ALMA MATER STUDIORUM – UNIVERSITÁ DI BOLOGNA

School of Engineering and Architecture

Master Degree in Electronics Engineering

Master Thesis in

HW/SW Design for Embedded Systems M

Design of a memory-to-memory tensor reshuffle unit for ultra-low-power deep learning accelerators

Candidate

Riccardo Gandolfi

Supervisor

Francesco Conti

Co-supervisor

Davide Rossi

Session I

Academic Year 2020/2021

Summary

Introduction

The work done for this thesis was carried out in the context of the PULP (*Parallel Ultra Low Power*) platform, a conjoined project between the University of Bologna and the ETH of Zurich. The PULP SoC (*System on Chip*) is a multicore platform comprising of a MCU (*Microcontroller Unit*) Subsystem and a cluster of RISCV cores capable of achieving very high energy efficiency while retaining flexibility and programmability, crucial features in the field of embedded systems and IoT (*Internet of Things*) applications.

Especially in the field of IoT, the perspective of moving the elaboration towards the endpoint sensors is a promising direction, since it allows to reduce both the energy spent on communication and the network load by exploiting extreme-edge computing and near-sensor analytics.

Also, Deep Neural Networks (DNNs) are experiencing a rapid growth in importance. These networks are able to offer high performance and accuracy in a wide range of artificial intelligence and image processing tasks but are quite resource demanding. To mitigate these costs, data marshaling and reordering operations could be carried out on data stored into memory. This approach could prove useful to reduce the costs associated to data movements and communications.

Also, the need for high performance and low-latency data transfers within MCUs has brought more and more focus on accelerators and DMAs for low power hardware platforms such as PULP. In this context, the following thesis illustrates a lightweight and easily configurable accelerator to perform data marshaling operations on tensors stored in the TCDM memory of a PULP cluster. This is particularly useful to perform HWC to CHW conversion on data utilized by image processing applications, with HWC (*Height-Width-Channel*) and CHW (*Channel-Height-Width*) being the two most common formats in which image data is stored into memory.

The proposed accelerator will be part of the Darkside prototype, a chip under design during the time frame of the development of this thesis work. This chip implements the PULP platform in tsmc65 technology and includes both a SoC and a cluster of 8 RISCV cores. This chip is mainly focused on image processing applications, and it's featured with a variety of hardware accelerators. Besides the Datamover, the cluster is provided with a tensor processing unit and a depthwise accelerator.

The first chapter will focus on a brief overview of the near-sensor analytics trend in IoT applications and extreme-edge processing in MCUs. The main advantages deriving from implementation of learning-based algorithms will be shown and a few state-of-the-art solutions will be illustrated. Then, an overview of the PULP platform architecture will follow.

The second chapter will concentrate on data marshaling in deep learning-based applications. The HWC and CHW formats will be illustrated and the need for data marshaling will be discussed.

In the third chapter, the Datamover structure and functionality will be discussed in detail. Particular attention will be given to the internal structure, the instantiated modules, and the utilized communication protocols used to interface the accelerator with the rest of the cluster. Also, the integration of the Datamover within the SoC will be explored and detailed. Finally, the internal buffering mechanism will be described, and focus will be given to how it can enable the accelerator to perform reordering operations on data widths lower than 32 bits.

In the fourth and final chapter, the synthesis procedure and setup for the Datamover will be shown and the related results will be discussed in terms of area occupation, power consumption and timing constraints. At the end, experimental results will be provided for the whole Darkside SoC.

1: Near-Sensor Analytics Devices

There are two main challenges to tackle when entering the field of near-sensor analytics. The first one being the need for collecting meaningful data-streams directly from endpoint sensors and use them in learning-based algorithms. This poses a challenge in terms of resources, with the MCUs being forced to operate in a critical environment in terms of memory size, power consumption and computational power. Of course, operating on-device allows for speed and power consumption to be optimized, but poses a serious challenge in terms of implementation and resource management.

The second main challenge is to efficiently provide data-security and privacy for the information being elaborated on the IoT devices. This is all the more important when considering the wide range of IoT applications, spanning from aerial surveillance to health monitoring and home automation. This can be dealt with by means of encryption engines, which however add a significant workload on the device, further increasing the computational stress on the end-nodes.

These challenges are, however, balanced by some promising points of strength. By exploiting end-nodes capable of elaborating data directly on the spot, lower latency is introduced with respect to a cloud-based computing solution. The key difference is to elaborate data as close as possible to its gathering point, which allows better support for latency-sensitive applications, and also allows for better scalability and flexibility. By not relying on network connectivity and a centralized computing solution, edge-computing can also be extended to remote areas and emergency situations, all scenarios in which a cloud-based solution would prove far less efficient and reliable.

1.1: Processing at the Extreme Edge

The challenges mentioned above derive from the trend of moving the computation in IoT applications towards the end-nodes, as to avoid transmitting large volumes of unprocessed data and allow elaboration directly on the spot. In this context, the IoT end-nodes must shift their role, from simple data collectors to being able to perform a pre-selection of valuable data and obtain high-level features to transmit onward in the system.

These nodes could increase their capabilities by adding specialized hardware and encryption engines, thus optimizing on-the-spot computation, and simultaneously addressing security concerns. Several trade-offs are to be considered though.

1.2: Traditional and Deep Learning-based Near-Sensor Analytics

IoT end-nodes must operate in a severe resource-constrained environment, thus having to deal with trade-offs between power consumption and computational power. The majority of current solutions rely on small batteries, being able to handle only low-bandwidth sensors (the likes of temperature and pressure sensors). In this context, many approaches rely on heavy duty cycling and deep-sleep power states, to achieve power consumption rates in the order of few tens of nW. Other approaches are based on high degrees of circuit-level optimization, with the implementation of leakage suppression modes, with the goal to provide minimum power consumption at very low frequencies (few Hz [1]).

These traditional approaches are undoubtedly limited when facing the requirements of the most recent IoT applications, focused on obtaining rich and meaningful data directly from the end-nodes, such as audio or video streaming, and performing computationally intensive tasks directly on the field of operation [1].

Luckily, DNNs and CNNs (*Convolutional Neural Networks*) algorithms can provide a crucial support in moving complex computations on MCUs and transmitting only classes and high-level features to the higher hierarchies of the system. This approach allows to avoid sending large volumes of raw sensor data, thus reducing the energy consumption and time needed for data communications. DNNs are indeed capable of delivering high accuracy and outstanding performances on a wide range of application fields, from speech recognition to image processing and artificial intelligence. Versatility and performances come at a price though. The computational complexity related to this kind of algorithms is quite high, with millions of MAC operations and a large memory footprint being required [2].

So, a crucial factor in near-sensor analytics is to move these complex and resource-demanding algorithms on the end-nodes of IoT systems and still achieve acceptable functionality and performances.

1.3: State-of-the-Art in SoC's for Near-Sensor Analytics

As previously mentioned, the need for secure and reliable data elaboration is one of the crucial requirements in near-sensor analytics, and the related workload is all the more demanding when combined with the one deriving from DNNs algorithms. The resulting computational effort required from the edge devices quickly becomes prohibitive and calls for urgent solutions.

The solution illustrated in [2] tackles these challenges by proposing a specialized SoC named Fulmine where multiple engines, either programmable cores or hardware accelerators, are connected to the L1 memory through a low-latency interconnect. This SoC is capable of sustaining intensive data analytics workloads with optimal security and privacy features thanks to an embedded encryption engine. As already mentioned, encryption engines take a considerable toll on the global workload, thus the integration with an efficient low-power system is mandatory. The studied use cases include aerial surveillance, CNN-based detection, and seizure detection, with a maximum energy consumption of 12.7 pJ per operation.

In [3] another solution is proposed, by relying on low bit-width QNNs (*Quantized Neural Networks*), complex machine learning models and algorithms can be deployed on IoT end-nodes, with the beneficial effect of reducing the required memory footprint. This can be done by exploiting the quantization technique, allowing to reduce inputs and weights to fixed-point formats such as 8-bits or even going to sub-byte widths.

Complexity though, is added on another side. Given the lack of sub-byte instructions in state-of-the-art microprocessors, this optimization becomes hard to exploit. Thus, the need for specialized SIMD (*Single Instruction Multiple Data*) instructions arises. The exciting feature is that each operand precision is set dynamically in a core status register rather than being explicitly encoded. This avoids increasing complexity in decoding stages and saturating the encoding space when compared to a variable-length instruction approach. To this end, a new RISCV ISA core MPIC (*Mixed Precision Inference Core*) is proposed.

In [1] the Mr. Wolf chip is proposed as a PULP SoC for IoT edge processing provided with a wide set of peripherals and an autonomous IO subsystem. Mr. Wolf also features an eight-core cluster to be utilized for the most computational-intensive tasks and an aggressive on-chip power management system to enable energy-efficient operation for always-on IoT end-nodes. This allows to improve performance by several orders of magnitude with respect to a traditional

single-core MCU. This is done by exploiting several on-chip memory-sharing techniques and implementing a parallel programmable processing engine for multi-sensor data analysis and fusion.

In [4], the Envision platform is proposed as a way to achieve higher performances than most state-of-the-art ConvNet (*Convolutional Networks*) accelerators and GPUs (by one order of magnitude) and meet the requirements for always-on applications. The main feature of the Envision platform is the possibility to scale the energy consumption while maintaining recognition rate and throughput in visual recognition applications. Specifically, the concept of hierarchical recognition is put to use, where many different and individually trained convolutional networks can be exploited. These networks differ for size, topology and computational precision, allowing to scale the energy consumption while constantly scanning an object and to choose the accuracy needed case-by-case.

Another solution with a degree of reconfigurability and flexibility is proposed in [5]. Eyeriss is a reconfigurable accelerator for deep convolutional neural networks that exploits concepts of data reuse and statistics to minimize energy consumption avoiding unnecessary computations and data accesses. Data movement is optimized through the intensive use of buffering to facilitate the temporal reuse of data. Besides this, by clock gating unused PEs and saving partial results to be later restored, configurability ad efficiency are obtained.

Besides purely hardware solutions, also software kernels are proposed to maximize the performance of neural networks while minimizing their memory footprint. In [6] the CMSIS-NN kernels are proposed as a way to achieve improvements in terms of throughput and energy efficiency. Functions that implement the most popular neural network layers, such as convolutions, depthwise convolutions and fully connected, are supported by utility functions including data conversion and activation tables. This way, more complex neural network modules can be built.

1.4: PULP Architecture

The following overview of the PULP platform takes the Darkside chip as a reference. Both the cluster and the MCU subsystem will be illustrated.

1.4.1: SoC Architecture

The SoC is built around the open-source full-RI5CY core ([7], [8]) , a two-pipeline stage RISC-V processor optimized for low power consumption. This core is connected to a low latency interconnect and implements the RV32IMC RISC-V ISA. It also includes an integer 32-bit sequential multiplier and a 32-bit divider. It provides support for the basic RISCV extensions as well as for the XpulpV2 extension, which introduces SIMD instructions for 16- and 8-bits formats, hardware loops and bit manipulation instructions. The SoC also features 16 SRAM banks of 32 KB each, for a total of 512 KB of interleaved memory, a private memory of 64 KB and 8 KB of ROM memory. All these components are connected through a low-latency interconnect [9].

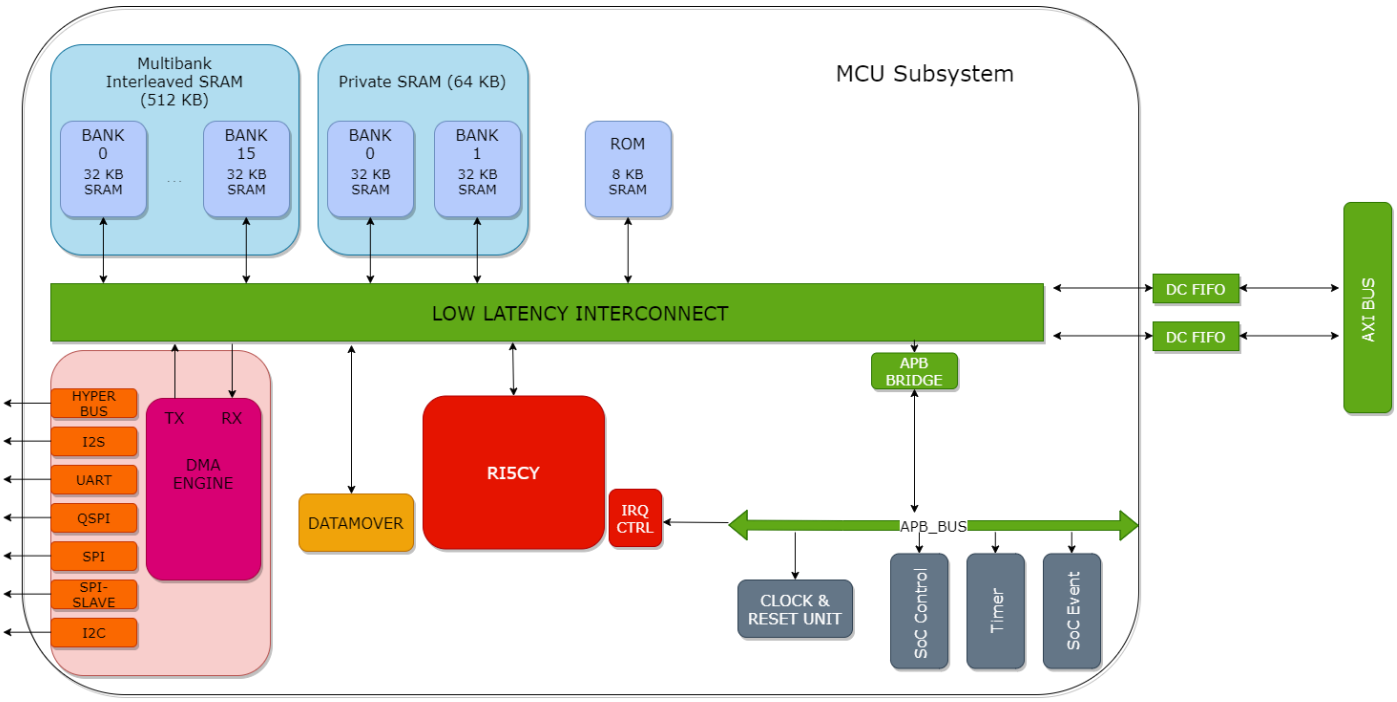

Figure 1: Darkside MCU Subsystem Architecture

A DMA controller [10] is also included in the SoC, as well as all the typical peripherals of an advanced MCU, such as I2S, UART, QSPI, and I2C. Also, an HyperBus peripheral is included to extend the on-chip memory through a DDR interface with 800 Mbit/s of bandwidth. Every one of these peripherals has a dedicated lightweight µDMA channel, which allows to control data transfers from and to the L2 memory. Besides granting a predictable latency related to memory accesses, this also allows multiple and concurrent data transfers towards external devices while operating at low frequency with no need for the peripherals to be connected to large buffers.

Also, a low-cost APB bus provides access to configuration registers, clock and power control, timers and a SoC event generator. As can be seen from Fig. 1, also the Datamover is connected to the low latency interconnect. This was done by writing a wrap of the accelerator in order to bind together the different interfaces used within the SoC and the cluster. The cluster and the SoC are connected by means of an AXI bus . The accelerator is connected to the interconnect by means of a XBAR_TCDM_BUS and an APB_BUS (based on the industrial standard AMBA, as well as for the AXI bus) interfaces.

Regarding the TCDM protocol, it's a simple memory protocol used to interface HWPEs with to L1/L2 external memories. In particular, the L1 (Level 1) shared memory is the closest memory available to the processor and it's meant to be accessed in a quick and easy manner. Thus, it's crucial that the interconnect provides a large bandwidth coupled with a very low latency. While accessing local shared memory can take up to a couple of cycles, accessing a global memory would prove highly inefficient, given the latency in the order of hundreds of cycles [9]. It supports only single outstanding transactions and operates around a two-signals handshake, which is based on two phases: *request* and *response*. Specifically, a valid handshake only happens when both the *req* and the *gnt* signals are asserted.

One of the key aspects to achieve high computing efficiency is the sharing of on-chip memory resources. For this reason, all the functional units share data through the L2 memory, by means of a double-buffering mechanism. This mechanism allows for data transfers from peripherals to L2 and from L2 to L1 to be completely overlapped. Also, the memory hierarchy is organized as a single address space, this way every master in the chip can access all memory locations, leading to an easier overall programmability of the system.

1.4.2: Cluster Architecture

The cluster is built around 8 RISCV-NN cores, 16 TCDM memory banks, a TCDM interconnect and the HWPEs. The cores of the cluster support the RVC32IMF instruction set, plus an extension for energy-efficient digital signal processing (*Xpulp*). This set of instructions includes hardware loops, load/store with post increment and MAC operations. Other instructions include vectorial ones and bit manipulation.

In the following image, the general cluster architecture for a PULP platform is shown.

