tightly coupled data memory (TCDM) with a set of L1-accelerators. Each L2 and L1 accelerator run at the same clock frequency as the SoC and/or cluster, but can be independently clock gated.

To summarize, we define three different levels of accelerator integration in the SoC template:

- **L0:** RISC-V co-processor
- **L1:** Tightly coupled accelerator
- **L2:** Loosely coupled AXI- accelerator

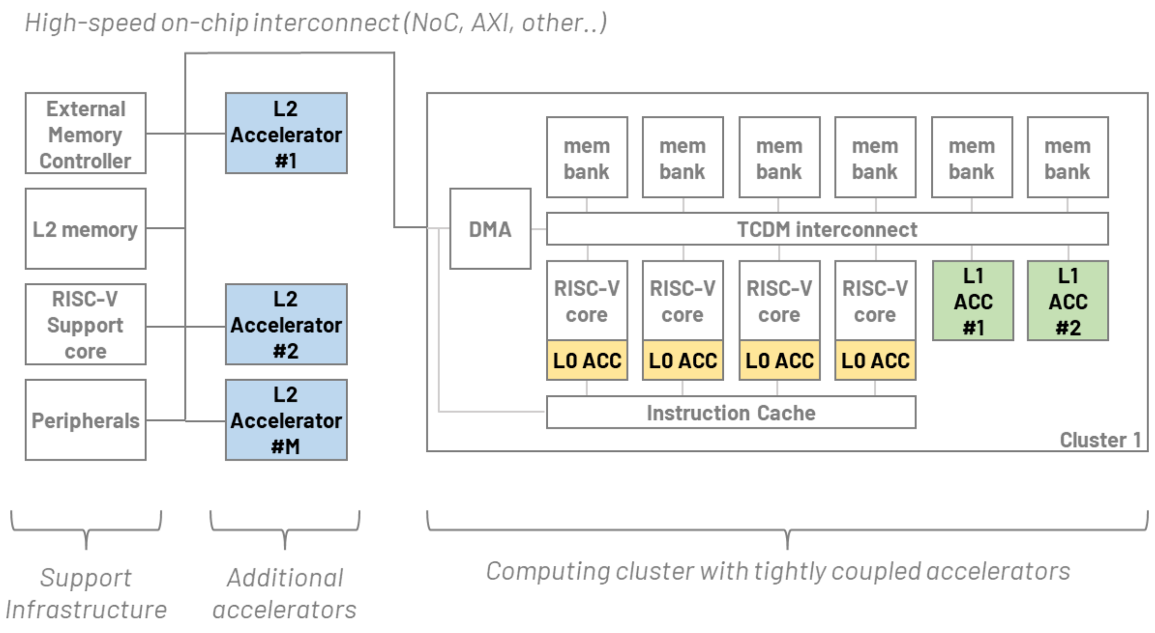


FIGURE 2: OVERVIEW OF THE HIGH-LEVEL SOC TEMPLATE WITH THE THREE MAIN ACCELERATOR TYPES.

[3] https://github.com/pulp-platform/

3.1.1. Support Infrastructure / Host Domain

The support infrastructure is the main frame for the SoC, providing general infrastructure for the functionality of and interfacing with the SoC. As shown on the left side of Figure 2, the support infrastructure includes a RISC-V support core, L2 main memory, peripherals, and an external memory controller.

The general-purpose RISC-V support core, also termed fabric controller (FC) or host processor, provides the necessary functionality to control the overall system. The core, in previous PULP systems based on a lightweight RISC-V core, acts as the primary controller for the system's peripherals and is responsible for managing the memory of the system, both external and the on-chip L2 main memory. As the main control core of the system, it helps coordinate the data processing operations between the various components of the system, namely the computing cluster, and accelerators.

The L2 memory offers a scratchpad memory for the system. It is designed to contain the program code for the host processor and limited data to be transferred to the cluster or accelerators. Designed to be fast, it offers limited space to temporarily buffer data between large external memory and data within the cluster or accelerators.

To interface with the outside world, the support infrastructure also contains peripherals, such as a JTAG for debugging. Other peripherals, such as SPI, UART, and GPIOs, are also available as memory-mapped devices. These peripherals allow the SoC to communicate with external sensors and connected devices.

Finally, an external memory controller is included, allowing the system to interface with a large, off-chip memory. This external memory controller handles all interfacing with the external device and offers on-chip devices access to the data.

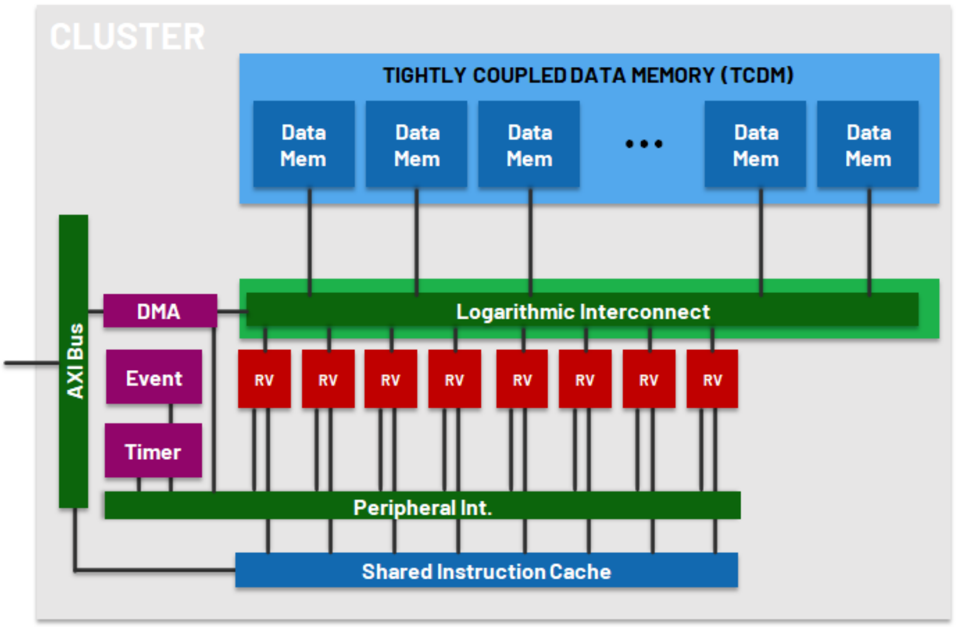


FIGURE 3: BASE COMPUTE CLUSTER.

3.1.2. Compute Cluster

A high-level overview of the compute cluster is shown on the right of Figure 2. The cluster consists of a parameterizable number *P* of general-purpose processing elements (PE) based on RISC-V cores and is connected to the host domain over a AXI4 port (shared for *control* and *data plane*, see Section 3.5.1). The base cluster can, therefore, be understood as a special type of L2 accelerator and is shown in Figure 3. It can be further extended with L0 or L1 accelerators on multiple levels as will be explained in Section 3.1.3. The compute cluster has a hierarchical instruction cache for the PEs that consists of private L0 and shared L1 instruction caches. All PEs and L1 accelerators share access to the tightly coupled data memory (TCDM). The TCDM is a multi-banked (typically a banking factor of 2 which results in 2x*P* banks) scratchpad memory that is explicitly managed by software and is the main way for the PEs and L1 accelerators to access data with a single cycle latency. The DMA module within the cluster can be programmed by the PEs to transfer data between the larger L2 main memory in the host domain and the compute cluster. An event unit provides the means for synchronization between the PEs.

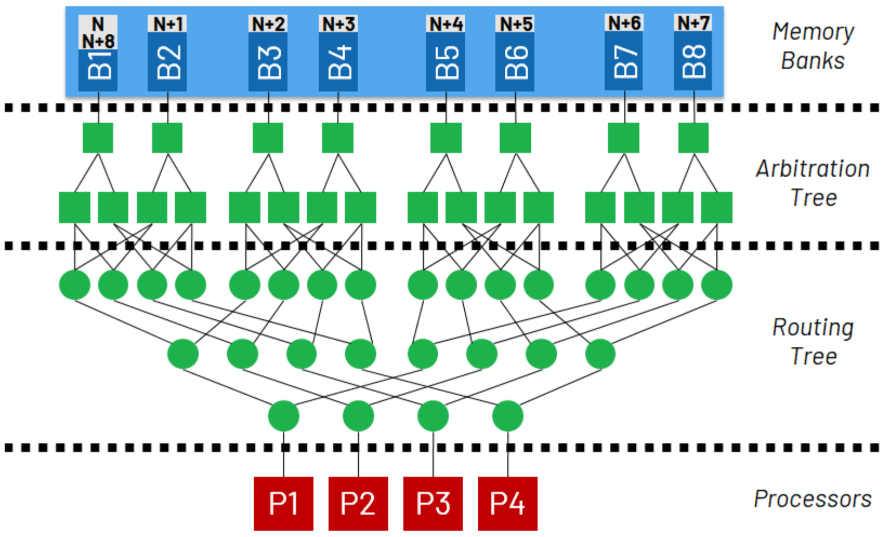


FIGURE 4: LOGARITHMIC INTERCONNECT.

Each PE is connected via two interconnects: The peripheral interconnect is a low bandwidth bus based on PERIPH protocol or the Register protocol (see Section 3.4.3) used for the configuration of the DMA, event unit, and, most importantly, the L1 accelerator's *control plane*. The TCDM interconnect provides low latency and high-bandwidth access to the shared TCDM memory and is based on the so-called Logarithmic Interconnect, a forest of arbitration trees providing parallel single-cycle access to the multi-banked memory with transparent arbitration in the case of bank conflicts between different processing elements. The TCDM interconnect is shown in Figure 4 and follows the TCDM protocol (see Section 3.4.2). The complexity in terms of area and timing of the Logarithmic Interconnect can explode if to many PEs and/or L1 accelerator ports get connected (Figure 5). Therefore, if more bandwidth is required, the cluster needs to use the Heterogeneous Cluster Interconnect (HCI) and shown in Figure 6. In that case the L1 accelerator should use the HCI protocol (see Section 3.4.2).