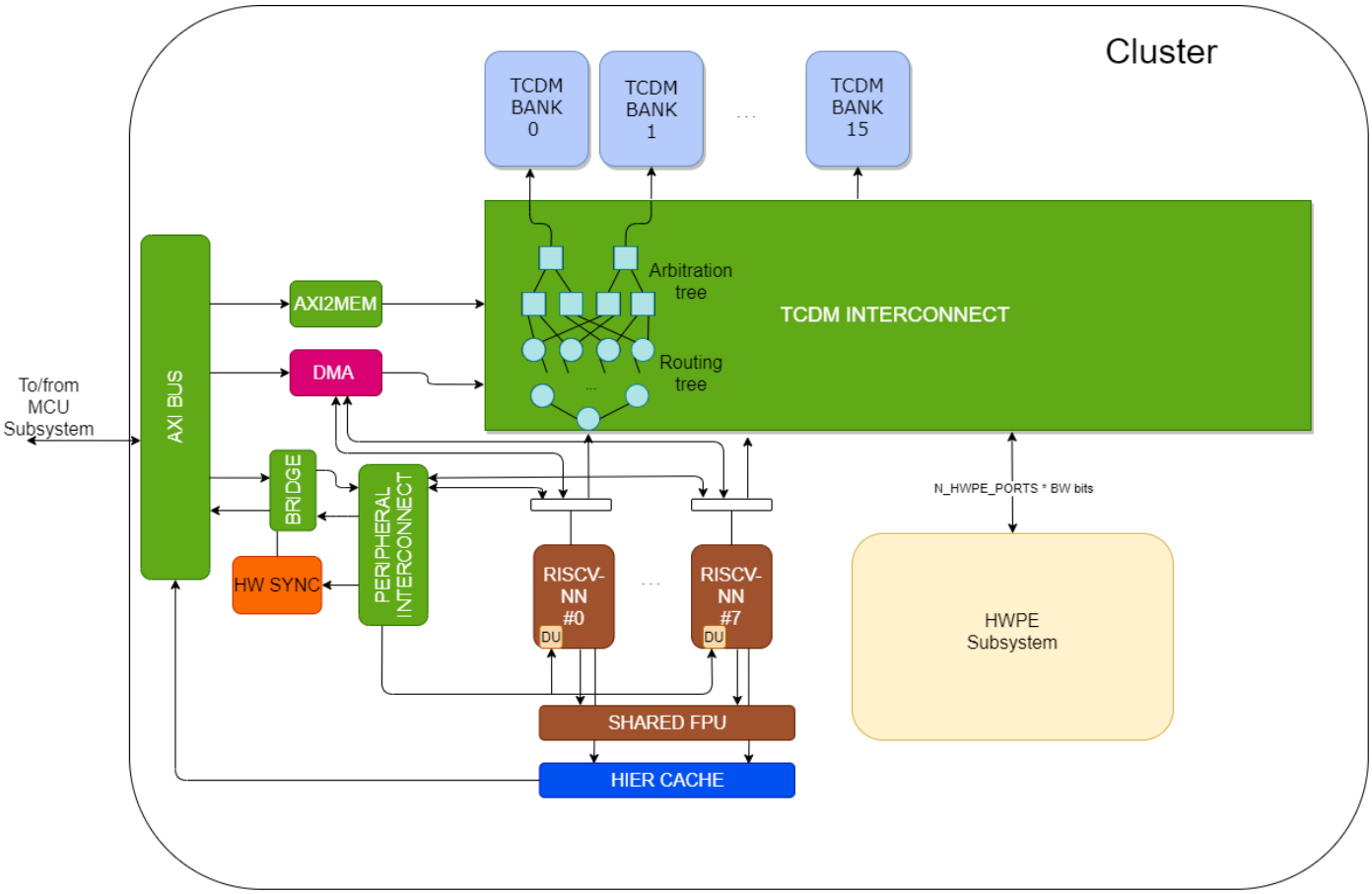

Figure 2: Cluster Architecture

The L1 memory can serve all memory requests in parallel with a single-cycle access latency by means of a low-latency logarithmic interconnect featuring a word-level interleaving scheme.

This TCDM interconnect consists of two layers:

- Routing tree: the routing switches route each packet between the processor side and the memory side. These switches are bidirectional and handle data, handshake signals and addresses.
- Arbitration tree: the arbitration switches use the round-robin algorithm. This way, a starvation-free arbitration mechanism is provided.

Further details about the TCDM interconnect will be given in section 3.1, where the HWPE communication protocols are described.

A dedicated peripheral interconnect is used to access the cluster peripherals such as a timer and the event unit (HW SYNC in Fig. 2) and the AXI bus as well. The lightweight DMA controller [11] supports data movements from L1 memory to L2 memory, and up to 16 outstanding transactions toward the AXI bus, thus hiding the access latency to the L2 memory.

The hierarchical cache [12] is implemented with a latch-based memory, which improves the access energy consumption, but significantly increases the area overhead. To ease this, the cache is shared by the 8 cores, in such a way to avoid instruction replication on the private caches which are typically employed in multi-core systems. Also, to reduce the impact of physical implementation of latch-based memories, the cache is split into four arrays, so that each array has a write port connected to the AXI bus. This multi-port architecture allows the cores to perform non-blocking access to the cache, with performances equal to a private cache.

Additionally, fast event management, parallel thread organization and synchronization are supported by a dedicated hardware event unit enabling low-overhead parallelism to boost performance and energy efficiency of parallel workloads with a fine-grain. This specialized hardware block also controls the clock gating at the top level of each core within the cluster. This allows to minimize the dynamic power consumption of a core waiting for a certain event and resume its execution in just two clock cycles.

Also, both the DMA controller and the event unit have dedicated ports to each core of the cluster in such a way to enable fast and non-blocking accesses. The connection strategy, a 2-level demuxing logic, ensures that accesses to the low-latency interconnect are prioritized over the peripherals.

Since FPUs are expensive in terms of area [1], a shared approach is adopted. This works since the percentage of floating-point operations is rarely greater than 50% in most applications, and FPUs need to be pipelined to match the frequency of the rest of the system.

The following operations are implemented by the FPU:

Unit	Latency	Pipelined/Iterative	# Of Shared Units
ADD	2	Pipelined	1
SUB	2	Pipelined	1
MUL	2	Pipelined	1
MAC	3	Pipelined	2
DIV	4	Iterative	1
SQRT	6	Iterative	1
CAST	2	Pipelined	1

Figure 3: FPU Supported Operations

Also, the shared FPU is integrated into the cluster through an auxiliary interconnect that, featuring a request/grant protocol with round-robin arbitration, allows each processor to access each unit of the FPU and to be stalled whenever the shared resource is used by another CPU.

While more focus will be given to the HWPE subsystem later in the thesis, a general structure is illustrated in the following figure.

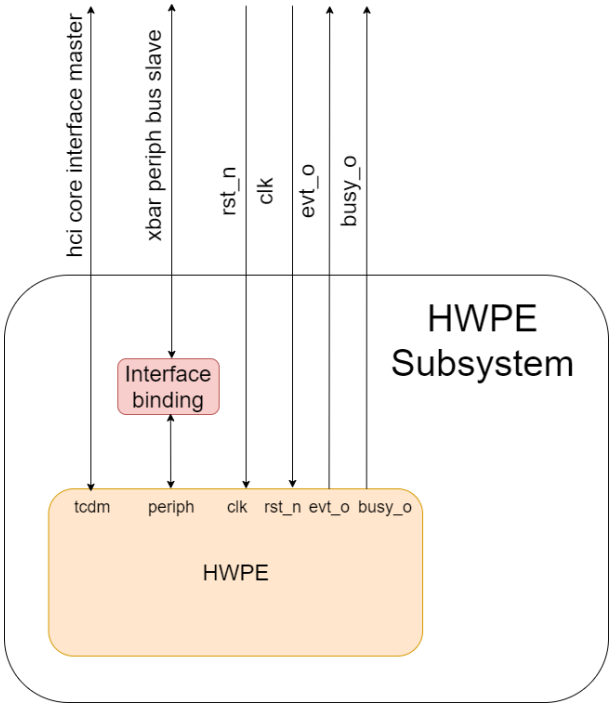

Figure 4: HWPE Subsystem scheme

The HWPEs are instantiated within the HWPE subsystem, and local bindings can be performed to adapt the interfaces of the accelerator. Specifically, two interfaces are needed:

- HCI (*Heterogeneous Cluster Interconnect*) core interface: this interface is used for data communication between the TCDM interconnect and the accelerator.
- Xbar periph bus interface: this interface is used for configuration purposes. Generally, a control module exposing one or more register contexts is instantiated inside the HWPE. Note that this interface requires a local binding inside the subsystem, since the HWPEs use a specialized interface named *hwpe_ctrl_intf_periph.*

These interfaces will be described in detail in section 3.1.

1.4.3: Hardware Accelerators in Darkside

Now, the HWPE subsystem implemented in Darkside will be described in detail, while a general structure has already been shown in Fig. 4.

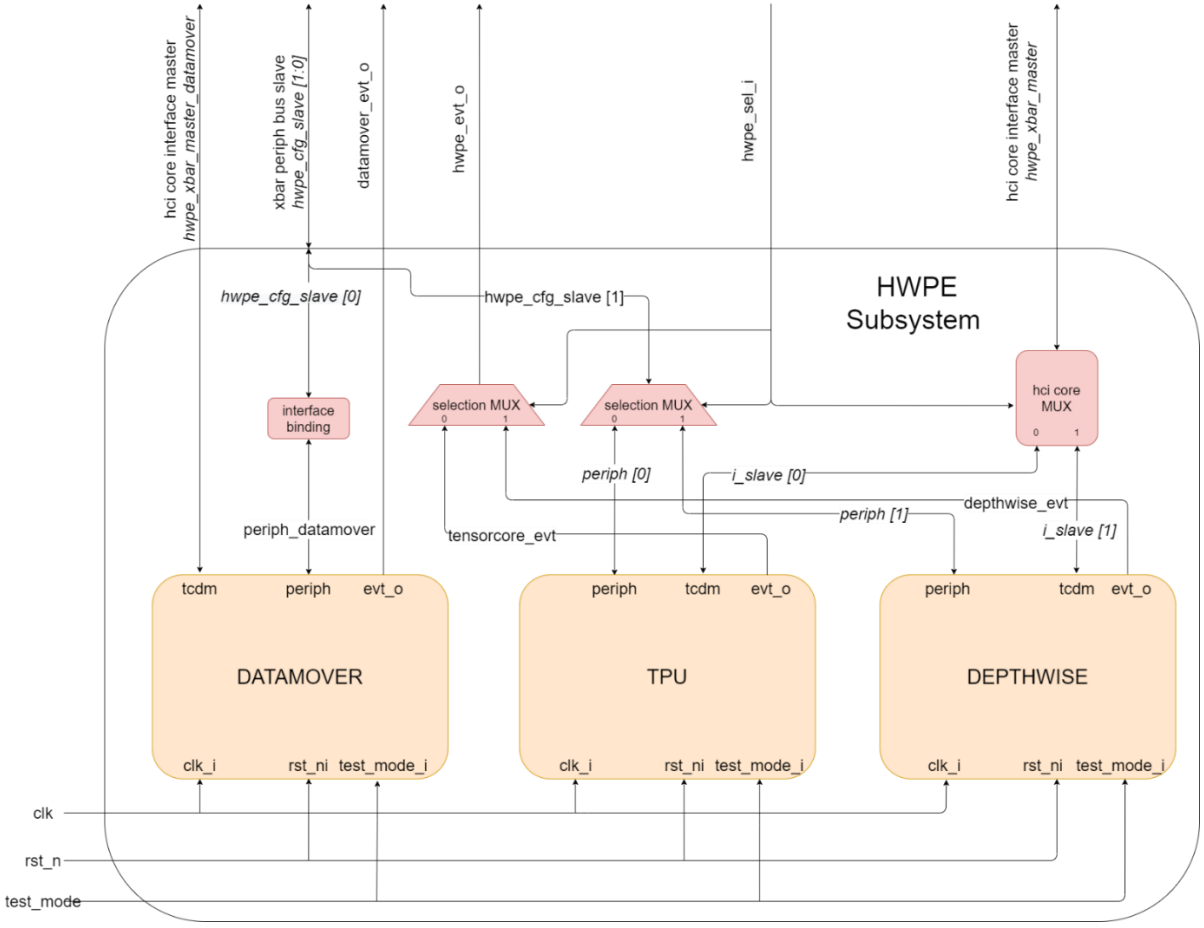

Figure 5: Darkside HWPE Subsystem scheme

Three hardware accelerators have been designed for the Darkside chip:

- Tensor Processing Unit: it performs mixed precision matrix products between floating point operands for neural network training and inference.
- Depthwise Accelerator: it allows to optimize the depthwise convolution of a 3x3 kernel by exploiting re-use of data stored into memory in the HWC format.
- Datamover: it performs reordering operations on tensors stored in the L2 memory of the cluster and the L1 memory of the SoC. The internal buffering mechanism allows these operations to be performed on data widths lower than 32 bits (specifically 8-, 4-, 2- and 1-bit) within each word being transferred. Also, the address generator modules allow to shuffle the elements of the tensor at word-level.

These three IPs are integrated within the PULP cluster and connected to the low-latency interconnect. The interfaces used in the HWPE subsystem are *hci core interfaces* for the tcdm ports and *hwpe ctrl periph* for the configuration ones. While the Datamover is directly connected, the other two accelerators are connected to a mux and can be switched between one another. This holds true for both the data and configuration interfaces, as well for the event signals coming from each accelerator. The selection signal *hwpe_sel_i* comes from a register instantiated in the cluster control unit.

Together, these IPs allow for a lot of flexibility and configurability when performing tasks involving neural networks, from data marshaling to depthwise convolutions and inference. Further in the thesis, the HWPE library modules and interfaces will be briefly discussed, while a bit more focus will be given to the communication protocols.

2: Data Marshaling in Deep Learning-based Applications

Data marshaling is the process of transforming the layout of data stored into memory in a more convenient format for its transmission or elaboration by a DNN [13]. Given the rising importance and diffusion of DNNs on MCUs, one of the crucial tasks in deep learning applications deployment is to ensure that these networks are not redundant. This means that no additional operations should be performed unless the quality of the results is improved. To this end, there are some trends that aim at minimizing the price of DNNs operation while maintaining a good quality of results. Two orthogonal directions can be identified:

- Trying to adapt DNNs for deployment on small devices by shrinking their topologies.
- Introduce quantization operations to minimize the cost-per-operation both in terms of energy and parameters stored into memory.

Another crucial task is to achieve maximum utilization of the computing units once the DNN has been deployed. In this sense, it's vital to reduce performance penalties related to transferring data across the memory hierarchies. In some cases, these penalties can be solved by realizing highly specialized hardware architectures to accelerate specific layers or entire networks, but this can lead to a lack of flexibility. This can be countered by providing a highly optimized software support instead of physical hardware blocks.

2.1: HWC and CHW Data Formats

These are the two most common formats in which image data can be found stored into memory:

- CHW: Channel-Height-Width
- HWC: Height-Width-Channel

Depending on the format, the dimension ordering is the same as that of the data stride. Considering an HWC format, the data along the channel is stored with a stride of 1 element, whereas the data along the width is stored with a stride equal to the channel count. At last, the data stored along the height is stored with a stride equal to $channel\ count \times image\ width$.

Analogously, considering a CHW format, data along the width dimension is stored with a stride of 1, data along the height with a stride equal to the image width. Again, data stored along the channel dimension has a stride of $image\ width \times image\ height$.

For a visual reference, the following image is proposed:

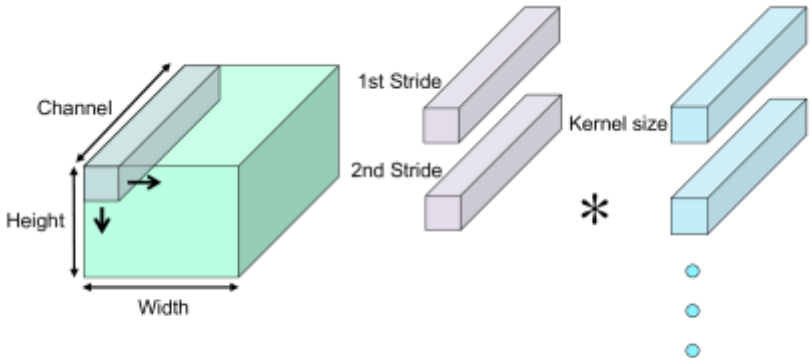

Figure 6: Data storage in 3D [6]

2.2: Impact of Data Marshaling

Thus, one of the main challenges is to efficiently move data across the memory hierarchy and minimize the chance for data traffic to become an application bottleneck. So, data re-use and tiling strategies become essential.

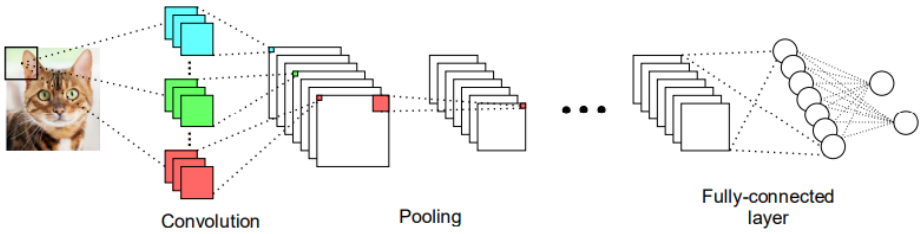

Figure 7: Typical Deep Neural Network Structure [6]

Popular neural network layers are convolution, depthwise separable convolution, fully connected, pooling and activation. Many of these require data reordering operations in memory. For example, a convolution layer extracts a new feature map by computing a dot product between filter weights and a small portion of the input feature map. Typically, a convolution is decomposed into two phases:

- Input reordering: in the form of an *im2col* operation.
- Dot product: to compute the activation values of the output feature map.

This decomposition is necessary to efficiently implement the convolution operation on an MCU-like device.

Specifically, *im2col* is the process of transforming the input into columns that contain the data required by each convolution filter. This allows to load the input features as a contiguous array into memory. An example of the *im2col* operation is shown below:

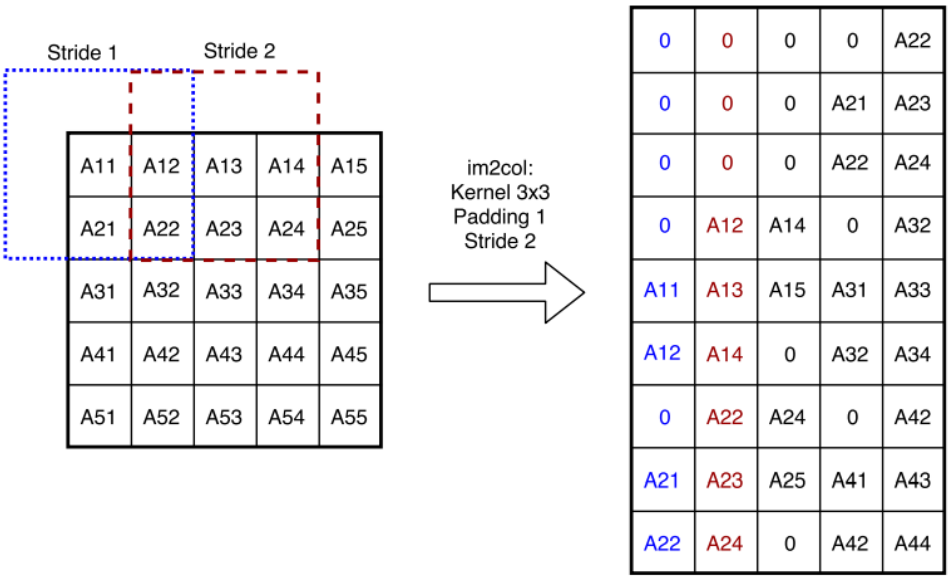

Figure 8: im2col operation [6]

One of the main challenges of the *im2col* operation is the increased memory footprint deriving from repeated pixels of the input image within the column buffers. Indeed, the required memory footprint can be computed as $C \times kw \times kh$. To alleviate this, partial *im2col* operations could be implemented ([6]) to only expand a limited number of columns, sufficient to boost the matrix-multiplication kernels while keeping the memory overhead low. So, the performance of convolution layers is significantly impacted by the layout with which image data is stored into memory.

The following image can be considered for reference:

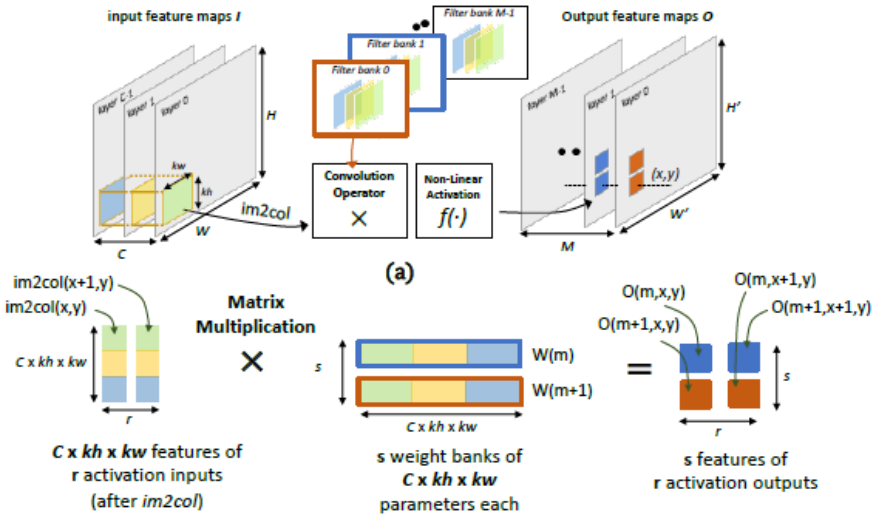

Figure 9: Convolution on 3D data [14]

Then, the values of the output feature map can be computed as:

$$O(m,x,y) = dot\ (W(m), im2col(x,y))$$

Where $W(m)$ is the m-th bank of the weight filter, whereas $im2col(x,y)$ is the unrolled input buffer of length $C \times kw \times kh$. Between the two formats introduced in section 2.1, the most efficient one is HWC. This format allows data for each pixel, which can be identified as a (x,y) coordinates couple for a specific channel, to be stored in a contiguous manner into memory. Data stored in such manner can also be copied efficiently by means of SIMD instructions, thus introducing lower runtime overhead when the *im2col* buffers are built.

In the following image some results gathered in [6] are shown:

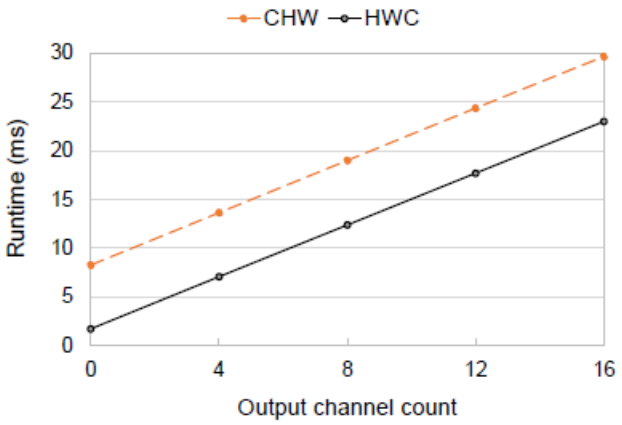

Figure 10: Comparison between CHW and HWC data layout [6]

These results are related to the convolution execution on a Cortex-M7. The HWC input is fixed at 16x16x16 at the beginning and then the number of channels is swept. With the same matrix multiplication performance, the *im2col* runtime is visibly lower for the HWC format.

So, designing a hardware accelerator able to perform conversions between the HWC and CHW formats on image data stored into memory could lead to significant reductions of runtime and increase of performance in relation to convolution layers. Indeed, a lot of attention will be given to the capability of 3D stridden data access and possibility of unpackaging data below the 8-bit data width, which can prove useful when deploying QNNs.

3: The Datamover: An Accelerator for Data Marshaling Operations

The Datamover is a simple accelerator capable of performing stridden accesses into the tcdm memory of the cluster, both to load and store data. This allows to perform a variety of reordering operations along three dimensions simply by configuring a set of registers. Alongside this, an internal buffering mechanism has been implemented in order to perform reordering operations with different granularities within each word of the tensor. This chapter is structured as follows. First of all, the HWPE library modules and communication protocols will be described, followed by an in-depth look at the Datamover internal structure and functionality. Then, both the integrations of the Datamover within the cluster and the SoC will be presented, while focusing on the interfaces and protocols used in each context. Then, the internal buffering mechanism of the Datamover and possible use cases will be illustrated.

3.1: HWPE communication protocols

HWPEs (*Hardware Processing Engines*) are specialized hardware accelerators that can be integrated in the SoC and the cluster of a PULP system to perform specific tasks and help the system to improve its overall efficiency and performance. A general structure of these accelerators is shown in the following image.

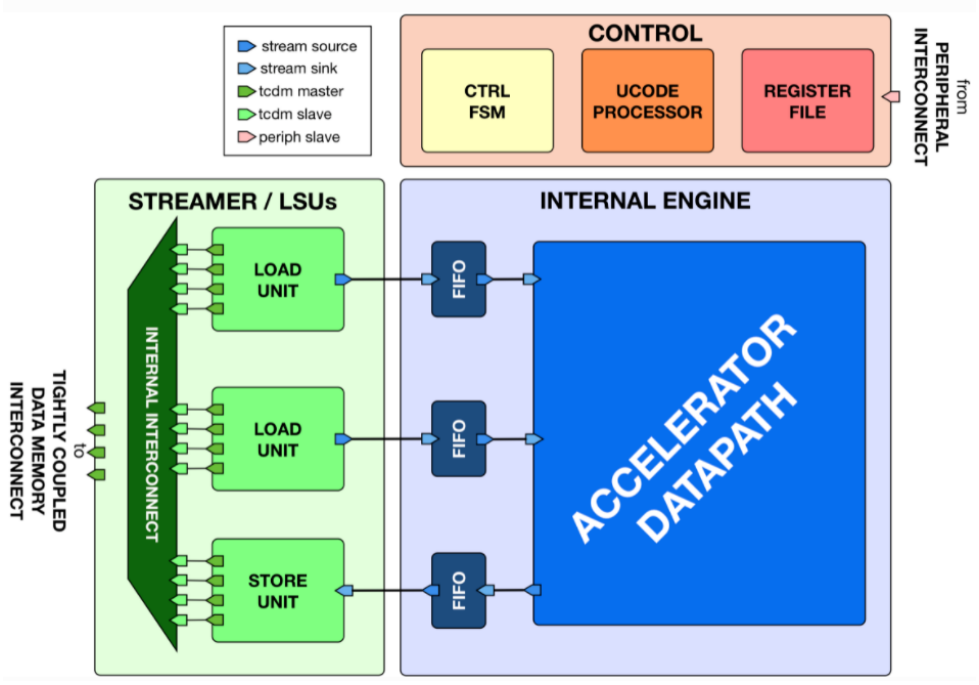

Figure 11: Typical HWPE structure

HWPE-Stream Protocol

First of all, the *HWPE-Stream* protocol is the one used to move data within the sub-components of the accelerator. So, these streams are generated and consumed within the accelerator. I''s possible to use them to cross different clock domains when coupled with a dual-clock FIFO. Particular attention must be put on the handshake procedure, which must occur in a fully synchronous way.

To use this protocol, a *hwpe_stream_intf_stream* must be instantiated. The interface signals and directions can be observed in the following figure.

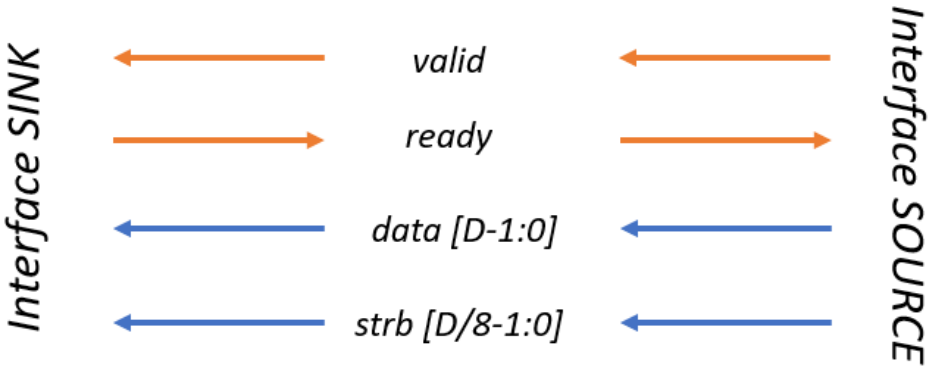

Figure 12: hwpe_stream_intf_stream signals

This interface has two modports: source and sink. Modports are lists of signals in which directions are declared for each signal of the interface. When instantiating a module that makes use of interfaces, the related modports need to be specified.

Referring to Fig. 12, the source modport is the one sending data, strobe and valid signals, whereas the sink port only responds with the ready signal. The data signal obviously contains the data payload, whereas each bit of the strobe signal indicates a meaningful byte of data. The handshake of the protocol relies on the valid and ready signals, both of which are asserted when set to 1. The following set of four rules is to be respected for a valid handshake to happen:

- Both *valid* and *ready* have to be asserted for a valid handshake to happen. Also, after the handshake, the current payload is considered consumed by the sink side.
- The *data* and *strb* signals can change value only when *valid* is de-asserted or in the cycle following a valid handshake.

- The *valid* signal can't depend in a combinational way on the *ready* signal for its assertion. Such a constraint is not required for the *ready* signal.
- The *valid* signal can only be de-asserted after a valid handshake.

By referring to Fig. 11, these are the kind of streams that the streamer and the internal engine of the HWPE use to communicate with each other.

TCDM-HWPE Protocol

The communication between the HWPE and the TCDM interconnect is handled by means of a *hci_core_intf* interface. The involved signals are shown in the following figure.

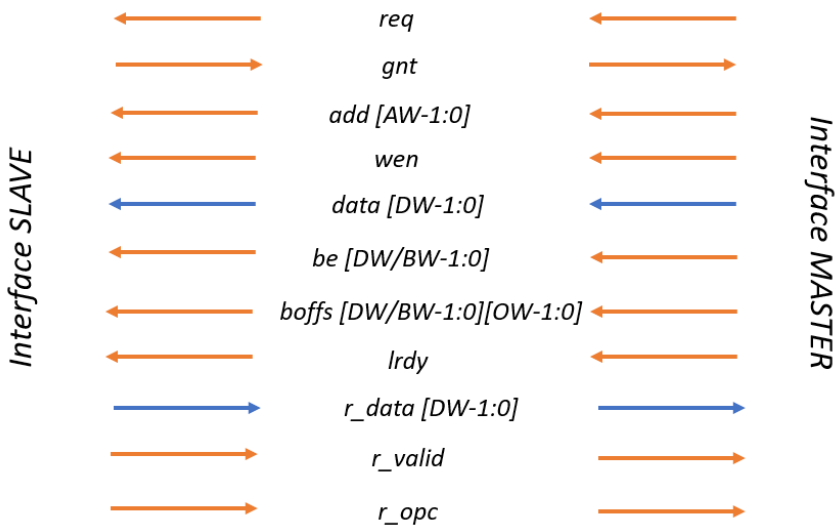

Figure 13: hci_core_intf signals

Note that the HPWE exposes a master modport. The signals meanings are:

- *Req:* the req signal is sent by the master of the transaction to the tcdm interconnect as a way to request access to a shared resource (e.g. a memory bank).
- *Gnt:* this signal is used to handle conflicts between different masters requesting access to the same memory bank within the same cycle. When asserted for a certain master, its request is confirmed, and the data is forwarded. Thus, the handshake procedure takes place. Otherwise, if the request cannot be immediately granted, a stall occurs.

- *Add:* this signal contains the desired address for the read/write access performed by the HWPE.
- *Wen:* this signals discriminates between write accesses (wen=0) and read accesses (wen=1) to memory.
- *Data:* this signal contains the data payload to be written into memory at the specified address.
- *Be:* this signal indicates a valid data byte when set to 1.
- *Boffs:* it specifies the intra-bank offset.
- *Lrdy:* indicates when data being sent from the master is ready for being loaded into memory.
- *R_data:* data being read from the master when a read access into memory is specified.
- *R_valid:* it indicates when valid data is being sent to the master for reading accesses.
- *R_opc:* it specifies the opcode of the current operation. This is not used in the Datamover.

By referring to Fig. 11, this interface that allows the streamer component of the HWPE to communicate with the TCDM interconnect.

The arbitration mechanism between the different masters accessing to the memory is implemented through a logarithmic interconnect, so a binary tree is established for each memory bank master port [15]. This kind of structure is shown in the following figure.

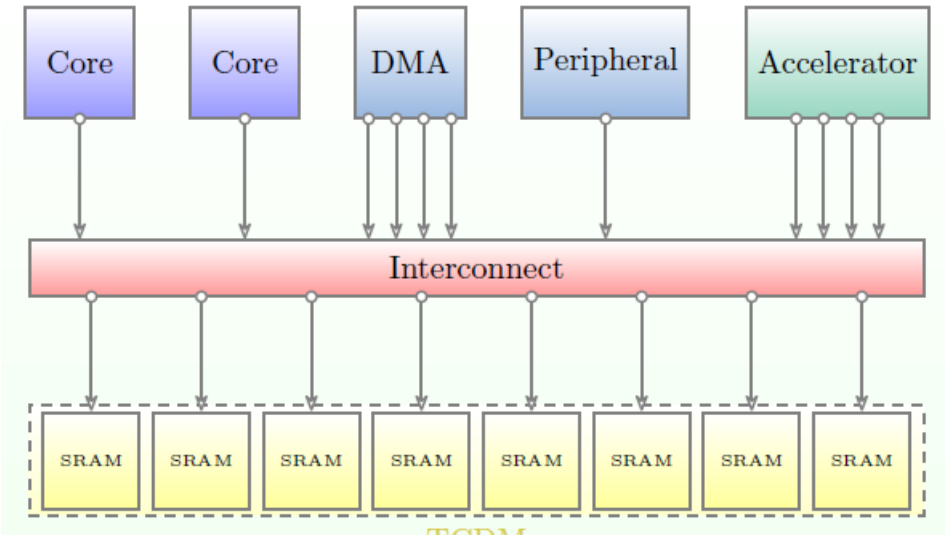

Figure 14: Example of HCI in a cluster [15]

It's quite clear how, by increasing the number of afferent ports to the interconnect, this structure can quickly grow in size and latency. So, for HWPEs exposing a lot of ports towards the interconnect, a separate structure can be implemented: the HWPE interconnect. The resulting internal structure of the interconnect is the following.

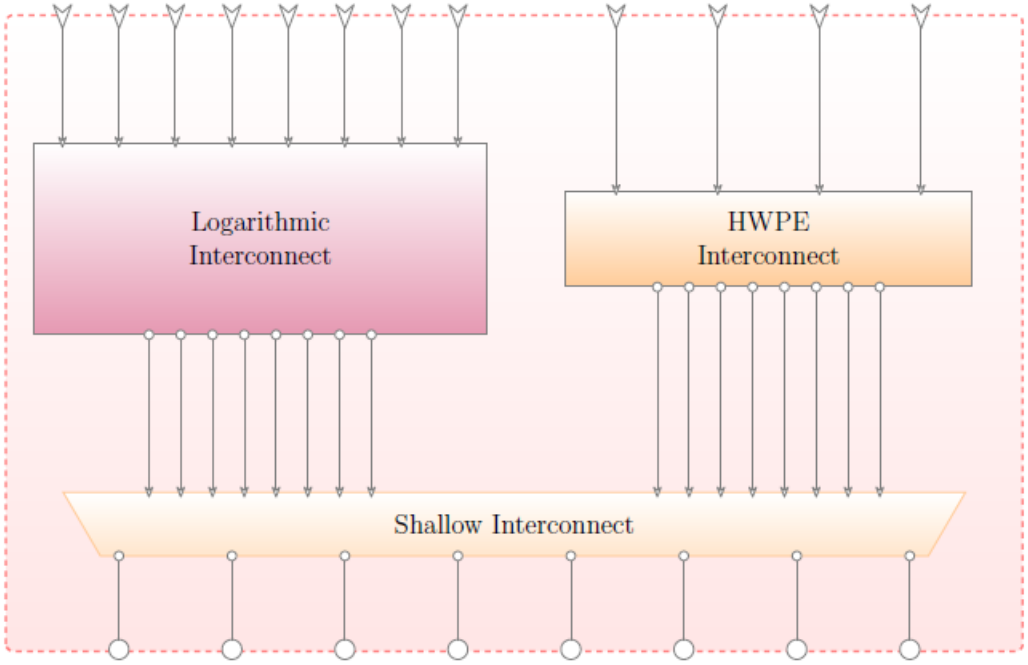

Figure 15: Internal structure of the HCI [15]

This solution allows to avoid the arbitration inherent to the logarithmic interconnect for the HWPEs. The main elements are:

- *HWPE Interconnect:* by adopting a word interleaved memory scheme and allocating consecutive addresses to adjacent memory banks, a spread of consecutive data over multiple banks is obtained. This mechanism allows for a high degree of flexibility when accessing data, since the accelerator will request data from contiguous memory banks, thus eliminating possible conflicts. Also, it's crucial that, at the memory side, all requests from the HWPE are collectively granted or stalled, avoiding situations where only a portion of data required by the accelerator is valid. This derives from merging all the slave ports of the HWPE into a single larger port.