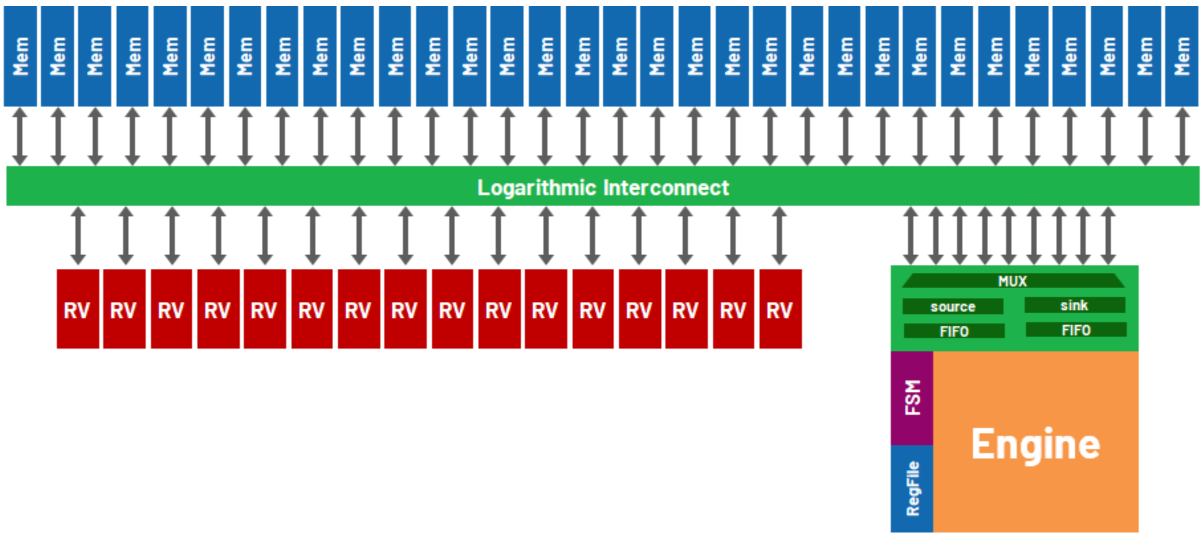


FIGURE 5: L1 ACCELERATOR INTEGRATION WITH LOGARITHMIC INTERCONNECT.

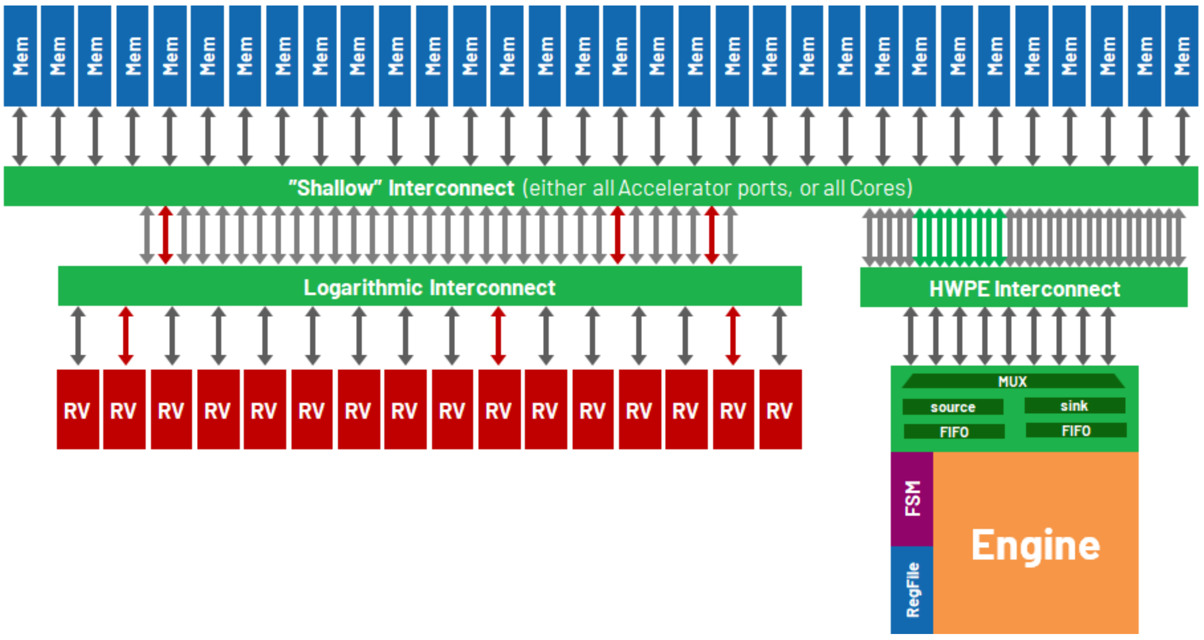


FIGURE 6: L1 ACCELERATOR INTEGRATION WITH HETEROGENEOUS CLUSTER INTERCONNECT (HCI).

3.1.3. Accelerator Integration Levels

We define three different levels of integration which can be used to integrate the accelerators:

- **L0:** a RISC-V co-processor
  - *control plane:* Core-V-X, RISC-V ISA extensions
  - *data plane:* Core-V-X, tightly coupled to the pipeline stages of the RISC-V processor, uses the RISC-V core's load-store unit (LSU)
- **L1:** Tightly coupled accelerator
  - *control plane:* PERIPH or Register protocol, memory-mapped register file to configurable control sequencer
  - *data plane:* TCDM or HCI protocol, independent of the RISC-V core, uses and controls its own streamers/DMAs
- **L2:** Loosely coupled AXI4- accelerator
  - *control plane:* AXI4
  - *data plane:* AXI4

TABLE 2: OVERVIEW OF THE SUPPORTED INTERFACES FOR EACH ACCELERATOR INTEGRATION LEVEL.

Integration Level	Plane	Supported Protocols (only one per plane)
L0	*Control & Data*	Core-V-X (Section 3.3.1)
L1	*Control*	Peripheral Interface, Register Interface (Section 3.4.3)
	Data	TCDM or HCI interface (Section 3.4.2)
L2	*Control & Data*	AXI4 (Section 3.5.1)

Table 2 gives a summary of the supported protocols for the *control* and the *data plane* for each accelerator integration level. The protocols are described in next sections.

In Section 3.2 we specify some specifications which all accelerators need to follow. Depending on the chosen integration level, the accelerators need to implement one of the interfaces instructions from Section 3.3 for L0 integration, Section 3.4 for L1 integration, and Section 3.5 for the L2 integration.

3.2. General Specifications for all Accelerators

To keep the accelerators aligned and easy to integrate, **every accelerator has to follow the specification described in the following subsections.**

3.2.1. General

All signal directions are stated from the perspective of the Accelerator. Therefore, input-signals are signals driven by the interface logic around the Accelerator while output-signals are driven by the Accelerator. **If not otherwise indicated, all signals are active-high.**

3.2.2. Clock & Reset

Table 3 gives an overview of all interface signals which each accelerator needs to support. All the interface signals of the accelerator are synchronized to the rising edge of the reference clock `REF_CLK`.

TABLE 3: CLOCK AND RESET SIGNALS.

Signal	Width [bits]	Direction	Description
`REF_CLK`	1	*input*	The reference clock with which the interface is synchronized.
`RSTN`	1	*input*	Asynchronous active-low reset.
`SOFT_CLEAR`	1	*input*	Synchronous active-low soft reset.

The asynchronous reset signal **`RSTN`** is asserted during system power-up and completely resets the accelerator. The synchronous soft clear signal **`SOFT_CLEAR`** on the other hand is used by the interface logic to reset the Accelerator Block to a known good state or when a Soft Clear was requested by a PE. A soft clear does not necessarily re-initialize the content of the non-volatile memory or long-lived configuration but must unconditionally and immediately put the accelerator into a known good idle state where it is ready to accept a new instruction.

3.3. L0 – Accelerator: RISC-V Co-processor

L0-accelerators are tightly-coupled Co-Processors (e.g. an FPU) to the RISC-V PEs which can implement custom ISA extensions to accelerate specific workloads. The PEs can offload instructions that it does not itself is able to process to the L0-accelerators. An overview of the L0-accelerator is shown Figure 7.

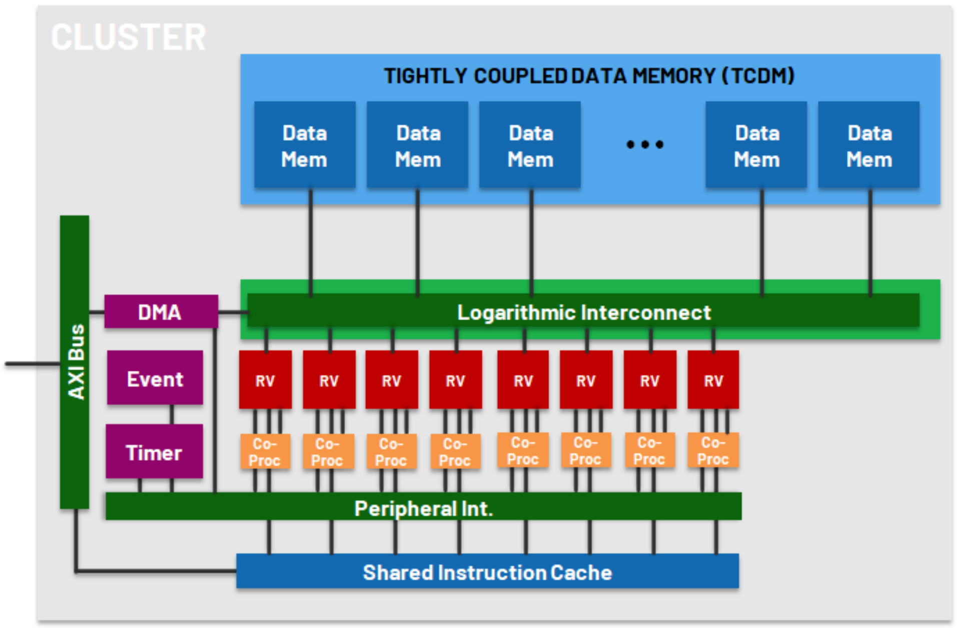


FIGURE 7: BASE COMPUTE CLUSTER WITH L0 INTEGRATED RISC-V CO-PROCESSOR.