- *Shallow Interconnect:* this block performs multiplexing between the access to the memory banks and the two interconnect structures. This is done by exposing two sets of slave ports, one for each interconnect, and featuring each memory bank with a 1-bit multiplexer that selects between the LIC (*Logarithmic InterConnect*) and the HWPE interconnect. This arbiter block is needed since both the interconnects expose master ports towards the memory banks.

In the following figure, an example of request forwarding towards the HWPE interconnect is shown.

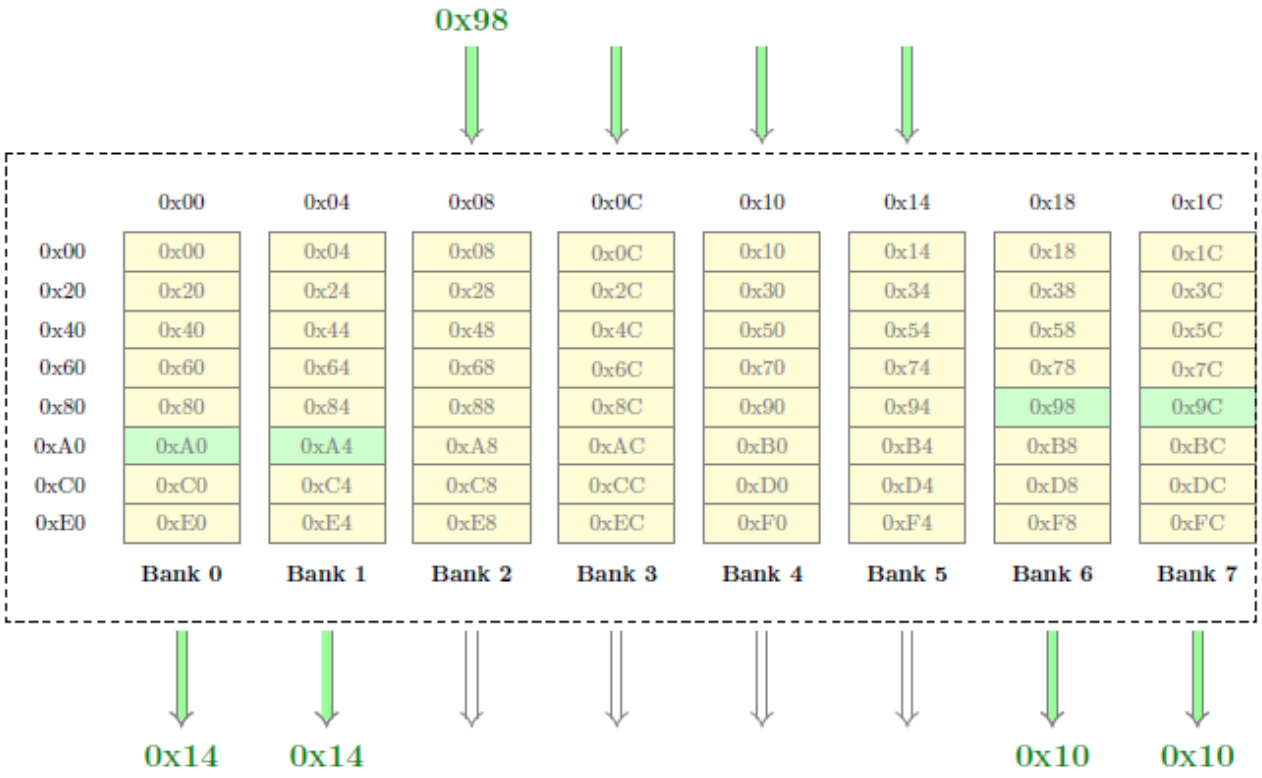

Figure 16: HWPE Interconnect request forwarding [15]

In this case, the HWPE needs to access four double words starting from address 0x98. By arranging data as previously mentioned, four different memory banks need to be accessed in order to satisfy the accelerator request.

This bypass solution is not strictly needed by the Datamover accelerator, since only a single 32-bits wide port is exposed towards the interconnect.

HWPE-Periph Protocol

HWPEs generally expose a control slave port for configuration purposes. This port is connected to the peripheral system interconnect by means of a *hwpe_ctrl_intf_periph*. The interface signals are shown in the following figure.

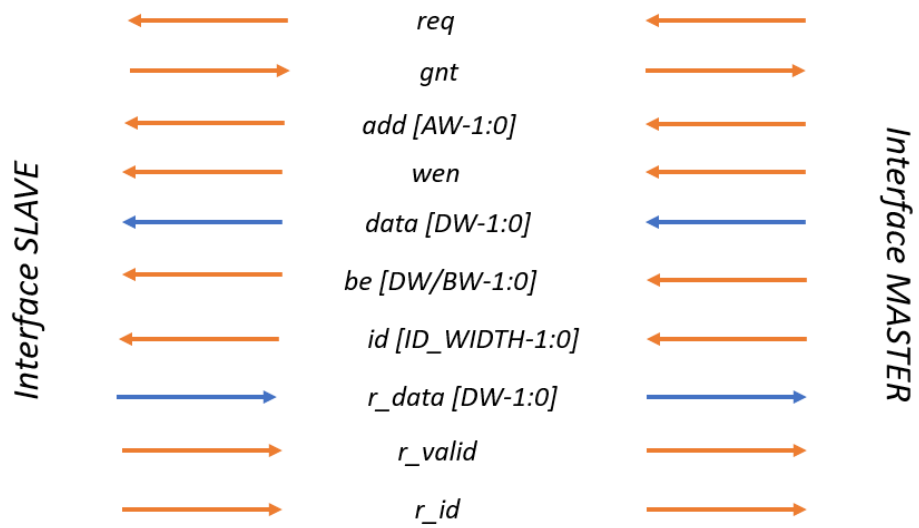

Figure 17: hwpe_ctrl_intf_periph signals

Most of the signals have the same role as in the *hci_core_intf*. The only differences are due to the *id* and *r_id* signals, which are used in load operations. The *id* identifies the master during the request phase, whereas the *r_id* is used during the response phase. This is very useful when data traffic has to be sorted through different masters. By referring to Fig. 11, this interface is used to configure the control sub-component of the HWPE through the peripheral interconnect.

3.2: Internal Structure and Functionality

The description of the Datamover structure begins from the top module. The interface exposed by the top module is the following:

```
module datamover_top #(
  parameter int unsigned ID        = 10,
  parameter int unsigned BW        = 32,
  parameter int unsigned N_CORES   = 8,
  parameter int unsigned N_CONTEXT = 2
) (
  // global signals
  input  logic                    clk_i,
  input  logic                    rst_ni,
  input  logic                    test_mode_i,
  // events
  output logic [N_CORES-1:0][1:0] evt_o,
  // tcdm master ports
  hci_core_intf.master            tcdm,
  // periph slave port
  hwpe_ctrl_intf_periph.slave     periph
);
```

Figure 18: Datamover top module interface

Besides the clock and reset (active low) signals, the interface includes a test mode signal, an array of events and the two interfaces mentioned in the previous section. The interface named *tcdm* is the one used for communicating with the tcdm interconnect, whereas *periph* is the one used for configuration of the control unit within the accelerator.

Within the top module, the following are instantiated:

- *datamover_streamer:* this modules receives the input stream from the tcdm memory and handles the internal data streams.
- *datamover_engine:* for the Datamover the engine is just a FIFO that takes the stream coming from the stream merger and sends it back to the streamer module.
- *hwpe_ctrl_slave:* this is the module used to configure the accelerator. It features one context of 13 configuration registers that can be accessed via-software during testing.
- *datamover_buffering:* this is the buffering module used to unpack the incoming data according to the width specified during configuration.

- *stream_splitter:* this module generates a variable number of data streams depending on the chosen buffering factor.
- *stream_merger:* this module recreates a single stream after the buffering modules. The conjoined functionality of this module and the buffers is what allows the reordering operations on the data being streamed.

Datamover streamer

The streamer module exposes the following interface:

```
module datamover_streamer #(
  parameter int unsigned TCDM_FIFO_DEPTH = 2,
  parameter int unsigned BW = 32
) (
  // global signals
  input  logic                    clk_i,
  input  logic                    rst_ni,
  input  logic                    test_mode_i,
  // local enable & clear
  input  logic                    enable_i,
  input  logic                    clear_i,
  // input data stream + handshake
  hwpe_stream_intf_stream.source data_in,
  // output data stream + handshake
  hwpe_stream_intf_stream.sink   data_out,
  // TCDM ports
  hci_core_intf.master           tcdm,
  // control channel
  input  ctrl_streamer_t         ctrl_i,
  output flags_streamer_t        flags_o
);
```

Figure 19: datamover_streamer interface

The internal streams of the accelerator are exposed by the streamer interface. These are handled by means of two *hwpe_stream_intf_stream* interfaces, which are declared within the top module. Also, the tcdm connection is passed to the streamer in order to generate the addresses to which data is accessed.

The data streams are:

- *data_in*: this is the stream that contains the data read from the tcdm memory and that goes into the datamover engine. Indeed, the interface modport at the streamer top level is the source one, meaning that data is sent as an output from the streamer.
- *data_out*: analogously to data_in, this is the stream that contains the data that will be written into the tcdm memory and that's sent back to the streamer from the datamover engine.

So, it's crucial that the tcdm interface is passed to the streamer in order to generate the two internal data streams. This is done by using modules of the HCI (*Heterogeneous Cluster Interconnect*) library. First of all, the *tcdm* interface is passed to the *hci_core_r_valid_filter* module, which filters out r_valid signals that may be generated by the cluster even when the access to the tcdm memory is a write one. By means of an interface binding, the master interface *tcdm* is connected to a slave interface of the same type, named *tcdm_prefilter*.

Then, this slave interface is connected to two different modules, depending on the value of the TCDM_FIFO_DEPTH parameter. When greater than zero, two modules are instantiated by means of a generate statement:

- *hci_core_fifo:* this module exposes two *hci_core_intf* modports. The master modport is connected to the *tcdm_prefilter* interface, whereas the slave one is connected to another interface named *tcdm_prefifo*. This one is then passed to the *hci_core_load_store_mixer* module.
- *hci_core_load_store_mixer:* this module exposes a *hci_core_intf* master modport (*tcdm_prefifo*) which is then bound to two different slave *hci_core_intf* modports (*virt_tcdm[0]* for load operations and *virt_tcdm[1]* for store operations) depending on the nature of the request.

Otherwise, when the parameter TCDM_FIFO_DEPTH is equal or lower than zero, only a *hci_core_mux_dynamic* is instantiated. Without using the mixer, the *tcdm_prefilter* interface is bound to the two *virt_tcdm* interfaces. This conditional statement on the fifo depth parameter is necessary since it's not possible to perform TCDM muxing before a FIFO, given there is no standard procedure to couple a response with the channel that requested it. Now, the two interfaces of the array *virt_tcdm* are connected to the two main modules of the streamer, the *hci_core_sink* and the *hci_core_source*.

Hci core sink & Hci core source

These two modules are responsible for generating the two internal streams of the accelerator and the related addresses at which data are read/written. This depends on the *hci_core_intf* modport that each module exposed by each module. The only difference between these two modules resides in the exposed modport of the stream interface: the *hci_core_source* exposes a source modport, whereas the *hci_core_sink* exposes a sink modport.

Inside of both modules, the addresses at which data is accessed are generated by a library module, the *hwpe_stream_addressgen_v3*. This module allows to perform 3D stridden access to into memory once it has been correctly configured.

The address generator module can be programmed with stride and length parameters along three dimensions: d0 (width), d1 (length) and d2 (channel). Also, a total number of elements to be transferred must be specified in order to correctly generate the required addresses. These parameters are written in a set of registers that can be accessed during testing and are assigned within the top module of the accelerator. The functionality of the address generator is based on a counter realized with three conditional assignments. The first condition checks the address counting on d0. If the current length is less than the specified length, the stride along d0 is added to the address. So, until the condition on d0_len gets verified, the address is increased along the dimension d0. Once that condition is no longer verified, the check on d1_len is performed, with the address getting increased by the stride specified along d1. The next increase will again be along the dimension d0, while the next increase along d1 will happen only when d0_len is reached again.

By imagining a 3D tensor, this can be seen as swiping the elements of the first row until d0_len is reached. If configured correctly, the d1_stride parameter will be such that the next accessed element is the first of the second row and so on. When the last row of this face of the tensor is completed, the address is increased by d2_stride. This stride will be configured in such a way that the next accessed element will be the first element of the next face of the tensor, at this point the behaviour of the address generator repeats itself until the total number of transfers is reached. The illustrated behaviour is realized by the code in Fig. 20.

```
if(overall_counter_q < ctrl_i.tot_len) begin
        addr_valid_d = 1'b1;
        if((d0_counter_q < ctrl_i.d0_len) || (ctrl_i.dim_enable_1h[0] == 1'b0))
begin
          d0_addr_d    = d0_addr_q + d0_stride;
          d0_counter_d = d0_counter_q + 1;
        end
        else if ((d1_counter_q < ctrl_i.d1_len) || (ctrl_i.dim_enable_1h[1] ==
1'b0)) begin
          d0_addr_d    = '0;
          d1_addr_d    = d1_addr_q + d1_stride;
          d0_counter_d = 1;
          d1_counter_d = d1_counter_q + 1;
        end
        else begin
          d0_addr_d    = '0;
          d1_addr_d    = '0;
          d2_addr_d    = d2_addr_q + d2_stride;
          d0_counter_d = 1;
          d1_counter_d = 1;
          d2_counter_d = d2_counter_q + 1;
        end
        overall_counter_d = overall_counter_q + 1;
      end
```

Figure 20: Address generation code

Also, enable signals are available for disabling the address counting along a certain dimension. Then, the address is sent out of the address generator module by means of a HWPE stream. This is done within both the *hci_core_source* and the *hci_core_sink*, thus allowing to generate 3D strided addresses for read and write accesses to the tcdm memory.

Also, the hci core modules can perform data realignment by adding 32 additional bits to the width of the tcdm port. With the bandwidth of the datamover being only 32 bits wide, this feature is excluded with the insertion of a parameter.

Datamover engine

The engine within the Datamover is quite simple, being just a *hwpe_stream_fifo* module that links the incoming data stream into the outgoing one. So, the only elaborations performed by the datamover are the reordering of words along three dimensions, the transposition of data with sub-word widths and mapping the input address range into an output one.

Hwpe_ctrl_slave

The slave module is the one exposing the *hwpe_ctrl_intf_periph* interface for configuration. This module provides one context of 11 registers needed to configure the accelerator. These registers are used to store the parameters of the address generator modules, so the lengths, strides and the base addresses of the read interval and write interval in the tcdm memory.

The Datamover also features a configurable buffer factor, which is specified in the configuration register [2] as the exponent of a power of 2. This feature allows to choose at which sub-byte granularity perform the additional transposition for each word being transferred. The possible values are:

- 0: the incoming 32-bits of data are kept as a single stream of 32-bits. Basically, this implements a simple transfer of the tensor data between the two specified address ranges.
- 1: the incoming 32-bits of data are split into 2 parallel streams of 16-bits each.
- 2: the incoming 32-bits of data are split into 4 parallel streams of 8-bits each.
- 3: the incoming 32-bits of data are split into 8 parallel streams of 4-bits each.
- 4: the incoming 32-bits of data are split into 16 parallel streams of 2-bits each.
- 5: the incoming 32-bits of data are split into 32 parallel streams of 1-bit each.

The buffer factor specifies the number of elements within each 32-bits buffer. The configuration register in which this option is implemented is mapped as follows:

$ADDR$	$Unused$	$Buffer\ factor$	$Total\ length$
$0x08$	$31 \div 30$	$29 \div 24$	$23 \div 0$

Figure 21: HWPE reg[2] mapping

Datamover FSM

The FSM (*Finite State Machine*) of the Datamover has four states:

- DM_IDLE: this state is reached when the active low reset signal is asserted. When *slave_flags.start* is asserted, the next state is set to DM_STARTING.
- DM_STARTING: this state is simply a passage to the DM_WORKING state. Also, the flags for requesting the hci core modules to start their operations are asserted.
- DM_WORKING: in this state, a check is performed on the streamer flags and the FIFO flag that signals when its empty. This way, whenever one between the *done* and *ready_start* gets asserted for each stream and the FIFO is empty, the next state is set to DM_FINISHED.
- DM_FINISHED: in this state, the next state is simply set to DM_IDLE and the flag that signals the end of operation for the slave module is asserted.

3.3: Internal Buffering Mechanism

Now the internal buffering mechanism of the Datamover will be described. This allows to perform transpositions at word-level on data widths spanning from 32 bits down to 1 bit. The general architecture is shown in the figure below.

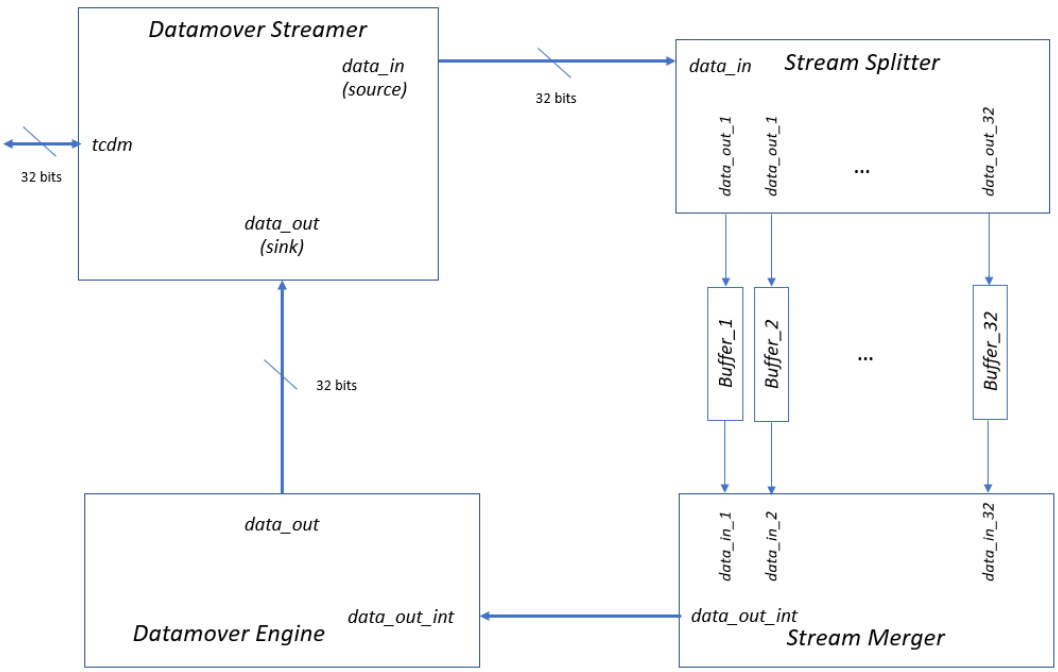

Figure 22: Datamover internal structure

Of course, the number of interfaces and buffers within the architecture is fixed:

- 32 buffers each wide 32 bits are instantiated between the stream splitter and the stream merger.
- 32 interfaces, each with a maximum data width of 8 bits, connect the stream splitter to the buffer array.
- Again, 32 interfaces, each with a data width of 32 bits, connect the buffer array to the stream merger.

Depending on the chosen buffer factor, the number of interfaces (and percentage of the data width) and buffers that are used change accordingly.

Stream Splitter module

In this case, the 32 bits coming from the streamer into the stream splitter are divided into 4 streams of 8 bits each. Of course, also the other signals of the stream interfaces are handled. The valid signal coming from the streamer is assigned to the four valid signals sent to the buffers, whereas the ready signal going to the streamer is obtained by means of an AND of the ready signals coming from the buffers. Also, the strobe signals are passed to the downstream signals with simple assignments. The stream splitter is structured as follows.

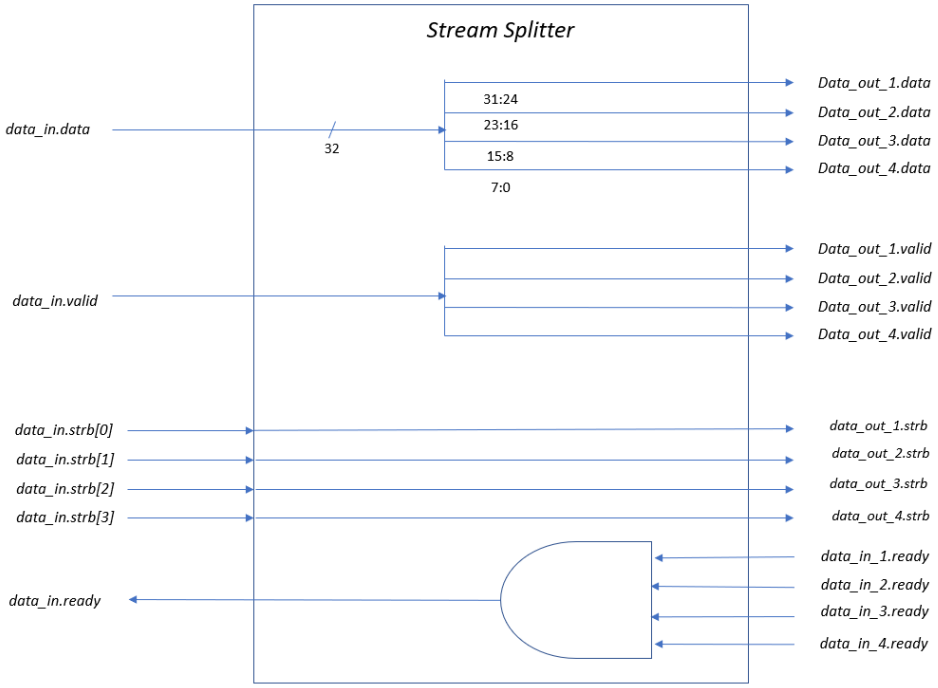

Figure 23: Stream Splitter structure

The four streams coming out of the splitter are then sent into four buffers of 32 bits each, which require, under the current hypothesis, four clock cycles to be filled. In the following section, the buffering module is described.

Buffer modules

Analogously to the other sub-components of the accelerator, the buffering module exposes two *hwpe_stream_intf_stream* interfaces to handle the internal data streams. Also, by acting on the internal data streams rather than interfacing with the tcdm all address-related information is handled by the address generator modules, thus reducing the number of signals and allowing to handle the streams through the use of simple counters.

```
module datamover_buffering #(
        parameter int unsigned BANDWIDTH     = 32
) (
        // global signals
        input logic [23:0]                              tot_len,
        input logic [3:0]                               buffer_factor,
        input logic                                     clk_i,
        input logic                                     rst_ni,
        input logic                                     enable_i,
        input logic                                     clear_i,

        input logic [5:0]                               buffer_count_control,
        input logic [23:0]                              buffer_tot_len_control,

        // Stream interface for data coming from the Datamover Streamer
        // into the Datamover Buffering
        hwpe_stream_intf_stream.sink            data_in,

        // Hci_core_intf slave interface to connect the Datamover Buffering
        // with the Datamover Engine
        hwpe_stream_intf_stream.source          data_out_int,
        output logic                                                    buffer_full
);
```

Figure 24: Datamover buffering module interface

This module is essentially a shift register controlled by a Mealy FSM made of four states which will be described now:

- IDLE: this state is reached when the reset signal is asserted or when the transfer procedure has ended. In the IDLE state a check on the enable signal is performed. When the enable signal is asserted, the next state is set as SHIFT or FULL if valid data is found as input, otherwise the next state is set as IDLE. Note that only when the chosen buffer factor is equal to 0, the FULL state can be reached directly from the IDLE state.
- SHIFT: again, a check on valid data is performed. If that's the case, the content of the register is shifted until it's been filled. Then, the next state is set as FULL. Note that, whenever setting the FULL state as the next state, the counter for the total number of transferred words is enabled. Otherwise, if the buffer is yet to be filled or if no valid data is given as input, the next state remains equal to SHIFT.
- FULL: first of all, the output valid and strobe signals are asserted. Then, a check is performed on the *data_out_int.ready* signal. If output data is ready to be sent out, the buffer content is maintained. This is necessary to perform the sub-word transposition operations. Indeed, the counter which keeps track of the elements within the buffer is enabled and, depending on the chosen buffer factor, the content of the buffer is kept constant for a certain number of clock cycles. It's also important to notice that the *buffer_full* signal is asserted only when *data_out_int.ready* is asserted as well. This avoids having a valid buffer content for more cycles than those needed. When the specified number of cycles has passed, a check on the current number of transferred words it's performed. If the last transfer has been reached, the next state is set as IDLE, otherwise valid data is checked again at the input. In the case of a buffer factor equal to zero, there is no need to go back to the SHIFT state, since only a single element needs to be shifted into the buffer. This mechanism though, requires that the counter that keeps track of the cycles spent in the FULL state is set back to zero. This is done by adding the additional FULL_RESET state.
- FULL_RESET: in this state, the only operation is the assertion of the *reset_full* signal. The next state is always set as FULL.

The FSM diagram can be found below.

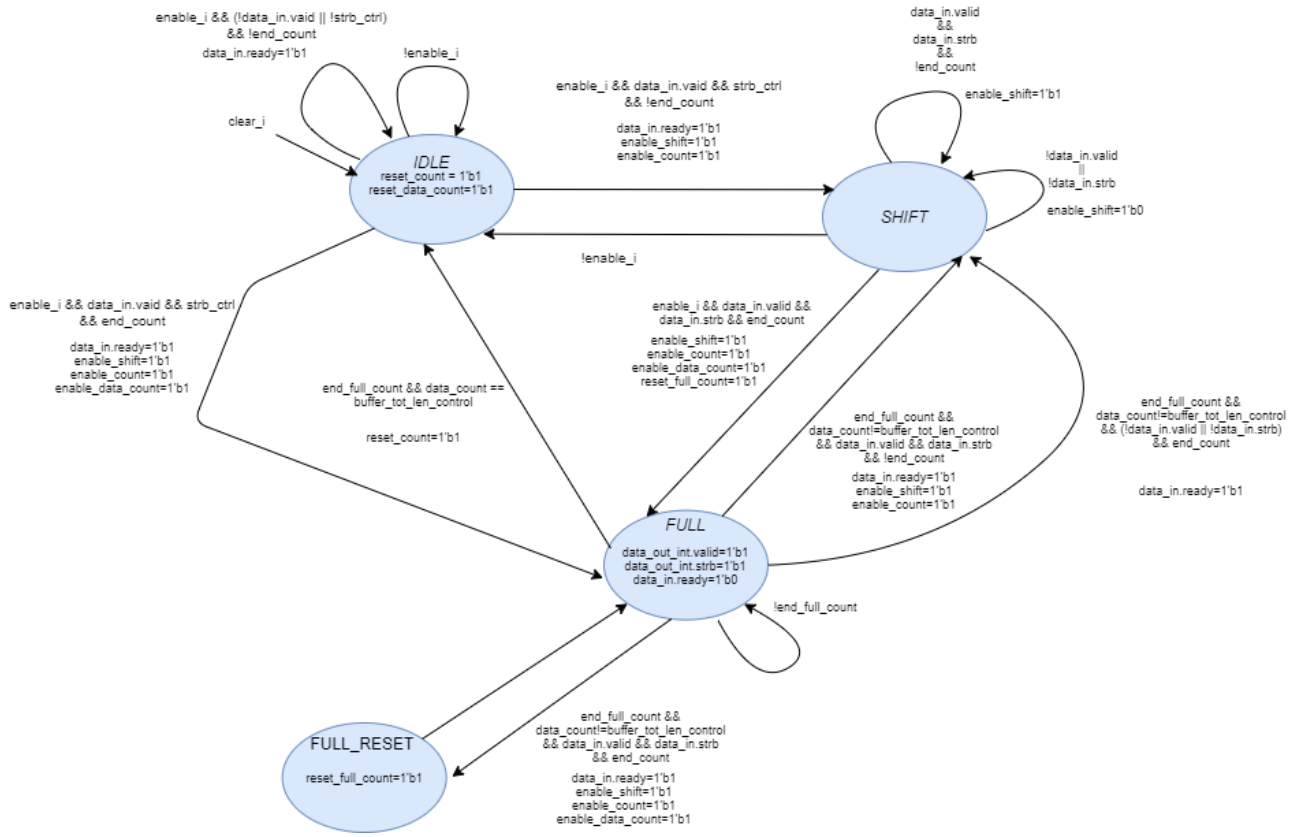

Figure 25: Buffering module FSM

It's crucial to point out that the buffering module is parametrized on the chosen buffer factor. Referring to the example of a buffer factor equal to 4, the shift register will fill with up to four 8-bits data. This means that only four out of the total 32 buffering modules will be used in this particular case. Also, transposition at sub-word level will only be possible with a 8-bit data width.

Stream Merger module

The streams coming from the buffers are then sent to the stream merger module. This module basically performs a transposition of the current content of the buffers. To implement this behaviour, a counter is necessary to keep constant che words stored in the buffers until the transposition is completed. The following figure refers to the buffer_factor=4 use case.

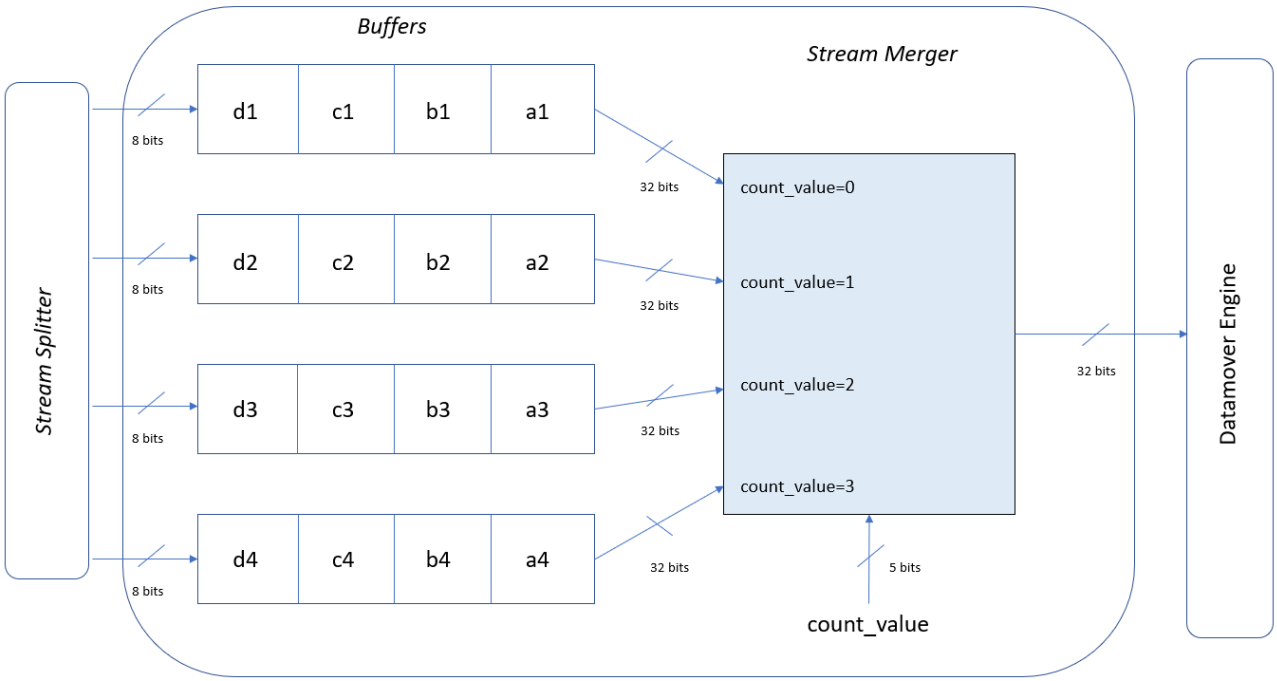

Figure 26: Stream Merger and Buffer modules

Then, the four 8 bits-wide streams coming from the splitter are stored into the 32-bits buffers. Each buffer is then sent to the engine depending on the value of the counter signal. This allows to transpose the matrix composed by the words stored into the four buffers. Each column of the matrix is loaded by the splitter module, whereas each row of the matrix is sent to the engine one at a time.

The counter is enabled by the *buffer_full_int* signal, which is obtained as an AND of the related signals coming from the four buffers. This way, even if only one of the buffers is not full the execution is halted.

Of course, all the other signals of the interfaces are also handled within the stream merger. The valid signal that's sent to the engine is the one related to the buffer being selected as a row of the buffer matrix. The ready signals being sent to the buffers are instead all equal to the one being sent by the engine downstream. The strobe signals instead are assigned to the related interface each time a row is selected and sent to the engine of the Datamover. This handling of the interface signals is illustrated in the figure below.