3.3.1. Control and Data plane: Core-V-X

L0-Accelerators (co-processors) can be attached to the RISC-V PEs via the Core-V-X interface. The Core-V-X enables extending CPU with (custom or standardized) instructions without the need to change the RTL of CPU itself. Extensions can be provided in separate modules external to CPU and are integrated at system level by connecting them to the Core-V-X. The Core-V-X provides low latency (tightly integrated) read and write access to the CPU register file. Opcodes which are not used (i.e., considered to be invalid) by CPU can be used for extensions. The official documentation can be found in the link[4].

The Core-V-X interface consist of six different interfaces which are described in Table 4. The exact signals are omitted, but are described in more detail in the documentation.

[4] https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/

TABLE 4: CORE-V-X INTERFACE.

Interface	Signal from CPU	Signals to CPU	Description
Compressed	`valid`, `req`	`ready`, `resp`	Compressed instruction to be offloaded.
Issue	`valid`, `req`	`ready`, `resp`	Uncompressed instruction to be offloaded including its register file based operands.
Commit	`valid`, `commit`	-	Signaling of control signals related to whether instructions can be committed or should be killed.
Memory req/resp	`valid`, `req`	`ready`, `resp`	Signaling of load/store related signals (i.e. its transaction request signals). This interface is optional.
Memory result	`valid`, `result`	-	Signaling of load/store related signals (i.e. its transaction result signals). This interface is optional.
Result	`ready`	`valid`, `result`	Signaling of the instruction result(s)

3.3.2. Programming Model

L0-accelerators can be programmed by extending the RISC-V ISA with custom ISA extensions. The RISC-V automatically offloads instructions that it cannot process over the Core-X-V interface to the L0-accelerators. To this end, compiler support must be added for the custom ISA extensions. Various PULP ISA extensions have already been added to the LLVM compiler[5] and GCC[6].

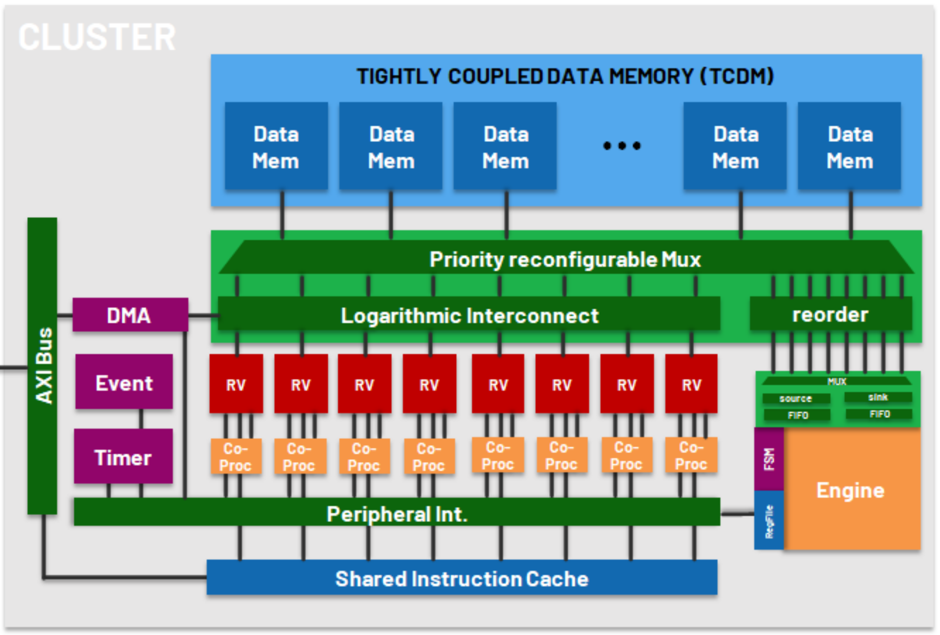


FIGURE 8: BASE COMPUTE CLUSTER WITH L0 AND L1 INTEGRATED ACCELERATORS USING THE HETEROGENEOUS CLUSTER INTERCONNECT (HCI).

[5] https://github.com/pulp-platform/llvm-project
[6] https://github.com/pulp-platform/riscv-gcc

3.4. L1 – Accelerator: Tightly-Coupled Accelerator

The PULP project has designed already various accelerators which correspond to the L1-level integration used in the CONVOLVE project and are called Hardware Processing Engines (HWPEs)[7]. The project has released some of the accelerators a set of IPs open-source on Github[8].

CONVOLVE partners are free to re-use these IPs to design and integrated their own accelerators into a PULP cluster as described in Section 3.1.3.

3.4.1. HWPE Overview

The HWPE are specialized accelerators that work in conjunction with a PULP system to enhance its performance and energy efficiency for specific tasks.

Unlike other accelerators mentioned in literature, HWPEs do not rely on an external DMA for input and output or being tied down to one core. Instead, they operate directly on the shared memory used by other elements, such as a general-purpose PE in a cluster, while their control is accessed through a peripheral bus or interconnect. Combining HW-based execution on an HWPE with general-purpose software code is easy as only pointers and configuration parameters need exchanging between them.

Figure 9 is giving an overview of a typical HWPE accelerator which is split into three domains: control, streamer, and engine.

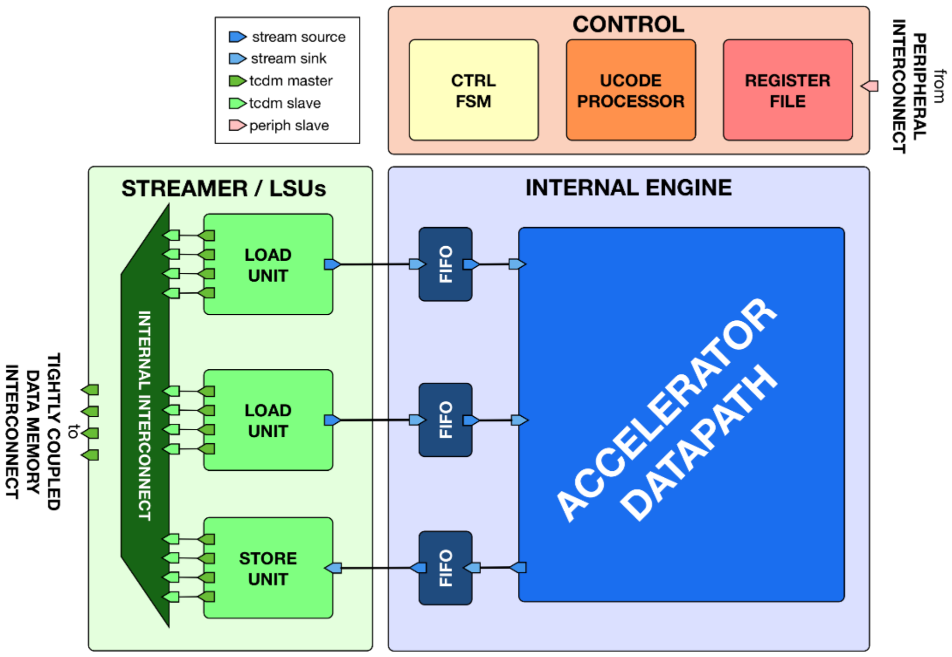


FIGURE 9: HWPE OVERVIEW.

[7] https://hwpe-doc.readthedocs.io/en/latest/chips.html
[8] https://hwpe-doc.readthedocs.io/en/latest/github.html

The **control** domain exposes a *control plane interface* used to program the accelerator. Typically, a general-purpose core writes a set of pointers and job configuration parameters over the interface into a register file which are used by a main finite-state-machine (FSM) to control the **streamer** and the **engine** of the accelerator. A set of IPs which can be used by partners to design the **control** domain can be found on the GitHub[9].

The **streamer** domain exposes a *data plane* interface and includes a set of address generators which are controlled by the main FSM of the **control** domain. The fetched data is typically converted into simple data streams which are valid/ready based and are given into the **engine**[10]. More information of the protocol conversion can be found in the documentation[11].

The **engine** domain contains the actual datapath of the accelerator and is controlled by the main FSM of the **control** domain. The engine typically also contains a set of local fine-grained FSMs.

A set of simpler example HWPE accelerators can be found here:

- Basic HWPE[12] example with basic streamers - MAC engine with single Multiply-Accumulate
- Basic HWPE example with HCI streamers - pure data mover[13]

A set of more complex example HWPE accelerators can be found here:

- Reconfigurable Binary Engine[14] - neural accelerator with flexible precision for weights and activations [33]
- Neural Engine[15] (16 input-channels) - neural accelerator with flexible precision for weights. The accelerator is used in the Gap9 SoC from Greenwaves[16]
- RedMulE[17] (REDuced-precision Matrix MULtiplication Engine) is a 8-bit and 16-bit floating-point systolic array [34]

In the following, the data plane and control plane interface which can be used for the L1-accelerator are described (see Figure 10).

[9] https://github.com/pulp-platform/hwpe-ctrl/
[10] https://hwpe-doc.readthedocs.io/en/latest/protocols.html#hwpe-stream-protocol
[11] https://hwpe-doc.readthedocs.io/en/latest/protocols.html#exchanging-data-between-hwpe-mem-and-hwpe-stream

[12] https://github.com/pulp-platform/hwpe-mac-engine
[13] https://github.com/pulp-platform/hwpe-datamover-example
[14] https://github.com/pulp-platform/rbe
[15] https://github.com/pulp-platform/ne16
[16] https://greenwaves-technologies.com/gap9_processor/
[17] https://github.com/pulp-platform/redmule

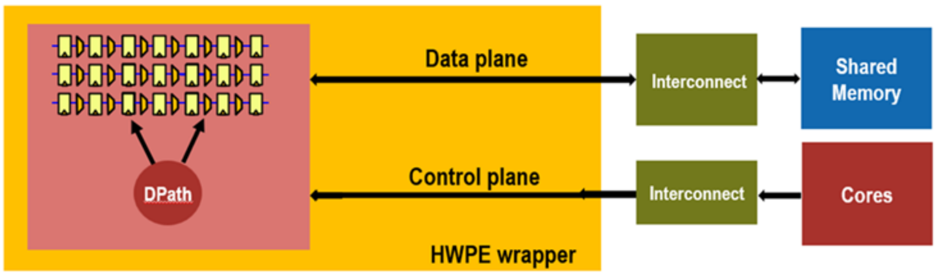


FIGURE 10: HWPE ABSTRACTION OF THE DATA PLANE AND CONTROL PLANE.