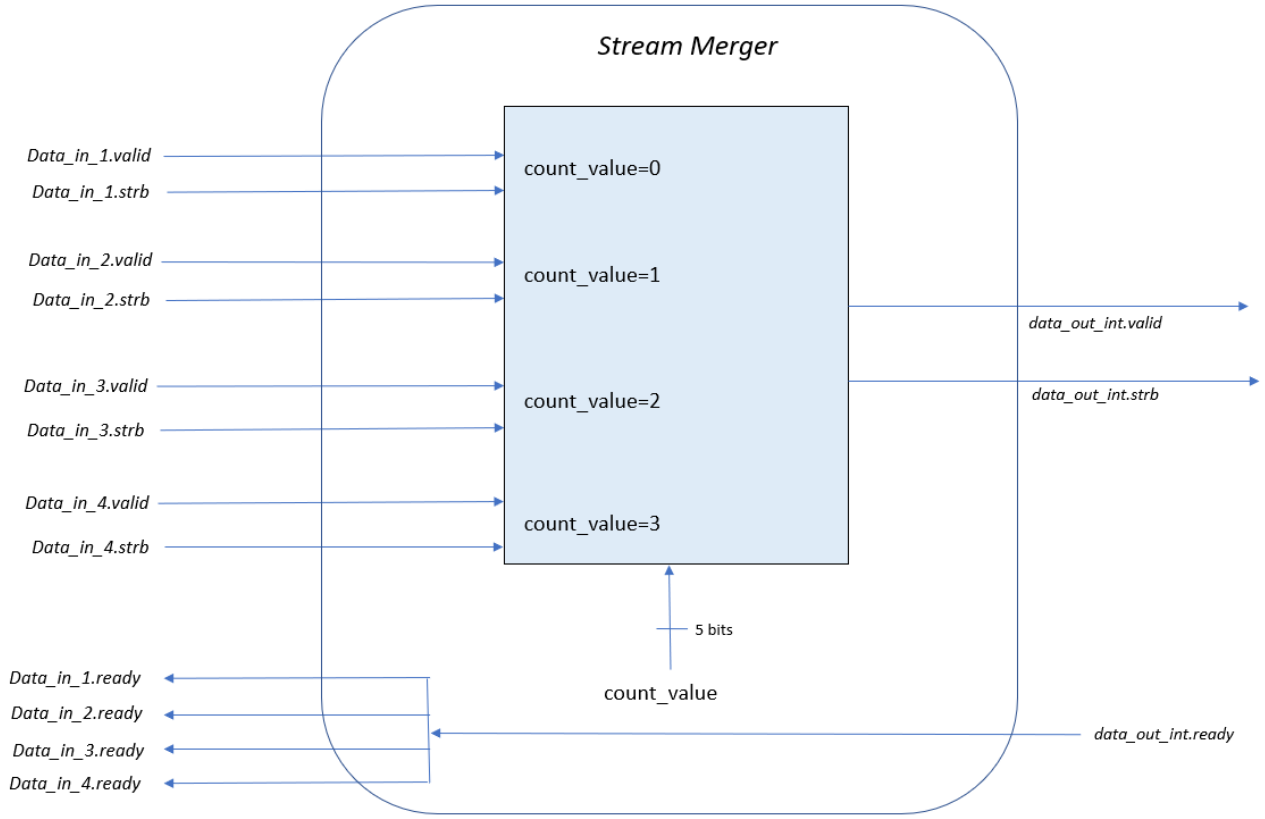

Figure 27: Interface handling in the Stream Merger

The 32-bits wide stream coming from the merger is then sent to the engine and back into the streamer. Then, it will be written into the tcdm memory according to the addresses elaborated by the address generator module.

3.4: Cluster Integration

Within the PULP cluster, three different accelerators have been integrated. These are, besides the Datamover, a Depthwise accelerator and a Tensor Processing Unit. All of these IPs are instantiated within the HWPE subsystem but are connected to the cluster interconnect in different ways. The Datamover is directly connected to the tcdm interconnect, whereas the Depthwise accelerator and the Tensor Processing Unit are muxed between each other.

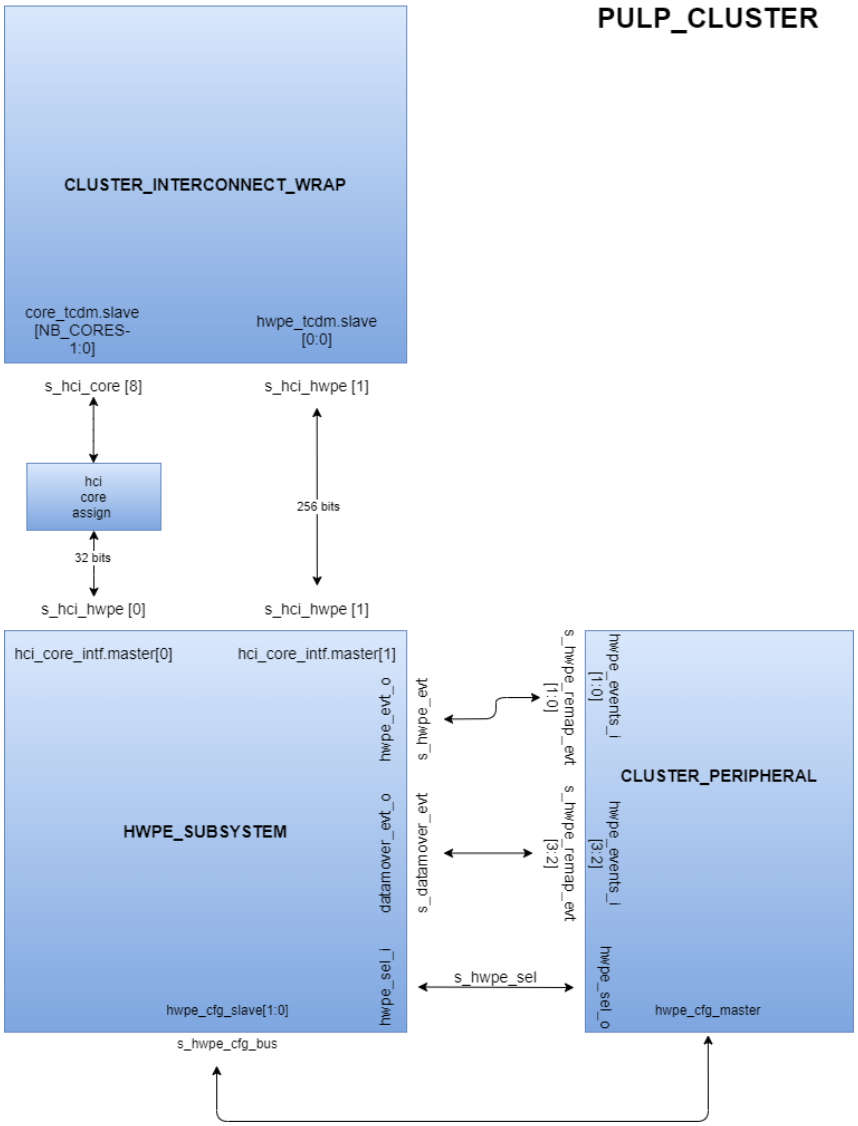

Figure 28: HWPE connections inside PULP cluster

As can be seen in Fig. 28, the HWPE subsystem is connected to the cluster interconnect wrap with two interfaces. These are:

- *s_hci_hwpe[1]*: this is the interface dedicated to the TPU and the Depthwise accelerator. It's connected to the slave port *hwpe_tcdm_slave[0]* of the cluster interconnect wrap.
- *s_hci_hwpe[0]*: this is the interface with which the Datamover is connected to the cluster interconnect wrap. Note that the Datamover gets connected as a master to the cluster interconnect wrap, so in a similar way to the cores of the cluster. Indeed, an additional *s_hci_core* interface has been declared for this purpose. The binding between the slave interface (*s_hci_hwpe[0]*) and the master (*s_hci_core[8]*) is performed by means of a hci core assign.

Also, a selection signal has been implemented to choose which accelerator to use between the depthwise and the TPU. This signal has been implemented by using the 13th bit of the cluster control unit register, which was previously unused. Then, the selection bit gets passed to the hwpe subsystem where is then passed to the mux module. When asserted to 1, the Depthwise accelerator is selected, otherwise the TPU is selected.

Of course, also the configuration ports of the accelerators must be considered. These interfaces are declared as an array of *XBAR_PERIPH_BUS* slave interfaces named *hwpe_cfg_slave* within the HWPE subsystem. The interface of index 0 is reserved to the Datamover, whereas the one of index 1 is muxed between the other two IPs. This array of interfaces is then passed to the cluster peripheral module, as can be seen in Fig. 29, where the two interfaces are then bounded to the *speriph_slave* interface array.

```
// TPU and Depthwise assign
     assign speriph_slave[SPER_HWPE_ID].gnt     = hwpe_cfg_master[1].gnt;
     assign speriph_slave[SPER_HWPE_ID].r_rdata = hwpe_cfg_master[1].r_rdata;
     assign speriph_slave[SPER_HWPE_ID].r_opc   = hwpe_cfg_master[1].r_opc;
     assign speriph_slave[SPER_HWPE_ID].r_id    = hwpe_cfg_master[1].r_id;
     assign speriph_slave[SPER_HWPE_ID].r_valid = hwpe_cfg_master[1].r_valid;

     assign hwpe_cfg_master[1].req   = speriph_slave[SPER_HWPE_ID].req;
     assign hwpe_cfg_master[1].add   = speriph_slave[SPER_HWPE_ID].add;
     assign hwpe_cfg_master[1].wen   = speriph_slave[SPER_HWPE_ID].wen;
     assign hwpe_cfg_master[1].wdata = speriph_slave[SPER_HWPE_ID].wdata;
     assign hwpe_cfg_master[1].be    = speriph_slave[SPER_HWPE_ID].be;
     assign hwpe_cfg_master[1].id    = speriph_slave[SPER_HWPE_ID].id;

// Datamover assign
    assign speriph_slave[SPER_DATAMOVER_ID].gnt     = hwpe_cfg_master[0].gnt;
    assign speriph_slave[SPER_DATAMOVER_ID].r_rdata = hwpe_cfg_master[0].r_rdata;
    assign speriph_slave[SPER_DATAMOVER_ID].r_opc   = hwpe_cfg_master[0].r_opc;
    assign speriph_slave[SPER_DATAMOVER_ID].r_id    = hwpe_cfg_master[0].r_id;
    assign speriph_slave[SPER_DATAMOVER_ID].r_valid = hwpe_cfg_master[0].r_valid;

    assign hwpe_cfg_master[0].req   = speriph_slave[SPER_DATAMOVER_ID].req;
    assign hwpe_cfg_master[0].add   = speriph_slave[SPER_DATAMOVER_ID].add;
    assign hwpe_cfg_master[0].wen   = speriph_slave[SPER_DATAMOVER_ID].wen;
    assign hwpe_cfg_master[0].wdata = speriph_slave[SPER_DATAMOVER_ID].wdata;
    assign hwpe_cfg_master[0].be    = speriph_slave[SPER_DATAMOVER_ID].be;
    assign hwpe_cfg_master[0].id    = speriph_slave[SPER_DATAMOVER_ID].id;
```

Figure 29: Periph interconnect binding for the HW accelerators

As can be seen in Fig. 29, these bindings are performed by assigning an ID to each hardware accelerator. These IDs are defined in the pulp cluster package and are needed to identify the correct address space reserved to each peripheral in the interconnect. Specifically, an address space of 0x400 is dedicated to each peripheral.

3.5: SoC Integration

Concerning the SoC integration, the Datamover has been instantiated by means of a top-level wrap within the fabric controller HWPE subsystem.

3.5.1: Protocols and Interfaces

Given the different interfaces that are used within the SoC with respect to the cluster, writing a wrapper for the IP was needed. The purpose of the wrapper is to bind the interfaces of the two different contexts:

- Within the cluster, the Datamover is connected to the tcdm memory trough a hci interface, whereas the control slave is programmed with a HWPE control interface.
- Within the SoC, the Datamover must be connected to a XBAR_TCDM_BUS (for transferring data) and to an APB_BUS interface (for configuration purposes).

The top wrap of the Datamover has been instantiated as follows.

```
datamover_top_wrap #(
        .ID  ( ID_WIDTH )
    ) i_datamover_top_wrap (
        .clk_i            ( clk_i          ),
        .rst_ni           ( rst_ni         ),
        .test_mode_i      ( test_mode_i    ),
        .tcdm_req         ( tcdm_req       ),
        .tcdm_gnt         ( tcdm_gnt       ),
        .tcdm_add         ( tcdm_add       ),
        .tcdm_wen         ( tcdm_wen       ),
        .tcdm_be          ( tcdm_be        ),
        .tcdm_data        ( tcdm_wdata     ),
        .tcdm_r_data      ( tcdm_r_rdata   ),
        .tcdm_r_valid     ( tcdm_r_valid   ),
        .periph_req       ( periph_req     ),
        .periph_gnt       ( periph_gnt     ),
        .periph_add       ( periph_add     ),
        .periph_wen       ( ~periph_we     ),
        .periph_be        ( periph_be      ),
        .periph_data      ( periph_wdata   ),
        .periph_id        ( '0             ),
        .periph r_data    ( periph_r_rdata ),
        .periph_r_valid   ( periph_r_valid ),
        .periph_r_id      ( periph_r_id    ),
        .evt_o            ( s_evt          )
    );
```

Figure 30: Datamover top wrap interface

Notice that the top wrap of the accelerator exposes the unpacked signals of the two interfaces. This is needed since the interface bindings are performed in the *fc_hwpe* module. Specifically, these bindings are shown in the two following figures.

```
genvar i;
    generate
        for (i=0;i<4;i++) begin : hwacc_binding
            assign hwacc_xbar_master[i].req   = tcdm_req   [i];
            assign hwacc_xbar_master[i].add   = tcdm_add   [i];
            assign hwacc_xbar_master[i].wen   = tcdm_wen   [i];
            assign hwacc_xbar_master[i].wdata = tcdm_wdata [i];
            assign hwacc_xbar_master[i].be    = tcdm_be    [i];
            // response channel
            assign tcdm_gnt     [i] = hwacc_xbar_master[i].gnt;
            assign tcdm_r_rdata [i] = hwacc_xbar_master[i].r_rdata;
            assign tcdm_r_valid [i] = hwacc_xbar_master[i].r_valid;
        end
    endgenerate
```

Figure 31: Tcdm interface binding in the fabric controller

```
apb2per #(
        .PER_ADDR_WIDTH ( 32             ),
        .APB_ADDR_WIDTH ( APB_ADDR_WIDTH )
    ) i_apb2per (
        .clk_i                ( clk_i                   ),
        .rst_ni               ( rst_ni                  ),
        .PADDR                ( hwacc_cfg_slave.paddr   ),
        .PWDATA               ( hwacc_cfg_slave.pwdata  ),
        .PWRITE               ( hwacc_cfg_slave.pwrite  ),
        .PSEL                 ( hwacc_cfg_slave.psel    ),
        .PENABLE              ( hwacc_cfg_slave.penable ),
        .PRDATA               ( hwacc_cfg_slave.prdata  ),
        .PREADY               ( hwacc_cfg_slave.pready  ),
        .PSLVERR              ( hwacc_cfg_slave.pslverr ),
        .per_master_req_o     ( periph_req              ),
        .per_master_add_o     ( periph_add              ),
        .per_master_we_o      ( periph_we               ),
        .per_master_wdata_o   ( periph_wdata            ),
        .per_master_be_o      ( periph_be               ),
        .per_master_gnt_i     ( periph_gnt              ),
        .per_master_r_valid_i ( periph_r_valid          ),
        .per_master_r_opc_i   ( periph_r_opc            ),
        .per_master_r_rdata_i ( periph_r_rdata          )
    );
```

Figure 32: Periph interface binding in the fabric controller

In the following section, the testing of the Datamover will be illustrated, both for the cluster and the SoC instances.

3.6: Testing the Datamover

The Datamover test is written in C language and requires a certain protocol of operations to be performed in order to operate correctly. Also, the configuration of the Datamover is carried out in the test file, by using functions to write into the control registers. Some optimization has been introduced:

- The number of contexts exposed from the *hwpe_ctrl_slave* module has been reduced from 2 to 1. This is useful to decrease the area occupation of the control slave module, but it prevents multiple jobs from being configured at the same time.
- Also, the same registers have been used for storing input and output lengths along the dimensions d0 and d1. This allowed to reduce the original number of registers from 13 to 11.

- Also, the register [2] has been used to store the total transfer length and the buffer factor.

Of course, the Datamover has been tested both for the cluster and the SoC instances. This required inserting a USE_CLUSTER parameter to select which instance to test. Also, a correct base address needs to be specified in order to correctly select and configure the accelerator registers. Of course, besides specifying a new base address, also a new set of functions within *hal_datamover.h* had to be defined to test the SoC instance of the Datamover. In this section, the protocol for testing the cluster instance of the accelerator will be described.

The following functions are used when launching a job on the Datamover:

- DATAMOVER_CG_ENABLE: this function enables the clock for the accelerator.
- DATAMOVER_SET_PRIORITY_DATAMOVER: this functions gives the priority to the Datamover with respect to the cores of the cluster and the DMA.
- DATAMOVER_RESET_MAXSTALL: this function resets the maximum stall previously configured.
- DATAMOVER_SET_MAXSTALL: this function sets the maximum consecutive stall to 8 cycles for the cores on the DMA side.

After these preliminary operations, a soft clear must be given to the Datamover. This is done by means of the DATAMOVER_WRITE_CMD function which, given an offset and a value as parameters, it writes the specified value to the address pointed at by the offset added to the Datamover base address. Then, the DATAMOVER_BARRIER_ACQUIRE function is called, in order to acquire the specified job.

Following on, the register context of the *hwpe_ctrl_slave* need to be configured. This is done by means of the DATAMOVER_WRITE_REG function which, in a similar way to the WRITE_CMD function, allows to write on the Datamover registers.

```
// set up datamover
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_PTR,  x);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_OUT_PTR, y);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_TOT_LEN,       BUFFER_FACTOR_AND_TOT_LEN);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_OUT_D0_LEN, LEN_IN_OUT_D0);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_D0_STRIDE,  STRIDE_IN_D0);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_OUT_D1_LEN, LEN_IN_OUT_D1);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_D1_STRIDE,  STRIDE_IN_D1);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_D2_STRIDE,  STRIDE_IN_D2);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_OUT_D0_STRIDE, STRIDE_OUT_D0);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_OUT_D1_STRIDE, STRIDE_OUT_D1);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_OUT_D2_STRIDE, STRIDE_OUT_D2);
```

Figure 33: Datamover setup

After having configured the Datamover, the job needs to be launched. This is again done by means of the WRITE_CMD function, for which the specified parameters are DATAMOVER_COMMIT_AND_TRIGGER and DATAMOVER_TRIGGER_CMD. Notice that also DATAMOVER_COMMIT_CMD can be passed as a second parameter. This can be done to configure more than one job on the accelerator and launch them in succession, provided at least two configuration contexts are exposed by the control slave.

At this point, the execution of the transfer takes place until the DATAMOVER_BARRIER function. Then, the clock needs to be disabled by means of the DATAMOVER_CG_DISABLE function. Then, the priority is given back to the core side by using the DATAMOVER_SETPRIORITY_CORE function. After the execution has taken place, the produced results are checked with respect to a golden model. The input data used in the test can be generated by means of a 32-bit LFSR (*Linear Feedback Shift Register*) or an 8-bit counter. The LFSR produces pseudo-random 32-bit words, whereas the 8-bit counter returns 32-bit words obtained by the concatenation of four successive values. The return value of the check function is computed as the number of errors found in the output data. If, given a certain output address, the read data is different from the expected one, a counter is increased. This allows to easily debug the behaviour of the accelerator.