3.4.2. Data Plane

TCDM Interface

L1-Accelerators are connected to external L1/L2 shared memory by means of a simple memory protocol using a request/grant handshake. The protocol used is called Tightly-Coupled Data Memory (TCDM) protocol, which is very similar to the OBI[18], and HWPE-mem[19]. It is the same protocol as the one used by cores and DMAs operating on memories. It only supports individual transactions (no bursts) and assumes very tight coupling to memories with very low latencies.

TABLE 5: TCDM PROTOCOL.

Signal	Width [bits]	Direction	Description
`req`	1	*Master → Slave*	Handshake request signal (1=asserted).
`gnt`	1	*Slave → Master*	Handshake grant signal (1=asserted).
`add`	32	*Master → Slave*	Word-aligned memory address.
`wen`	1	*Master → Slave*	Write enable signal (1=read, 0=write).
`be`	4	*Master → Slave*	Byte enable signal (1=valid byte).
`data`	32	*Master → Slave*	Data word to be stored.
`r_data`	32	*Slave → Master*	Loaded data word.
`r_valid`	1	*Slave → Master*	Response valid (1=asserted).
`req`	1	*Master → Slave*	Handshake request signal (1=asserted).
`gnt`	1	*Slave → Master*	Handshake grant signal (1=asserted).

It supports neither multiple outstanding transactions nor bursts, as the accelerators are assumed to be closely coupled to memories, with single-cycle latencies when there is no contention. The TCDM protocol is used to connect a master to a slave. Table 5 reports the signals used by the TCDM protocol.

[18] https://github.com/openhwgroup/obi/blob/188c87089975a59c56338949f5c187c1f8841332/OBI-v1.5.0.pdf

[19] https://hwpe-doc.readthedocs.io/en/latest/protocols.html#hwpe-mem

The handshake signals `req` and `gnt` are used to validate transactions between masters and slaves. Transactions are subject to the following rules (see Figure 11 and Figure 12):

1. **A valid handshake occurs in the cycle when both `req` and `gnt` are asserted**. This is true for both write and read transactions.
2. **Every transaction is completed with the `r_valid` signal being asserted for one cycle.** In the case of read requests, the asserted `r_valid` d indicates that the requested data is now provided at `r_data`. For write transactions the asserted `r_valid` signal indicates the completion of the write request. In this case `r_data` contains invalid data.
3. **The assertion of `req` (transition 0 → 1) cannot depend combinationally on the state of `gnt`.** On the other hand, the assertion of gnt (transition 0 → 1) can depend combinationally on the state of req (and typically it does). This rule avoids deadlocks in ping-pong logic.

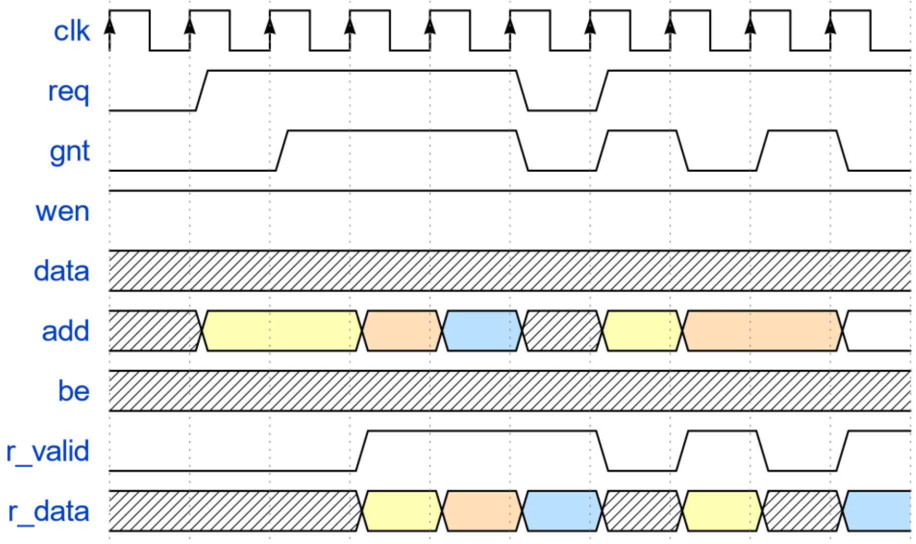


FIGURE 11: MULTIPLE TCDM READ REQUESTS.

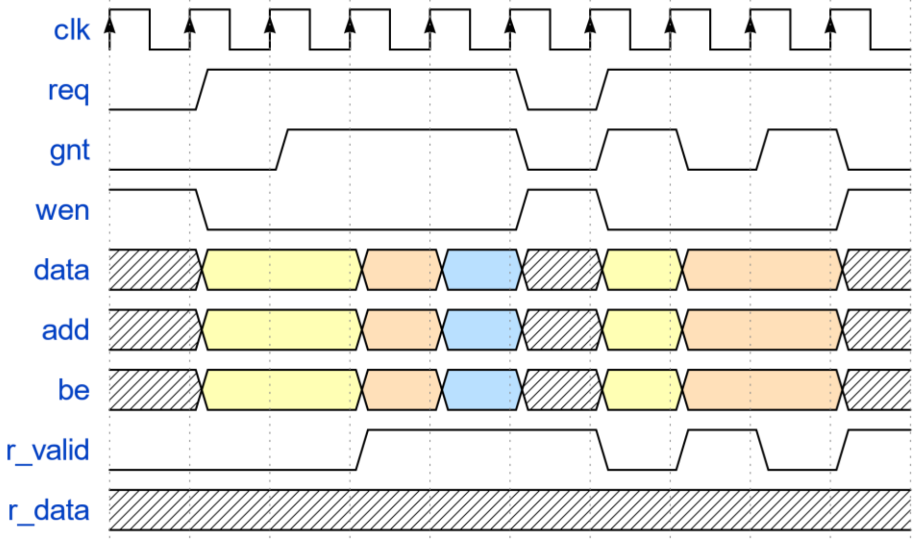


FIGURE 12: MULTIPLE TCDM WRITE REQUESTS.

HCI-Core Interface

HCI-Core (Heterogeneous Cluster Interconnect – Core) is a protocol designed as a lightweight extension of TCDM better suited for the needs of accelerators, specifically cluster-coupled HWPEs. If more bandwidth is required, we recommend using the HCI interface (see Section 3.1.2). The HCI-Core interface supports multiple outstanding transactions and is further specified in the documentation[20]. The interface provides signals in addition to the TCDM signals. Please discuss using these additional signals with ETHZ before accelerator design.

3.4.3. Control Plane

Peripheral Interface

To enable control of the Accelerators, they typically expose a slave port to the peripheral system interconnect (see Section 3.1.2). The slave port follows an extension of the TCDM protocol which we can call PERIPH. The PERIPH protocol is the same exposed by most peripherals in a PULP system and used by the GP cores to communicate with them. Table 6 gives an overview of all PERIPH signals.

The PERIPH protocol is distinguished by the TCDM protocol by the `id` and `r_id` side channels. They are used in load operations issued through a PERIPH interface: the `id` identifies the master during the request phase, is buffered by the slave peripherals and accompanies the response phase as `r_id`. In this way, multiple masters can distinguish which traffic is related to themselves.

TABLE 6: PERIPH PROTOCOL.

Signal	Width [bits]	Direction	Description
`req`	1	*Master → Slave*	Handshake request signal (1=asserted).
`gnt`	1	*Slave → Master*	Handshake grant signal (1=asserted).
`add`	32	*Master → Slave*	Word-aligned memory address.
`wen`	1	*Master → Slave*	Active-low write enable signal (1=read, 0=write).
`be`	4	*Master → Slave*	Byte enable signal (1=valid byte).
`data`	32	*Master → Slave*	Data word to be stored.
`id`	`ID_WIDTH`	*Master → Slave*	ID used to identify the master (request)
`r_data`	32	*Slave → Master*	Loaded data word.
`r_valid`	1	*Slave → Master*	Valid loaded data word (1=asserted).
`r_id`	`ID_WIDTH`	*Slave → Master*	ID used to identify the master (reply).

[20] https://hwpe-doc.readthedocs.io/en/latest/protocols.html#hci-core

Register Interface

A simple protocol for memory-mapped registers can also be used (see Figure 13 and Figure 14). Register generation is possible with the **reggen** tool using the version in the GitHub[21].

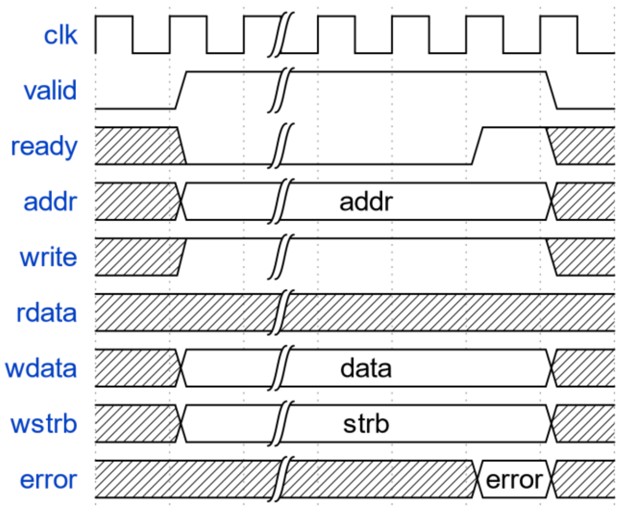


FIGURE 13: REGISTER INTERFACE WRITE TRANSACTION.

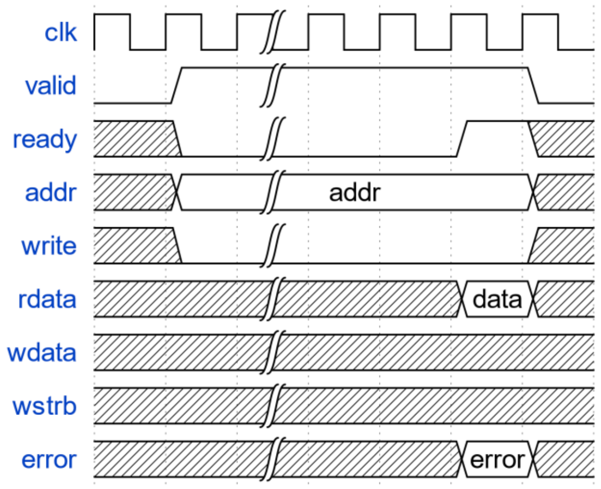


FIGURE 14: REGISTER INTERFACE READ TRANSACTION.

3.4.4. Programming Model

To initialize the accelerator, its configuration port is used, passing the required information. Typically, one PE writes to the memory-mapped control registers. The accelerator is responsible for managing its own data, fetching and storing it back to the TCDM memory. Typically, these accelerator types should allow multiple contexts for job configuration to achieve a performance on full applications and not only on kernels. Additionally, we recommend the use of a **done** signal, which can be connected to the event unit and is used to wake-up the PE once the offloaded job running on the accelerator is finished.

[21] https://github.com/pulp-platform/register_interface.

3.5. L2 – Accelerator: Loosely-Coupled Accelerator

An L2 accelerator is loosely coupled to the system, attaching to the global AXI-compatible interconnect or network.

3.5.1. Merged: Control & Data Plane: AXI

Both for control and data management, the accelerators use AXI4, compatible with the AXI4 specification from ARM[22]. For integration, using the pulp-platform AXI4 IPs[23] is recommended. The specification describes the protocol fields to be implemented for an AXI4 interface and how they should behave w.r.t. timing, ordering, etc., so we will abstain from providing further details here. The PULP AXI4 IPs provide a library of compatible interconnect management IPs that can be used for the design.

An accelerator will have two AXI ports, a single slave port allowing incoming requests for control, and a single master port allowing outgoing requests for data movement. Please note that these will have different ID widths for proper interconnect design.

3.5.2. Programming Model

To initialize the accelerator, its configuration port is used, passing the required information. The accelerator is responsible for managing its own data, fetching and storing it back to a main memory (e.g. L2). The architecture to handle the data movement is responsibility of the accelerator designer, but can be accomplished using a dedicated Direct Memory Access (DMA) engine, such as the iDMA[24]. Typically, these accelerator types should allow a form of double-buffering and multiple contexts for job configuration. Additionally, we recommend the use of a **done** signal, which can be connected to a system-level event unit and is used to wake-up the host core once the offloaded job running on the accelerator is finished.

[22] https://developer.arm.com/documentation/ihi0022/e/
[23] https://github.com/pulp-platform/axi
[24] https://github.com/pulp-platform/iDMA

4. Collaboration Tools

All hardware designed for the project is stored in corresponding `git` repositories, one for each IP, stored in the Eindhoven `gitlab` server.

Each IP will provide a `Bender.yml` file for the design, indicating its dependencies and the source files required for the design. This file is designed to be used with `bender`[25]. A full description of the format can be found in the `ReadMe` of `bender`.

`Git` and `bender` will be used to combine the IPs into a full SoC.

[25] https://github.com/pulp-platform/bender

5. Required Models and Views to achieve Goals of WP6

The next tasks of WP6 include the development and use of a compositional performance analysis model to model energy and latency at the SoC level, including the modelling of the SoC level memory hierarchy and processing host, as well as integrating the different accelerator component models of WP2/3. The use of such a model will moreover enable run-time performance assessment of an application when the platform configuration changes.

In the following the required accelerator models that will be needed from WP2/3 to such that the accelerators can be evaluated as part of the performance model that will be developed as part of T6.2 and used in T6.3. Firstly, there is the GVSoC simulator, which can be used for rapid SW prototyping and performance evaluation. Secondly, there is ZigZag which enables high-level accelerator design space explorations (DSE).

5.1. GVSoC Simulator

Simulating an entire heterogeneous SoCs which combines general-purpose cores with application-specific accelerators is rather complex and slow. Overall, this heterogeneity level requires a complex hardware and a full-fledged software stack to evaluate applications while exploiting all platform features. For this reason, enabling agile design space exploration becomes a crucial asset for exploring such heterogeneous SoCs. In this scenario, high-level simulators play an essential role in breaking the speed and design effort bottlenecks of cycle-accurate simulators and FPGA prototypes, respectively, while preserving functional and timing accuracy.

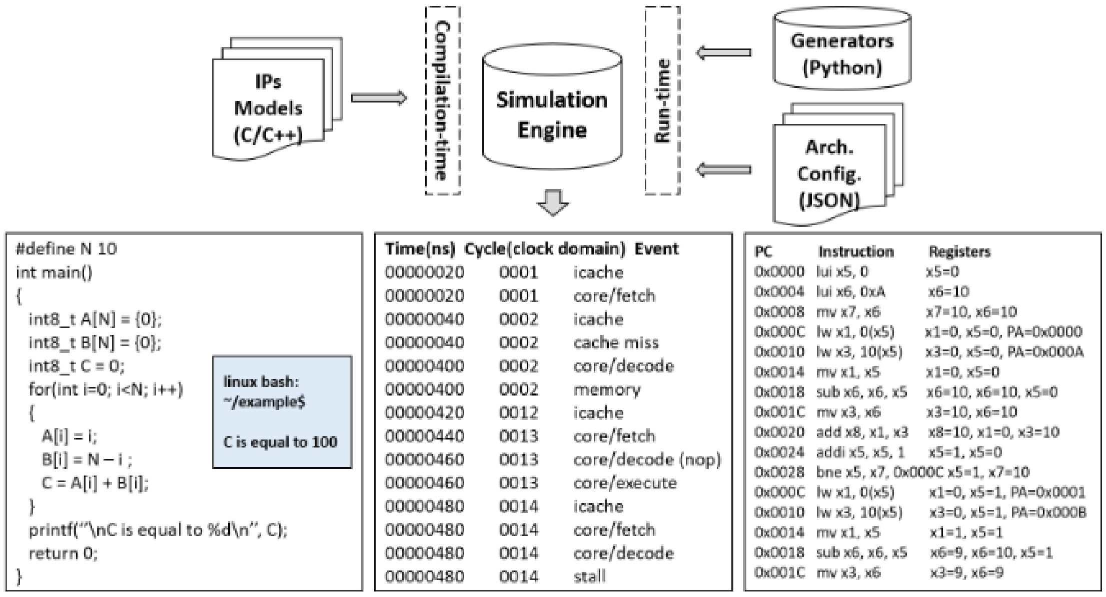
For this reason, the PULP project has developed GVSoC, a highly configurable and timing-accurate event-driven simulator that combines the efficiency of C++ models with the flexibility of Python configuration scripts. An overview is shown in Figure 15. GVSoC is fully open-sourced, with the intent to drive future research in the area of highly parallel and heterogeneous RISC-V based IoT processors, which aligns with the goals of WP6 in CONVOLVE.

GVSoC leverages three foundational features:

- Python-based modular configuration of the hardware description
- easy calibration of platform parameters for accurate performance estimation
- high-speed simulation

Overall, GVSoC enables practical functional and performance analysis and design exploration at the full-platform level (processors, accelerators, memory, peripherals and IOs) with a speed-up of 2500x with respect to cycle-accurate RTL simulation with cycle count errors typically below 10% for performance analysis [1].

The python-based modular configuration used to describe the entire platform plugs together a set of individual C++ models. Figure 15 gives an overview of the individual GVSoC components. For example, various accelerators can be developed individually and be enabled or disabled very easily for a system simulation. GVSoC already implements several L1-integrated accelerators (Section 3.4) and the standard interface to use as a template for new ones. Adding a new accelerator requires specifying its address space, the name and the number of ports, and the events that it can raise. Similarly, GVSoC partially supports already the use of multiple compute clusters (a type of L2-accelerators) [2] as described in Section 3.1.2.



```
#define N 10
int main()
{
  int8_t A[N] = {0};
  int8_t B[N] = {0};
  int8_t C = 0;
  for(int i=0; i<N; i++)
  {
    A[i] = i;
    B[i] = N – i ;
    C = A[i] + B[i];
  }
  printf("\nC is equal to %d\n", C);
  return 0;
}
```

```
linux bash:
~/example$

C is equal to 100
```

```
Time(ns)  Cycle(clock domain)  Event
00000020      0001      icache
00000020      0001      core/fetch
00000040      0002      icache
00000040      0002      cache miss
00000400      0002      core/decode
00000400      0002      memory
00000420      0012      icache
00000440      0013      core/fetch
00000460      0013      core/decode (nop)
00000460      0013      core/execute
00000480      0014      icache
00000480      0014      core/fetch
00000480      0014      core/decode
00000480      0014      stall
```

```
PC       Instruction        Registers
0x0000   lui x5, 0          x5=0
0x0004   lui x6, 0xA        x6=10
0x0008   mv x7, x6          x7=10, x6=10
0x000C   lw x1, 0(x5)       x1=0, x5=0, PA=0x0000
0x0010   lw x3, 10(x5)      x3=0, x5=0, PA=0x000A
0x0014   mv x1, x5          x1=0, x5=0
0x0018   sub x6, x6, x5     x6=10, x6=10, x5=0
0x001C   mv x3, x6          x3=10, x6=10
0x0020   add x8, x1, x3     x8=10, x1=0, x3=10
0x0024   addi x5, x5, 1     x5=1, x5=0
0x0028   bne x5, x7, 0x000C x5=1, x7=10
0x000C   lw x1, 0(x5)       x1=0, x5=1, PA=0x0001
0x0010   lw x3, 10(x5)      x3=0, x5=1, PA=0x000B
0x0014   mv x1, x5          x1=1, x5=1
0x0018   sub x6, x6, x5     x6=9, x6=10, x5=1
0x001C   mv x3, x6          x3=9, x6=9
```

FIGURE 15: OVERVIEW OF GVSOC'S MAIN COMPONENTS [1].