The golden models are specified as arrays in the *golden_arrays.h* file, and they cover all the supported data-widths (32-, 16-, 8-, 4-, 2- and 1-bit) for different sizes of transfers (64-, 128-, 256-, 512- and 1024-words). The golden model to check the results with can be selected inside the check function.

3.7: Use Cases

The Datamover has been tested for a variety of tensor sizes and reordering granularities within each word. These are the main use cases for the Datamover, and the performance results will be shown in the last chapter of the thesis. Besides these, it's also possible to act on the dimensional parameters of the address generators to shuffle the tensor at word-level.

First of all, the transposition of rows and columns in the (W, H) plane can be performed by configuring the address generators with the following sets of strides:

$$Input\ strides: < d0, d1, d2 > \mid Output\ strides: < d1, d0, d2 >$$

So, data which was previously stored in the d0 dimension with a stride of 4, gets reordered with a stride of 8*4 (e.g. in the case of a $8 \ x \ 8 \ x \ 4$ tensor) in the output address range. This operation can easily be represented graphically as:

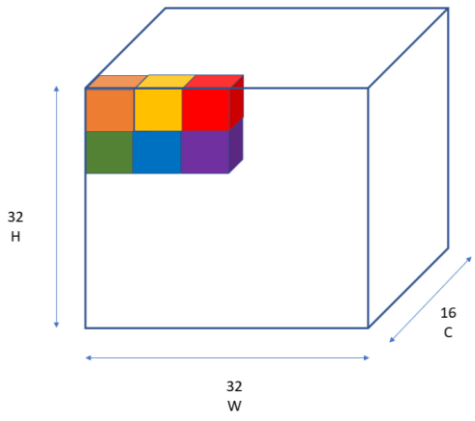

Figure 34: Input data layout

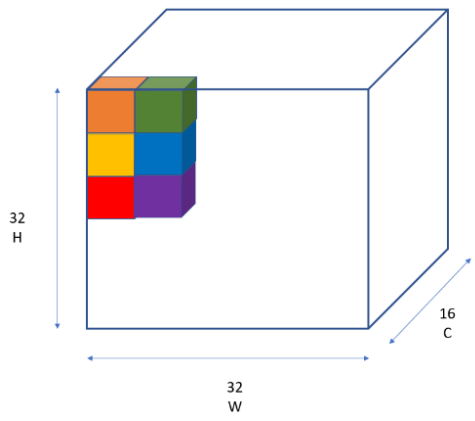

Figure 35: Output data layout after transposition

Where each cube is a 32-bit word stored into an address of the tcdm memory. So, the input tensor can be transposed in the (H,W) plane simply by inverting the strides along the dimensions d0 (W) and d1 (H) on the output address generator. Of course, this transposition along these two dimensions occurs for every section of the tensor, so by counting channels along the d2 dimension.

Another reordering operation that can be performed by the Datamover is the conversion between the HWC and the CHW data formats. This requires a different configuration for the

address generator modules. Specifically, the input and output sets of strides are chosen as follows:

$$Input\ strides: < d1, d0, d2 > | Output\ strides: < d2, d1, d0 >$$

The operation can be graphically represented as:

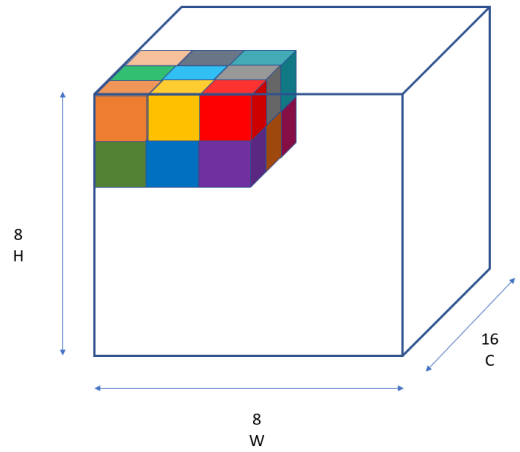

Figure 36: Input data before HWC to CHW conversion

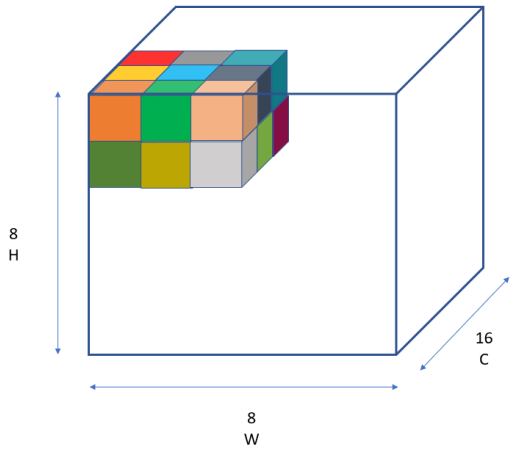

Figure 37: Data after HWC to CHW conversion

So, data stored along the channel dimension in the input tensor is stored along the width dimension in the output tensor. Basically, this can be seen as a transposition of the tensor data in the (W,C) plane. Indeed, data placed across the diagonal of the upper face remains in the same position after the conversion. This allows to access the data across the channels as rows in the (W,H) plane.

Of course, these are just two of the possible configurations for the address generator modules. By correctly setting the lengths along each dimension and the related strides, other reordering operations can be carried out.

Besides reordering at word-level, data can be transposed within each word of the tensor for the supported data widths listed at the beginning of this section. The following figures show some examples of this behaviour.

```
Input Addr: 10002428 | Input Data: 10203 | Output Addr: 10002528 | Output Data: 4080c
Input Addr: 1000242c | Input Data: 4050607 | Output Addr: 1000252c | Output Data: 105090d
Input Addr: 10002430 | Input Data: 8090a0b | Output Addr: 10002530 | Output Data: 2060a0e
Input Addr: 10002434 | Input Data: c0d0e0f | Output Addr: 10002534 | Output Data: 3070b0f
Input Addr: 10002438 | Input Data: 10111213 | Output Addr: 10002538 | Output Data: 1014181c
Input Addr: 1000243c | Input Data: 14151617 | Output Addr: 1000253c | Output Data: 1115191d
Input Addr: 10002440 | Input Data: 18191a1b | Output Addr: 10002540 | Output Data: 12161a1e
Input Addr: 10002444 | Input Data: 1c1d1e1f | Output Addr: 10002544 | Output Data: 13171b1f
Input Addr: 10002448 | Input Data: 20212223 | Output Addr: 10002548 | Output Data: 2024282c
Input Addr: 1000244c | Input Data: 24252627 | Output Addr: 1000254c | Output Data: 2125292d
Input Addr: 10002450 | Input Data: 28292a2b | Output Addr: 10002550 | Output Data: 22262a2e
Input Addr: 10002454 | Input Data: 2c2d2e2f | Output Addr: 10002554 | Output Data: 23272b2f
```

Figure 38: Reordering on 8-bit data

In the previous figure, the transposition within each output word is performed with an 8-bit granularity. For reference, the first output word is 0x0004080c, which is clearly the concatenation of the MSB 8-bits of the first four input words. Analogously, the second output word is the concatenation of the 2nd 8-bits elements of the first four input words, resulting in 0x0105090d and so on.

```
Input Addr: 10002428 | Input Data: 10203 | Output Addr: 10002528 | Output Data: 1111
Input Addr: 1000242c | Input Data: 4050607 | Output Addr: 1000252c | Output Data: 48c048c
Input Addr: 10002430 | Input Data: 8090a0b | Output Addr: 10002530 | Output Data: 1111
Input Addr: 10002434 | Input Data: c0d0e0f | Output Addr: 10002534 | Output Data: 159d159d
Input Addr: 10002438 | Input Data: 10111213 | Output Addr: 10002538 | Output Data: 1111
Input Addr: 1000243c | Input Data: 14151617 | Output Addr: 1000253c | Output Data: 26ae26ae
Input Addr: 10002440 | Input Data: 18191a1b | Output Addr: 10002540 | Output Data: 1111
Input Addr: 10002444 | Input Data: 1c1d1e1f | Output Addr: 10002544 | Output Data: 37bf37bf
Input Addr: 10002448 | Input Data: 20212223 | Output Addr: 10002548 | Output Data: 22223333
Input Addr: 1000244c | Input Data: 24252627 | Output Addr: 1000254c | Output Data: 48c048c
Input Addr: 10002450 | Input Data: 28292a2b | Output Addr: 10002550 | Output Data: 22223333
Input Addr: 10002454 | Input Data: 2c2d2e2f | Output Addr: 10002554 | Output Data: 159d159d
```

Figure 39: Reordering on 4-bit data

Following on, these are the observed results when the transposition is performed with a 4-bit granularity. If the first output word, which is 0x00001111, is again considered as reference, it's clear that it's the result of the concatenation of the MSB 4-bits of each of the first eight input words. The same holds for the following words.

In the two figures below, the results for 2-bits and 1-bit granularities are shown.

```
Input Addr: 10002428 | Input Data: 10203 | Output Addr: 10002528 | Output Data:  0
Input Addr: 1000242c | Input Data: 4050607 | Output Addr: 1000252c | Output Data: 55aaff
Input Addr: 10002430 | Input Data: 8090a0b | Output Addr: 10002530 | Output Data: 1b1b1b1b
Input Addr: 10002434 | Input Data: c0d0e0f | Output Addr: 10002534 | Output Data:  0
Input Addr: 10002438 | Input Data: 10111213 | Output Addr: 10002538 | Output Data:  0
Input Addr: 1000243c | Input Data: 14151617 | Output Addr: 1000253c | Output Data: 55aaff
Input Addr: 10002440 | Input Data: 18191a1b | Output Addr: 10002540 | Output Data: 1b1b1b1b
Input Addr: 10002444 | Input Data: 1c1d1e1f | Output Addr: 10002544 | Output Data: 55555555
Input Addr: 10002448 | Input Data: 20212223 | Output Addr: 10002548 | Output Data:  0
Input Addr: 1000244c | Input Data: 24252627 | Output Addr: 1000254c | Output Data: 55aaff
Input Addr: 10002450 | Input Data: 28292a2b | Output Addr: 10002550 | Output Data: 1b1b1b1b
Input Addr: 10002454 | Input Data: 2c2d2e2f | Output Addr: 10002554 | Output Data: aaaaaaaa
Input Addr: 10002458 | Input Data: 30313233 | Output Addr: 10002558 | Output Data:  0
Input Addr: 1000245c | Input Data: 34353637 | Output Addr: 1000255c | Output Data: 55aaff
Input Addr: 10002460 | Input Data: 38393a3b | Output Addr: 10002560 | Output Data: 1b1b1b1b
Input Addr: 10002464 | Input Data: 3c3d3e3f | Output Addr: 10002564 | Output Data: ffffffff
Input Addr: 10002468 | Input Data: 40414243 | Output Addr: 10002568 | Output Data: 55555555
Input Addr: 1000246c | Input Data: 44454647 | Output Addr: 1000256c | Output Data: 55aaff
Input Addr: 10002470 | Input Data: 48494a4b | Output Addr: 10002570 | Output Data: 1b1b1b1b
Input Addr: 10002474 | Input Data: 4c4d4e4f | Output Addr: 10002574 | Output Data:  0
```

Figure 40: Reordering on 2-bit data

```
Input Addr: 10002428 | Input Data: 10203 | Output Addr: 10002528 | Output Data:  0
Input Addr: 1000242c | Input Data: 4050607 | Output Addr: 1000252c | Output Data: ffff
Input Addr: 10002430 | Input Data: 8090a0b | Output Addr: 10002530 | Output Data: ff00ff
Input Addr: 10002434 | Input Data: c0d0e0f | Output Addr: 10002534 | Output Data: f0f0f0f
Input Addr: 10002438 | Input Data: 10111213 | Output Addr: 10002538 | Output Data: 33333333
Input Addr: 1000243c | Input Data: 14151617 | Output Addr: 1000253c | Output Data: 55555555
Input Addr: 10002440 | Input Data: 18191a1b | Output Addr: 10002540 | Output Data:  0
Input Addr: 10002444 | Input Data: 1c1d1e1f | Output Addr: 10002544 | Output Data:  0
Input Addr: 10002448 | Input Data: 20212223 | Output Addr: 10002548 | Output Data:  0
Input Addr: 1000244c | Input Data: 24252627 | Output Addr: 1000254c | Output Data: ffff
Input Addr: 10002450 | Input Data: 28292a2b | Output Addr: 10002550 | Output Data: ff00ff
Input Addr: 10002454 | Input Data: 2c2d2e2f | Output Addr: 10002554 | Output Data: f0f0f0f
Input Addr: 10002458 | Input Data: 30313233 | Output Addr: 10002558 | Output Data: 33333333
Input Addr: 1000245c | Input Data: 34353637 | Output Addr: 1000255c | Output Data: 55555555
Input Addr: 10002460 | Input Data: 38393a3b | Output Addr: 10002560 | Output Data:  0
Input Addr: 10002464 | Input Data: 3c3d3e3f | Output Addr: 10002564 | Output Data: ffffffff
Input Addr: 10002468 | Input Data: 40414243 | Output Addr: 10002568 | Output Data:  0
Input Addr: 1000246c | Input Data: 44454647 | Output Addr: 1000256c | Output Data: ffff
Input Addr: 10002470 | Input Data: 48494a4b | Output Addr: 10002570 | Output Data: ff00ff
Input Addr: 10002474 | Input Data: 4c4d4e4f | Output Addr: 10002574 | Output Data: f0f0f0f
```

Figure 41: Reordering on 1-bit data

4: Experimental Results

In this chapter, the synthesis results will be illustrated for both the Datamover and the overall Darkside SoC. Each design is synthesized with tsmc65 libraries for a 65 nm technology.

4.1: Standalone Results

In this section, the results for the synthesis of the Datamover are shown. First of all, the structure of the synthesis script will be illustrated, followed by the results in terms of area and timing.

4.1.1: Setup

The design has been synthesized with the Synopsys Design Compiler. The synthesis scripts is structured as follows:

- First of all, the paths to the main folders containing the rtl of the Datamover and the IPs used within it are specified.
- Then, the options for *timing_enable_through_paths* and *compile_timing_high_effort* are set as true.
- Following on, all the rtl of the design is analysed with a bottom-up approach. Starting from the packages all the way to the top module of the accelerator.
- After having analysed the whole hierarchy of the design, the link command is executed, with the correct return value being 1. This assures that no unresolved references are found in the design. Then, the constrains are set in terms of clock period, clock uncertainty and the option to not optimize the clock network is specified.
- Then, the input and output delays are set, together with the driving cell and library used for synthesizing the design. Also the load is specified during this phase.
- At this point the design is checked and compiled. Then, the reports are generated. These include area occupation, power measures, timing constraints and an overall report on the design.
- A Verilog netlist is also produced for post synthesis simulations.

The following results were obtained in a worst-case scenario, with the chosen library being *sc8_cln65lp_base_lvt_ss_typical_max_0p90v_125c*, so with the transistors threshold set as low, the alimentation voltage as 0.90 V and the temperature being 125 °C.

4.1.2: Timing and area

The following results were gathered from the reports generated by means of the synthesis script. First of all, the minimum clock period for which no timing violations occur in the design is:

$Clock\ period$	$Slack$
2.7 ns	0.0001 ns

The longest path for data is the between two registers for the *data_count* signal used inside the buffers to keep track of the total number of transfers.

The minimum clock period found for the Datamover is quite smaller than the one used for synthesizing the cluster, which is 4.5 ns. So, a decrease in area occupation is to be expected. The area occupation for both clock periods is:

$Clock\ period$	$Area$
2.7 ns (standalone)	77749.881 μm^2
4.5 ns $(cluster)$	69382.401 μm^2

Referring to the synthesis with a clock period of 2.7 ns, more than half of the final area occupation is due to the array of 32 buffers, each one with an area of 1500 μm^2 approximately. The other main contributions are given by the slave control module, which alone has an area of more than 10000 μm^2 and the streamer, which accounts for more than 15000 μm^2.

When inserted in the cluster, the Datamover accounts for around 2% of the total area occupation, making it slightly smaller than a single core.

4.1.3: Performance

In this section the results obtained through the use of performance counters are illustrated. These performance counters allow to measure various statistics related to the execution of a task. Performance results have been gathered for the cluster instance of the Datamover. The results that follow are organized as below:

- Table [1]: performances for 64-words transfer with 8-, 4-, 2- and 1-bit transpositions.
- Table [2]: performances for 128-words transfer with 8-, 4-, 2- and 1-bit transpositions.
- Table [3]: performances for 256-words transfer with 8-, 4-, 2- and 1-bit transpositions.
- Table [4]: performances for 512-words transfer with 8-, 4-, 2- and 1-bit transpositions.
- Table [5]: performances for 1024-words transfer with 8-, 4-, 2- and 1-bit transpositions.

The performance of the accelerator with respect to the software emulation has been measured by means of the PI_PERF_CYCLES, which allows to measure the cycles needed for a specific task. This counter can be simply used by configuring it with the desired metric and placing the start and stop functions at the edges of the task to be measured.