5.1.1. Required Models

GVSoC already supports a few L1-integrated accelerators which serve as an example for the required model:

- NE16[26] – a convolution accelerator which is in the GAP9 product from Greenwaves[27]
- IMA[28]– an in-memory-compute accelerator

These accelerators are compiled from the pulp-sdk[29] and with the traditional PULP Open cluster as it includes the low-level register map and HAL needed to run e.g., some kernels on the IMA accelerator.

For CONVOLVE, each accelerator should deliver a similar event-based C++ model (as those pointed to above) which functionally behaves the same way as the accelerator and is calibrated against a cycle-accurate simulation.

[26] https://github.com/gvsoc/gvsoc-pulp/tree/pulp_devel/models/pulp/ne16
[27] https://greenwaves-technologies.com/gap9_processor/
[28] https://github.com/gvsoc/gvsoc-pulp/tree/pulp_devel/models/pulp/ima
[29] https://github.com/pulp-platform/pulp-sdk

5.2. ZigZag Framework

Building efficient embedded deep learning (DL) systems requires a tight co-design between DNN algorithms, hardware, and algorithm-to-hardware scheduling, a.k.a. dataflow or mapping. Different hardware architectures (single-core or multi-core accelerators) are being designed, supporting many different scheduling possibilities, for different optimizing targets (energy, latency, memory footprint). However, owing to the large joint design space, finding an optimal solution through RTL simulation or physical implementation becomes infeasible. To tackle this problem, a unified high-level DL accelerator design space exploration (DSE) framework infrastructure, called ZigZag [35], will be used, and improved in the CONVOLVE project.
ZigZag[30] targets rapid DSE for DNN accelerator platforms supporting a broad set of hardware architectures and workload scheduling scenarios beyond other existing frameworks. Stream[31] is an extension of ZigZag [36] capable of modelling multi-core DNN acceleration employing fine-grained layer-fused processing.

5.2.1. Required Models

For CONVOLVE, each partner should deliver their accelerator model with the description of their architecture. For CONVOLVE, each partner should deliver their accelerator model with a description of their hardware architecture. ZigZag framework will help the partners to evaluate the scheduling and observe the interaction of their accelerator proposals with the different workloads. ZigZag gives the CONVOLVE partners the opportunity of optimizing their design choices driven by the observations of the system-level impact. Next subsections describe the following inputs required by ZigZag to describe the model:

- **Workload:** A neural network model defined in ONNX format or ZigZag's own format.
- **Hardware Architecture:** A high-level HW architecture description.
- **Mapping:** A file that specifies core allocation, spatial mapping, temporal ordering, and memory operand link.

Hardware Architecture

In this subsection, we introduce the general concept of how HW accelerators are modelled within ZigZag and the different well-known accelerators we provide as examples. We start from the smallest building block defined in ZigZag and work our way up towards an accelerator.

Operational Unit: Accelerating inference of a NN requires execution of multiplications and summations (accumulations) across multiple intermediate data (activations) using trained parameters (weights). The operational unit, typically a Multiplier, executes the multiplication of two data elements, typically an activation and a weight. `r_data`

The operational unit object has following attributes:

- `input_precision`: List of input operand (data) precision in number of bits for each input operand (typically 2 for Multiplier).
- `output_precision`: The bit precision of the operation’s output.

[30] https://github.com/ZigZag-Project/zigzag
[31] https://github.com/ZigZag-Project/stream

- `energy_cost`: Energy of executing a single multiplication.
- `area`: The HW area overhead of a single multiplier.

Operational Array: Inferencing a NN typically requires millions of operations, and an accelerator typically includes an array of operational units that can execute these operations. This can speed significantly up the computations, as well as increase energy efficiency which is covered later. The array has multiple dimensions, each with a size. The importance of these dimensions is explained in the introduction of the memory hierarchy.
The operational array object has:

- `operational_unit`: The operational unit from which the array is built.
- `dimensions`: The dimensions of the array. This should be defined as a python dictionary, with the keys being the identifier of each dimension of the array (typically 'D1', 'D2, …) and the values being the size of this dimension (i.e., the size of the array along that dimension).

Memory Instance: In order to store the different activations and weights used for the computations in the operational array, different memory instances are attached in a hierarchical fashion. The instances define how big each memory is in terms of capacity and area overhead, what the cost of writing and reading from these memories is, what it's bandwidth is, and how many read/write/read-write ports it includes.
The memory instance object has:

- `name`: A name for the instance
- `size`: The memory size in bits.
- `r_bw/w_bw`: A read and write bandwidth in number of bits per cycle.
- `r_cost/w_cost`: A read and write energy cost.
- `area`: Area overhead of the instance.
- `r_port/w_port/rw_port`: The number of read/write/read-write ports the instance has available.
- `latency`: The latency of an access in number of cycles.

Memory Hierarchy: Besides knowing what the specs of each memory instance are, the memory hierarchy encodes information with respect to the interconnection of the memories to the operational array, and to the other memory instances. This interconnection is achieved through multiple calls to the `add_memory()`, where the first call(s) adds the first level of memories, which connects to the operational array, and later calls connect to the lower memory levels. This builds a hierarchy of memories.

To know if the memory should connect to the operational array or another lower memory level, it needs to know which data will be stored within the memories. To decouple the algorithmic side from the hardware side, this is achieved through the concept of *'memory operands'* (as opposed to *'algorithmic operands which are typically the I/O activations and weights W'*). The designer can think of the memory operands as virtual operands, which will later be linked to the actual algorithmic operands in the mapping file through the `memory_operand_links` attribute.

Similarly to how the operational unit can be unrolled (forming an operational array), the memories can also be unrolled, where each memory accompanies either a single operational unit or all the operational units in one or more dimensions of the operational array. This is encoded through the `served_dimensions` attribute, which specifies if a single memory instance of this memory level serves all operational units in that dimension. This should be a set of one-hot-encoded tuples.

Lastly, the different `read`/`write`/`read-write` ports a memory instance has, are assigned to the different data movements possible in the hierarchy. There are four types of data movements in a hierarchy: from high (`fh`), to high (`th`), from low (`fl`), to low (`tl`). At the time of writing, these can be manually linked to one of the read/write/read-write ports through the following syntax: `{port_type}_port_{port_number}`, `port_type` being `r`, `w` or `rw` and `port_number` equal to the port number, starting from 1, which allows to allocate multiple ports of the same type. Alternatively, these are automatically generated as a default if not provided to the `add_memory()` call.

Internally, the `MemoryHierarchy` object extends the `NetworkXDiGraph` object, so its methods are available.
The memory hierarchy object includes:

- `operational_array`: The operational array to which this memory hierarchy will connect. This is required to correctly infer the interconnection through the operational array's dimensions. Through the `add_memory()` calls it adds a new `MemoryLevel` to the graph. This requires for each call a:
- `memory_instance`: A `MemoryInstance` object to be added to the hierarchy.
- `operands`: The virtual memory operands this `MemoryLevel` stores.
- `port_alloc`: The directionality of the memory instance's different ports, as described above.
- `served_dimensions`: The different dimensions that this memory level will serve, as described above.

Core: The operational array and the memory hierarchy together form a core of the accelerator. The core object includes:

- `id`: The id of this core.
- `operational_array`: The operational array of this core.
- `memory_hierarchy`: The memory hierarchy of this core.

HW Accelerator Model: Multiple cores are combined into the HW Accelerator, which is the main object modelling the HW behaviour.

The accelerator object includes:

- `name`: A user-defined name for this accelerator.
- `core_set`: The set of cores comprised within the accelerator.
- `global_buffer`: A memory instance shared across cores.

Examples:

We have modeled 5 well-known DNN accelerators in the ZigZag GitHub Repository[32] , which are Meta prototype [37], TPU [38], Edge TPU [39], Ascend [40], Tesla NPU [41], and, for our depth-first scheduling research. To make a fair and relevant comparison, we normalized all of them to have 1024 MACs and maximally 2MB global buffer (GB) but kept their spatial unrolling and local buffer settings, as shown in Table 7 (`IDX` 1/3/5/7/9). Besides, we constructed a variant of every normalized architecture (by changing its on-chip memory hierarchy), denoted with *'DF'* in the end of the name, as shown in Table 7 (`IDX` 2/4/6/8/10).

TABLE 7: SETTINGS EXAMPLE FOR ZIGZAG FOR SEVERAL EXISTING HARDWARE ARCHITECTURES. K IS FOR OUTPUT CHANNEL; C IS FOR INPUT CHANNEL; OX AND OY ARE THE OUTPUT FEATURE MAP'S SPATIAL DIMENSIONS; FX AND FY ARE THE WEIGHT'S SPATIAL DIMENSIONS.

IDX	HW Architecture	Spatial Unroll. (1024 MACs)	Register per MAC or MAC group	Local Buffer	2nd level LB	Global Buffer (max: 2MB)
1	Meta-proto-like	*K 32* *C 2* *OX 4* *OY 4*	W: 1B O: 2B	W: 64B I: 32KB	-	W: 1MB I&O: 1MB
2	Meta-proto-like DF			W: 32B I&O: 64KB	-	
3	TPU-like	*K 32* *C 32*	W: 128B O: 1KB	-	-	I&O: 2MB
4	TPU-like DF		W: 64B O: 1KB	I&O: 64KB	-	W: 1MB I&O: 1MB
5	Edge-TPU-like	*K 8* *C 8* *OX 4* *OY 4*	W: 1B O: 2B	W: 32KB	-	I&O: 2MB
6	Edge-TPU-like DF			W: 16KB I&O: 16KB	-	W: 1MB I&O: 1MB
7	Ascend-like	*K 16* *C 16* *OX 2* *OY 2*	W: 1B O: 2B	W: 64B I: 64KB O: 256KB	-	W: 1MB I&O: 1MB
8	Ascend-like DF			W: 64B I&O: 64KB	I&O:256KB	
9	Tesla-NPU-like	*K 32* *OX 8* *OY 4*	W: 1B O: 2B	W: 1B I: 1KB	-	W: 1MB I&O: 1MB
10	Tesla-NPU-like DF			W: 1B I: 1KB	W: 64KB; I&O: 64KB	W: 1MB I&O: 896KB

Workload

The recommended way of defining an algorithmic workload is through an ONNX model. An ONNX model can contain multiple operator types, which in the context of ML are often referred to as layers, some of which are automatically recognised and parsed by ZigZag. Alternatively, the layers can be manually defined for more customization. Following operators are supported by ZigZag and will automatically be parsed into `LayerNode` objects when using a ONNX model within the framework:

- `Conv`
- `QLinearConv`
- `MatMul`

[32] https://github.com/ZigZag-Project/zigzag

Manual layer definition: It is also possible to manually define custom workload layers. In that case there the `main.py` file should be executed instead of `main_onnx.py`. Moreover, the workload file should be provided as input together with the accelerator, thus there is no ONNX model and mapping file loaded. The mapping information is inserted for each layer alongside the layer shape definition, identically to how it was defined in the mapping file.

Each layer definition is represented as a dictionary which should have the following attributes: equation: The operational equation for this layer. The dimensions should be small letters, whereas the operands are large letters. `O` should always be used for the output operand; the input operands can be named freely.

- `dimension_relations`: The relationship between different dimensions presents in the equation. This is often used in convolutional layers, where there is a relationship between the spatial input indices and the spatial output indices through the stride and with the filter indices through the dilation rate.

- `loop_dim_size`: The size of the different dimensions presents in the equation. Dimensions defined (i.e., on the left-hand side) in the **dimension_relations** are not to be provided and are inferred automatically.

- `operand_precision`: The bit precision of the different operands presents in the equation. `O` should always be used, which represents the partial output precision. `O_final` represents the final output precision.

- `operand_source`: The layer id the input operands of this layer come from. This is important to correctly build the NN graph edges.

- `constant_operands`: The operands of this layer which are constants and do not depend on prior computations.

- `core_allocation`: The core that will execute this layer.

- `spatial_mapping`: The spatial parallelization strategy used for this layer. If none is provided, the `SpatialMappingGeneratorStage` should be used within ZigZag's execution pipeline.

- `memory_operand_links`: The link between the virtual memory operands and the actual algorithmic operands. For more information, read the hardware readme.

Mapping

The mapping defines how the algorithmic operations are mapped onto the computational hardware resources. The ZigZag framework automates (parts of) this mapping, but some aspects need to be (at the time of writing) user defined. The mapping input file is required for running ZigZag in combination with the ONNX interface. When manually defining the algorithmic layers, the mapping information is encoded within the workload definition.

The mapping file should contain following aspects for every ONNX node that will be mapped onto the accelerator:

- `core_allocation`: The accelerator core id onto which this ONNX node is mapped (the id provided when creating the core in the HW description file).
- `spatial_mapping`: The spatial parallelization strategy to execute the node with (this can be automated through the `SpatialMappingGeneratorStage`).
- `memory_operand_links`: The memory operand links, which link the memory operands (defined in the memory hierarchy of the core) to the layer operands (which are generated in the `ONNXModelParserStage` and are typically `O`, `I`, `W` for a convolutional layer). This extra memory mapping is added to allow flexible memory allocation schemes. A default entry can also be defined. This is useful to have different ONNX node names, or for customizing the workload for every mapped node. The default entry is automatically detected under the default key of the mapping dictionary.

6. Overview of WP6 Work Plan

In this section, an overview of the ongoing and upcoming tasks as well as deliverables of WP 6 are given. An overview of all tasks, deliverables including the expected timelines is given in Table 8.

6.1. Task 6.2: Performance Analysis and Management of ML Applications on Modular Architectures

In this task, the partners will work on the modelling infrastructure to model energy and latency at the SoC level considering the SoC host and overall memory hierarchy, as well as the smooth integration of the new accelerator models developed by WP2/3 and specified in Section 5.2. The development of efficient compositional models focuses on enabling both design-time and run-time (dynamic) flexibility and re-configuration. Furthermore, the compositional models will enable run-time performance assessment of an application when the platform configuration changes.

6.2. Task 6.3: Modular Flexibility-Aware DSE Framework for Efficiency and Fault-Tolerance

In this task, the DSE framework ZigZag will be extended with modelling capabilities for multi-core compute cluster, multi-accelerator, as well as layout-aware cost modelling. In addition, the DSE framework will be enhanced to model the memory hierarchy and architecture of the open-source hardware IPs used to design the SoC template described in this deliverable D6.1. The goal is that the DSE framework which can derive the optimal combination of accelerators, together with the best design- and run-time flexibility parameter (ranges), given target workloads and application constraints. This task builds on the models of T6.2, as well as the compiler directives of WP5.

6.3. Task 6.2: Rapid Design Instantiation and Validation of Modular and Flexible Architectures

This task enabled the rapid instantiation and verification of the optimized multi-accelerator machine learning SoC architectures. This includes the development of the parameterized (System)Verilog SoC template and the supported interfaces described in this deliverable, as well as the extension and development of the simulator. Furthermore, the partners will work on automated integration of the different components, including testbench generation, for the parametrized SoC architectures. The verification flow orients on the industry needs and includes tests for functional correctness, timing criticality, security, and fault tolerance into the methodology. The task will result in a prototype SoC developed with the target methodology.

TABLE 8: OVERVIEW AND TIMELINE OF WP6.

Working Package 6	Participants	3	6	9	12	15	18	21	24	27	30	33
Compositional architecture DSE and SoC generation		▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
T6.1 Modular architecture template definition	ETHZ, TUE, TUD, KUL, RUB, UMU	▨	▨									
T6.2 Performance analysis and management of ML Applications on Modular Architectures	TUE, FMI, KUL, BOS, NXP, UED, UMU	▨	▨	▨	▨	▨	▨	▨	▨			
T6.3 Modular flexibility-aware DSE framework for efficiency and fault-tolerance	KUL, ETHZ, FMI, BOS		▨	▨	▨	▨	▨	▨	▨	▨		
T6.4 Rapid design instantiation and validation of modular and flexible Architectures	BOS, ETHZ, KUL, RUB, UED				▨	▨	▨	▨	▨	▨	▨	▨

D6.1 Modular architecture template definition	Lead ETHZ		■									
D6.2 Description of the gen1 performance analysis framework and DSE framework	Lead TUE						■					
D6.3 Description SoC architecture, and the rapid design & prototyping environment	Lead BOS						■					
D6.4 Final integrated modelling, design exploration and generation tool flow	Lead KUL											■

7. Conclusion

To summarize, this deliverable D6.1 specified the high-level RISC-V SoC template and standardized interfaces to enable a modular and easy integration of multiple ML and security accelerators. The described three integration levels cover various accelerator design styles: L0 co-processors, L1 tightly coupled accelerators, and the L2 loosely coupled accelerators. In addition, the deliverable specifies the accelerator models required to consider the individual accelerators in a compositional simulator based on GVSoC and in the high-level accelerator design space explorations based on ZigZag.

Overall, this specification deliverable D6.1 lays the foundation to successfully combine the work of individual CONVOLVE work packages to enable a rapid and modular design flow for heterogeneous ultra-low-power reliable and secure edge AI devices, including their SW and compiler infrastructure.

8. References

[1] N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, "GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors," in *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Institute of Electrical and Electronics Engineers Inc., 2021, pp. 409–416. doi: 10.1109/ICCD53106.2021.00071.

[2] N. Bruschi *et al.*, "Scale up your In-Memory Accelerator: Leveraging Wireless-on-Chip Communication for AIMC-based CNN Inference," in *Proceeding - IEEE International Conference on Artificial Intelligence Circuits and Systems, AICAS 2022*, Institute of Electrical and Electronics Engineers Inc., 2022, pp. 170–173. doi: 10.1109/AICAS54282.2022.9869996.

[3] A. Garofalo *et al.*, "A 1.15 TOPS/W, 16-Cores Parallel Ultra-Low Power Cluster with 2b-to-32b Fully Flexible Bit-Precision and Vector Lockstep Execution Mode," in *ESSCIRC 2021 - IEEE 47th European Solid State Circuits Conference, Proceedings*, Institute of Electrical and Electronics Engineers Inc., Sep. 2021, pp. 267–270. doi: 10.1109/ESSCIRC53450.2021.9567767.

[4] "STM32L4R5xx STM32L4R7xx STM32L4R9xx." [Online]. Available: www.st.com

[5] "This is information on a product in full production. STM32U575xx Ultra-low-power Arm ® Cortex ®-M33 32-bit MCU+TrustZone ® +FPU, 240 DMIPS, up to 2 MB Flash memory, 786 KB SRAM Datasheet-production data Features Includes ST state-of-the-art patented technology Ultra-low-power with FlexPowerControl Core • Arm ® 32-bit Cortex ®-M33 CPU with TrustZone ® , MPU, DSP, and FPU," 2023. [Online]. Available: www.st.com

[6] J. Yue *et al.*, "7.5 A 65nm 0.39-to-140.3TOPS/W 1-to-12b Unified Neural Network Processor Using Block-Circulant-Enabled Transpose-Domain Acceleration with 8.1 × Higher TOPS/mm2and 6T HBST-TRAM-Based 2D Data-Reuse Architecture," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 138–140. doi: 10.1109/ISSCC.2019.8662360.

[7] Z. Yuan *et al.*, "A Sparse-Adaptive CNN Processor with Area/Performance balanced N-Way Set-Associate PE Arrays Assisted by a Collision-Aware Scheduler; A Sparse-Adaptive CNN Processor with Area/Performance balanced N-Way Set-Associate PE Arrays Assisted by a Collision-Aware Scheduler," 2019.

[8] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo, "7.7 LNPU: A 25.3TFLOPS/W Sparse Deep-Neural-Network Learning Processor with Fine-Grained Mixed Precision of FP8-FP16," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 142–144. doi: 10.1109/ISSCC.2019.8662302.

[9] J. Song *et al.*, "7.1 An 11.5TOPS/W 1024-MAC Butterfly Structure Dual-Core Sparsity-Aware Neural Processing Unit in 8nm Flagship Mobile SoC," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 130–132. doi: 10.1109/ISSCC.2019.8662476.

[10] S. Ryu *et al.*, "A 44.1TOPS/W Precision-Scalable Accelerator for Quantized Neural Networks in 28nm CMOS," in *2020 IEEE Custom Integrated Circuits Conference (CICC)*, 2020, pp. 1–4. doi: 10.1109/CICC48029.2020.9075872.

[11] Z. Yuan *et al.*, "14.2 A 65nm 24.7μJ/Frame 12.3mW Activation-Similarity-Aware Convolutional Neural Network Video Processor Using Hybrid Precision, Inter-Frame Data Reuse and Mixed-Bit-Width Difference-Frame Data Codec," in *2020*

IEEE International Solid- State Circuits Conference - (ISSCC), 2020, pp. 232–234. doi: 10.1109/ISSCC19947.2020.9063155.

[12] Z. Li *et al.*, "An 879GOPS 243mW 80fps VGA Fully Visual CNN-SLAM Processor for Wide-Range Autonomous Exploration," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 134–136. doi: 10.1109/ISSCC.2019.8662397.

[13] Y.-C. Lo *et al.*, "Physically Tightly Coupled, Logically Loosely Coupled, Near-Memory BNN Accelerator (PTLL-BNN)," in *ESSCIRC 2019 - IEEE 45th European Solid State Circuits Conference (ESSCIRC)*, 2019, pp. 241–244. doi: 10.1109/ESSCIRC.2019.8902909.

[14] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision," *IEEE J Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, 2019, doi: 10.1109/JSSC.2018.2865489.

[15] B. Moons, D. Bankman, L. Yang, B. Murmann, and M. Verhelst, "BinarEye: An always-on energy-accuracy-scalable binary CNN processor with all memory on chip in 28nm CMOS," in *2018 IEEE Custom Integrated Circuits Conference (CICC)*, 2018, pp. 1–4. doi: 10.1109/CICC.2018.8357071.

[16] I. A. Papistas *et al.*, "A 22 nm, 1540 TOP/s/W, 12.1 TOP/s/mm2 in-Memory Analog Matrix-Vector-Multiplier for DNN Acceleration," in *2021 IEEE Custom Integrated Circuits Conference (CICC)*, 2021, pp. 1–2. doi: 10.1109/CICC51472.2021.9431575.

[17] H. Jia *et al.*, "15.1 A Programmable Neural-Network Inference Accelerator Based on Scalable In-Memory Computing," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, 2021, pp. 236–238. doi: 10.1109/ISSCC42613.2021.9365788.

[18] P.-C. Wu *et al.*, "A 28nm 1Mb Time-Domain Computing-in-Memory 6T-SRAM Macro with a 6.6ns Latency, 1241GOPS and 37.01TOPS/W for 8b-MAC Operations for Edge-AI Devices," in *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, 2022, pp. 1–3. doi: 10.1109/ISSCC42614.2022.9731681.

[19] J. Yue *et al.*, "15.2 A 2.75-to-75.9TOPS/W Computing-in-Memory NN Processor Supporting Set-Associate Block-Wise Zero Skipping and Ping-Pong CIM with Simultaneous Computation and Weight Updating," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, 2021, pp. 238–240. doi: 10.1109/ISSCC42613.2021.9365958.

[20] J. Yue *et al.*, "14.3 A 65nm Computing-in-Memory-Based CNN Processor with 2.9-to-35.8TOPS/W System Energy Efficiency Using Dynamic-Sparsity Performance-Scaling Architecture and Energy-Efficient Inter/Intra-Macro Data Reuse," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 234–236. doi: 10.1109/ISSCC19947.2020.9062958.

[21] Q. Liu *et al.*, "33.2 A Fully Integrated Analog ReRAM Based 78.4TOPS/W Compute-In-Memory Chip with Fully Parallel MAC Computing," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 500–502. doi: 10.1109/ISSCC19947.2020.9062953.

[22] Z. Chen, X. Chen, and J. Gu, "15.3 A 65nm 3T Dynamic Analog RAM-Based Computing-in-Memory Macro and CNN Accelerator with Retention Enhancement, Adaptive Analog Sparsity and 44TOPS/W System Energy Efficiency," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, 2021, pp. 240–242. doi: 10.1109/ISSCC42613.2021.9366045.

[23] E. Flamand *et al.*, "GAP-8: A RISC-V SoC for AI at the Edge of the IoT," in *29th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2018, Milano, Italy, July 10-12, 2018*, IEEE Computer Society, 2018, pp. 1–4. doi: 10.1109/ASAP.2018.8445101.

[24] GreenWaves, "GAPuino."

[25] Syntiant, "NDP120."

[26] D. Rossi *et al.*, "Vega: A Ten-Core SoC for IoT Endnodes with DNN Acceleration and Cognitive Wake-Up from MRAM-Based State-Retentive Sleep Mode," *IEEE J Solid-State Circuits*, vol. 57, no. 1, pp. 127–139, Jan. 2022, doi: 10.1109/JSSC.2021.3114881.

[27] I. Miro-Panades *et al.*, "SamurAI: A 1.7MOPS-36GOPS Adaptive Versatile IoT Node with 15,000× Peak-to-Idle Power Reduction, 207ns Wake-Up Time and 1.3TOPS/W ML Efficiency," in *2020 IEEE Symposium on VLSI Circuits*, 2020, pp. 1–2. doi: 10.1109/VLSICircuits18222.2020.9163000.

[28] M. Molendijk, F. de Putter, M. Gomony, P. Jääskeläinen, and H. Corporaal, "BrainTTA: A 35 fJ/op Compiler Programmable Mixed-Precision Transport-Triggered NN SoC," Nov. 2022, [Online]. Available: http://arxiv.org/abs/2211.11331

[29] M. Scherer, A. Di Mauro, G. Rutishauser, T. Fischer, and L. Benini, "A 1036 TOp/s/W, 12.2 mW, 2.72 μJ/Inference All Digital TNN Accelerator in 22 nm FDX Technology for TinyML Applications," in *25th IEEE Symposium on Low-Power and High-Speed Chips and Systems, COOL Chips 2022 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., 2022. doi: 10.1109/COOLCHIPS54332.2022.9772668.

[30] V. Jain, S. Giraldo, J. De Roose, L. Mei, B. Boons, and M. Verhelst, "TinyVers: A Tiny Versatile System-on-Chip With State-Retentive eMRAM for ML Inference at the Extreme Edge," *IEEE J Solid-State Circuits*, pp. 1–12, Jan. 2023, doi: 10.1109/jssc.2023.3236566.

[31] P. Houshmand *et al.*, "DIANA: An End-to-End Hybrid DIgital and ANAlog Neural Network SoC for the Edge," *IEEE J Solid-State Circuits*, vol. 58, no. 1, pp. 203–215, Jan. 2023, doi: 10.1109/JSSC.2022.3214064.

[32] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020, doi: 10.1145/3361682.

[33] F. Conti *et al.*, "22.1 A 12.4TOPS/W @ 136GOPS AI-IoT System-on-Chip with 16 RISC-V, 2-to-8b Precision-Scalable DNN Acceleration and 30%-Boost Adaptive Body Biasing," in *2023 IEEE International Solid- State Circuits Conference (ISSCC)*, IEEE, Feb. 2023, pp. 21–23. doi: 10.1109/ISSCC42615.2023.10067643.

[34] Y. Tortorella, L. Bertaccini, D. Rossi, L. Benini, and F. Conti, "RedMulE: A Compact FP16 Matrix-Multiplication Accelerator for Adaptive Deep Learning on RISC-V-Based Ultra-Low-Power SoCs," in *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*, in DATE '22. Leuven, BEL: European Design and Automation Association, 2022, pp. 1099–1102.

[35] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, 2021, doi: 10.1109/TC.2021.3059962.

[36] A. Symons, L. Mei, S. Colleman, P. Houshmand, S. Karl, and M. Verhelst, "Towards Heterogeneous Multi-core Accelerators Exploiting Fine-grained Scheduling of

Layer-Fused Deep Neural Networks," Dec. 2022, [Online]. Available: http://arxiv.org/abs/2212.10612

[37] H. E. Sumbul *et al.*, "System-Level Design and Integration of a Prototype AR/VR Hardware Featuring a Custom Low-Power DNN Accelerator Chip in 7nm Technology for Codec Avatars," in *Proceedings of the Custom Integrated Circuits Conference*, Institute of Electrical and Electronics Engineers Inc., 2022. doi: 10.1109/CICC53496.2022.9772810.

[38] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings - International Symposium on Computer Architecture*, Institute of Electrical and Electronics Engineers Inc., Jun. 2017, pp. 1–12. doi: 10.1145/3079856.3080246.

[39] K. Seshadri, B. Akin, J. Laudon, R. Narayanaswami, and A. Yazdanbakhsh, "An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks," in *Proceedings - 2022 IEEE International Symposium on Workload Characterization, IISWC 2022*, Institute of Electrical and Electronics Engineers Inc., 2022, pp. 79–91. doi: 10.1109/IISWC55918.2022.00017.

[40] H. Liao *et al.*, "Ascend: A Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : stry Track Paper," in *Proceedings - International Symposium on High-Performance Computer Architecture*, IEEE Computer Society, Feb. 2021, pp. 789–801. doi: 10.1109/HPCA51647.2021.00071.

[41] E. Talpes *et al.*, "Compute solution for tesla's full self-driving computer," *IEEE Micro*, vol. 40, no. 2, pp. 25–35, Mar. 2020, doi: 10.1109/MM.2020.2975764.