The setup for measuring the accelerator performance is the following:

```
pi_perf_conf (1 << PI_PERF_CYCLES);
    pi_perf_reset();
    pi_perf_start();

    // acquire job
    int job_id = -1;
    DATAMOVER_BARRIER_ACQUIRE(job_id);

    // printf ("Job acquired \n");

    // set up datamover
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_PTR,  x);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_OUT_PTR, y);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_TOT_LEN,       BUFFER_FACTOR_AND_TOT_LEN);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_OUT_D0_LEN, LEN_IN_OUT_D0);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_D0_STRIDE,  STRIDE_IN_D0);
```

```
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_OUT_D1_LEN, LEN_IN_OUT_D1);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_D1_STRIDE,  STRIDE_IN_D1);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_IN_D2_STRIDE,  STRIDE_IN_D2);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_OUT_D0_STRIDE, STRIDE_OUT_D0);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_OUT_D1_STRIDE, STRIDE_OUT_D1);
    DATAMOVER_WRITE_REG(DATAMOVER_REG_OUT_D2_STRIDE, STRIDE_OUT_D2);

    // printf ("Datamover set up done \n");

    // commit and trigger datamover operation
    DATAMOVER_WRITE_CMD(DATAMOVER_COMMIT_AND_TRIGGER, DATAMOVER_TRIGGER_CMD);

    // printf ("Commit and trigger operation done \n");

    // wait for end of computation
    DATAMOVER_BARRIER();

    pi_perf_stop();
    uint32_t cnt_cycles_acc = pi_perf_read(PI_PERF_CYCLES);
```

Figure 42: PI_PERF_CYCLES accelerator setup

So, the measured cycles account for the configuration of the accelerator, the barriers, and the execution of the transfer. The number of cycles is then stored into a dedicated variable. A similar procedure has been put in place for measuring the performance of the software emulation.

In the following page, only the code behind the software emulation for the 8-bits and 4-bits reordering is shown since the same code also works for the remaining cases when appropriately scaled.

```
for (addr_x = addr_first_x; addr_x < addr_last_x; addr_x += 16)
        {
                        for (addr_index = 0 ; addr_index <4; addr_index ++)
                        {
                                addr_tmp = addr_x + 3 - addr_index;
                                for (i=0; i<4; i++)
                                {
                                        data_tmp[3-i] = *(uint8_t *)(addr_tmp);
                                        addr_tmp = addr_tmp + 4;
                                }
                                for (i=0; i<4; i++)
                                        *(uint8_t *)(addr_y+i) = data_tmp[i];
                                addr_y = addr_y + 4;
                        }
        }
```

Figure 43: Software emulation for 8-bits reordering

```
for (addr_x = addr_first_x; addr_x < addr_last_x; addr_x += 32)
{
        for (addr_index = 0 ; addr_index < 4; addr_index ++)
        {
                addr_tmp = addr_x + 3 - addr_index;
                for (i=0; i<8; i++)
                {
                        byte_tmp_x = *(uint8_t *)(addr_tmp);
                        data_tmp_1[i] = byte_tmp_x >> 4;
                        data_tmp_2[i] = byte_tmp_x & 0x0F;
                        addr_tmp = addr_tmp + 4;
                }
                for (i=0; i<8; i++)
                        word_tmp = word_tmp | (data_tmp_1 [i] << (32-4*(i+1)));
                *(uint32_t *)(addr_y) = word_tmp;
                word_tmp = 0;
                addr_y = addr_y + 4;
                for (i=0; i<8; i++)
                        word_tmp = word_tmp | (data_tmp_2 [i] << (32-4*(i+1)));
                *(uint32_t *)(addr_y) = word_tmp;
                word_tmp = 0;
                addr_y = addr_y + 4;
        }
}
```

Figure 44: Software emulation for 4-bits reordering

Whereas 8-bits wide data can be easily accessed at a certain address, handling sub-byte data requires some bitwise manipulations. This was done by implementing for loops with a parametrized shifting mechanism to handle data. To this end, some bitwise operations made available in C have been used.

The gathered results are organized in the following tables:

$64 - words\ transfer$	$Software\ (cycles)$	$Hardware\ (cycles)$
$32 - bits\ reordering$	346	429 (367)
$16 - bits\ reordering$	449	375(358)
$8 - bits\ reordering$	805	359
$4 - bits\ reordering$	3133	351
$2 - bits\ reordering$	5761	347
$1 - bit\ reordering$	17983	369

Table 1: Reordering performance of a 64-words transfer

128 $words\ transfer$	$Software\ (cycles)$	$Hardware\ (cycles)$
$32-bits\ reordering$	538	495
$16-bits\ reordering$	738	503
$8-bits\ reordering$	1481	517
$4-bits\ reordering$	5235	510
$2-bits\ reordering$	10649	503
$1-bit\ reordering$	33987	499

Table 2: Reordering performance of a 128-words transfer

256 $words\ transfer$	$Software\ (cycles)$	$Hardware\ (cycles)$
$32-bits\ reordering$	920	748
$16-bits\ reordering$	1314	791
$8-bits\ reordering$	2540	805
$4-bits\ reordering$	9690	783
$2-bits\ reordering$	20191	767
$1-bit\ reordering$	66141	759

Table 3: Reordering performance of a 256-words transfer

512 $words\ transfer$	$Software\ (cycles)$	$Hardware\ (cycles)$
$32-bits\ reordering$	1691	1239
$16-bits\ reordering$	2466	1380
$8-bits\ reordering$	4652	1393
$4-bits\ reordering$	18145	1288
$2-bits\ reordering$	38845	1257
$1-bit\ reordering$	130540	1241

Table 4: Reordering performance of a 512-words transfer

1024 $words\ transfer$	$Software\ (cycles)$	$Hardware\ (cycles)$
$32-bits\ reordering$	3229	2286
$16-bits\ reordering$	5282	2543
$8-bits\ reordering$	9128	2504
$4-bits\ reordering$	35389	2419
$2-bits\ reordering$	77266	2356
$1-bit\ reordering$	258762	2323

Table 5: Reordering performance of a 1024-words transfer

Some observations on the results:

- First of all, it's clear how the accelerator allows to significantly speed up the reordering task, especially for sub-byte data widths. This becomes even more clear when increasing the number of transferred words. Referring to Table [1] (64 words transfer), roughly a 2.24x speedup is measured for 8-bits data. This increases up to nearly 49x when considering 1-bit data. The performances are already better when considering Table [2]

(128 words transfer) where a peak speedup of 68x can be observed. The overall best result is, as expected, the one for a 1024-words transfer with 1-bit reordering (Table [5]), where the accelerator achieves a 111x speedup when compared to the software execution.

- It's also quite clear how the software increases its execution time by a factor 2 each time the number of words is doubled, whereas the overhead is significantly smaller for the accelerator.
- An overhead was observed in the first version of the buffer FSM. This was due to the request signal of the tcdm interface going low for a single clock cycle each time a word was processed. This issue was solved by adding an intermediate state (FULL_RESET) to reset the counter that keeps track of the cycles during which the content of the buffer must be kept fixed. By eliminating this additional cost, the performances for the 32-bits and 16-bits have been improved as can be seen in the previous tables (cycles between parenthesis). This overhead also explains why reducing the data-width on which the reordering is performed allows to reduce the hardware execution time. When selecting a 32-bit granularity (let-through transfer), the extra cycle occurred much more often than when choosing a lower data-width.
- This overhead also led to the accelerator being outperformed when choosing to simply transfer the tensor and perform no transposition operations. After having removed the overhead, the accelerator is outperformed only when choosing a transfer size of 64 words.

The results gathered in the previous tables are visually represented in the following graphs. Specifically, the first two graphs expose the comparison between the software and hardware execution for 64-words and 1024-words transfers. The last graph summarizes the results in terms of cycles/byte for every hardware execution.

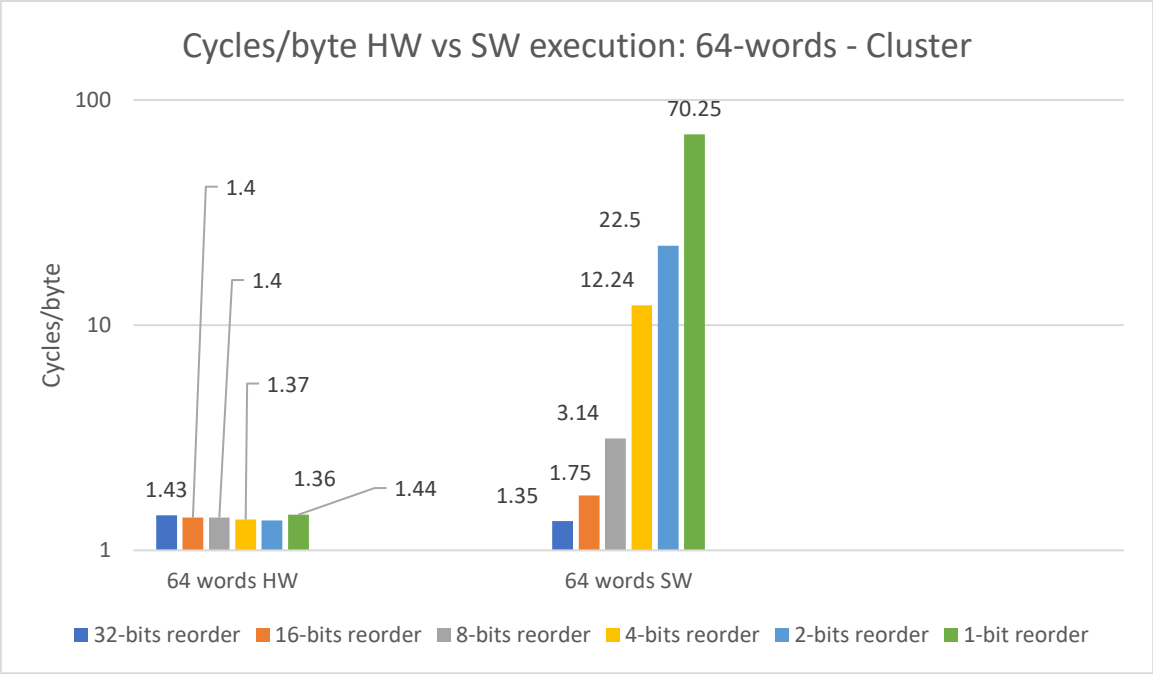

Figure 45: Comparison between HW and SW executions of a 64-words transfer in the cluster

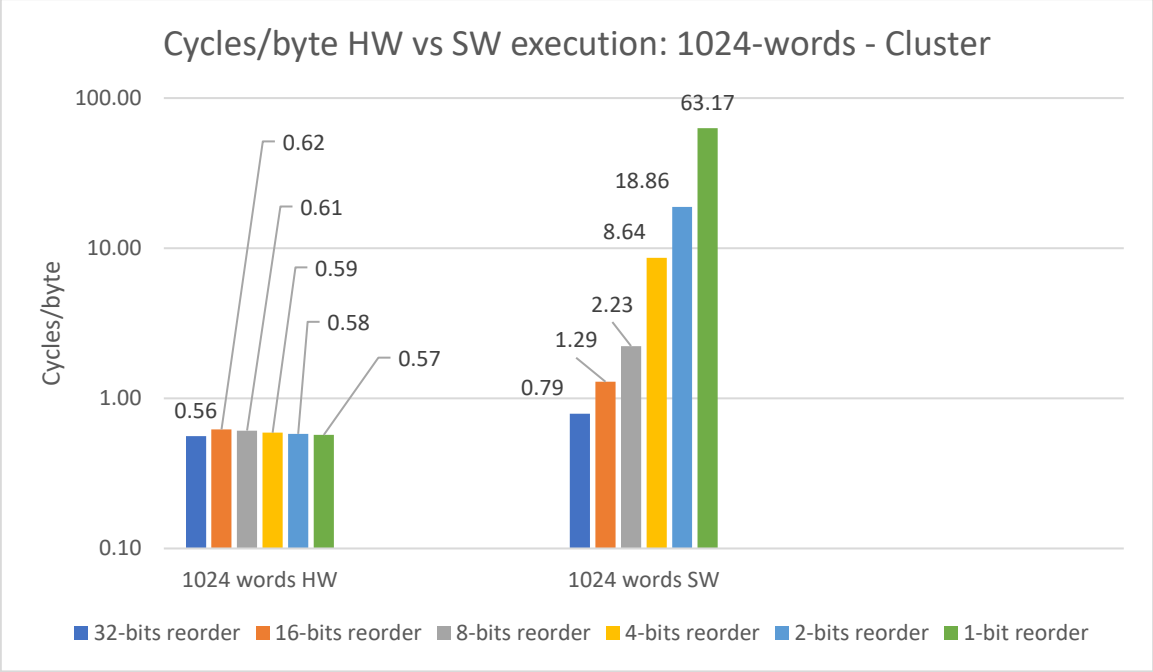

Figure 46: Comparison between HW and SW executions of a 1024-words transfer in the cluster

It's clear how the execution time of the accelerator remains almost constant regardless of the chosen granularity, whereas the software execution is deeply affected by it. From being close to the accelerator when performing no reorder operations, the software performance gets dramatically worse when decreasing the data-width on which the transposition is performed.

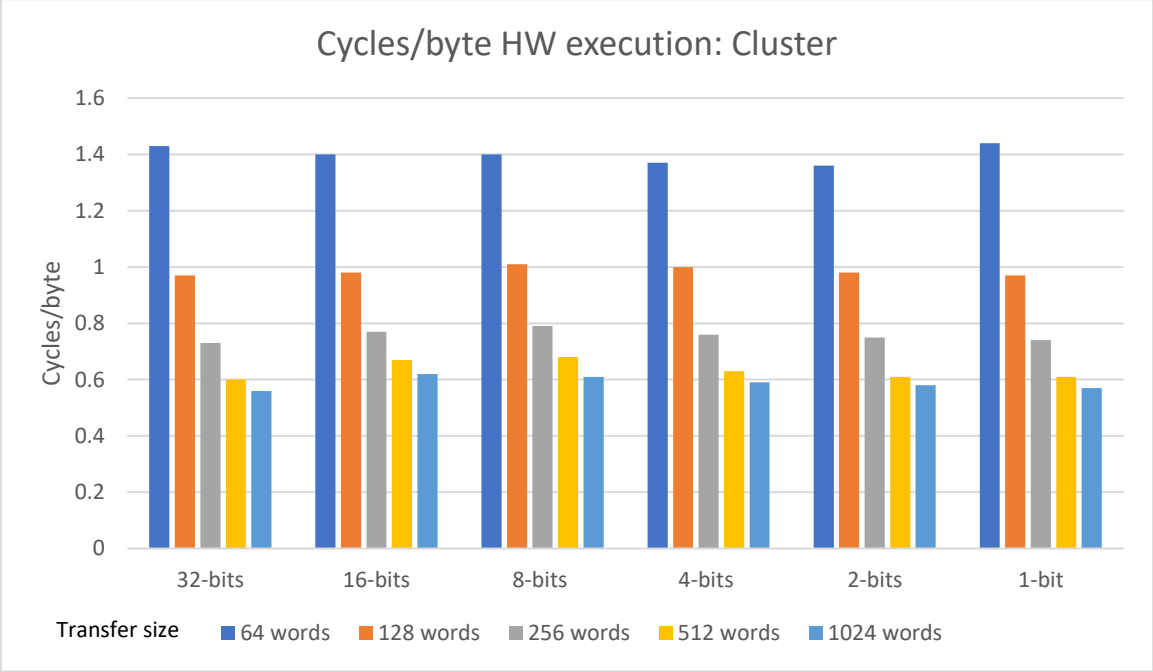

Figure 47: Overall HW performances

By looking at Fig. 47 it's clear how the performance of the accelerator significantly increases for larger transfer sizes. This is due to the configuration overhead being absorbed once the number of words goes over 64.

In the following tables, the performance results related to the SoC instance of the Datamover are shown, although similar results are to be expected:

$64\ words\ transfer$	$Software\ (cycles)$	$Hardware\ (cycles)$
$32-bits\ reordering$	220	237
$16-bits\ reordering$	317	253
$8-bits\ reordering$	581	254
$4-bits\ reordering$	1722	245
$2-bits\ reordering$	3655	241
$1-bit\ reordering$	14757	239

Table 6: Reordering performance of a 64-words transfer

128 $words\ transfer$	$Software\ (cycles)$	$Hardware\ (cycles)$
$32-bits\ reordering$	412	365
$16-bits\ reordering$	605	397
$8-bits\ reordering$	1109	398
$4-bits\ reordering$	3402	381
$2-bits\ reordering$	7271	373
$1-bit\ reordering$	29489	369

Table 7: Reordering performance of a 128-words transfer

256 $words\ transfer$	$Software\ (cycles)$	$Hardware\ (cycles)$
$32-bits\ reordering$	796	621
$16-bits\ reordering$	1181	685
$8-bits\ reordering$	2165	686
$4-bits\ reordering$	6762	653
$2-bits\ reordering$	14503	637
$1-bit\ reordering$	58953	629

Table 8: Reordering performance of a 256-words transfer

512 $words\ transfer$	$Software\ (cycles)$	$Hardware\ (cycles)$
$32-bits\ reordering$	1564	1134
$16-bits\ reordering$	2333	1260
$8-bits\ reordering$	4277	1261
$4-bits\ reordering$	13482	1198
$2-bits\ reordering$	28967	1166
$1-bit\ reordering$	117881	1150

Table 9: Reordering performance of a 512-words transfer

1024 $words\ transfer$	$Software\ (cycles)$	$Hardware\ (cycles)$
$32-bits\ reordering$	3102	2157
$16-bits\ reordering$	5148	2413
$8-bits\ reordering$	8756	2414
$4-bits\ reordering$	26796	2285
$2-bits\ reordering$	58149	2221
$1-bit\ reordering$	235642	2189

Table 10: Reordering performance of a 1024-words transfer

The results are quite similar to those obtained for the cluster instance of the Datamover. Indeed, the accelerator is outperformed by the software execution only when choosing to transfer 64 words without performing any reorder operation, as in the cluster.

Also, by checking the top performance of the accelerator (1024-words transfer with 1-bit reordering), the obtained speedup is around 107x, whereas a top speedup of 111x was observed in the cluster.

In the following graphs, the Cycles/byte metric is evaluated for SoC Datamover instance.

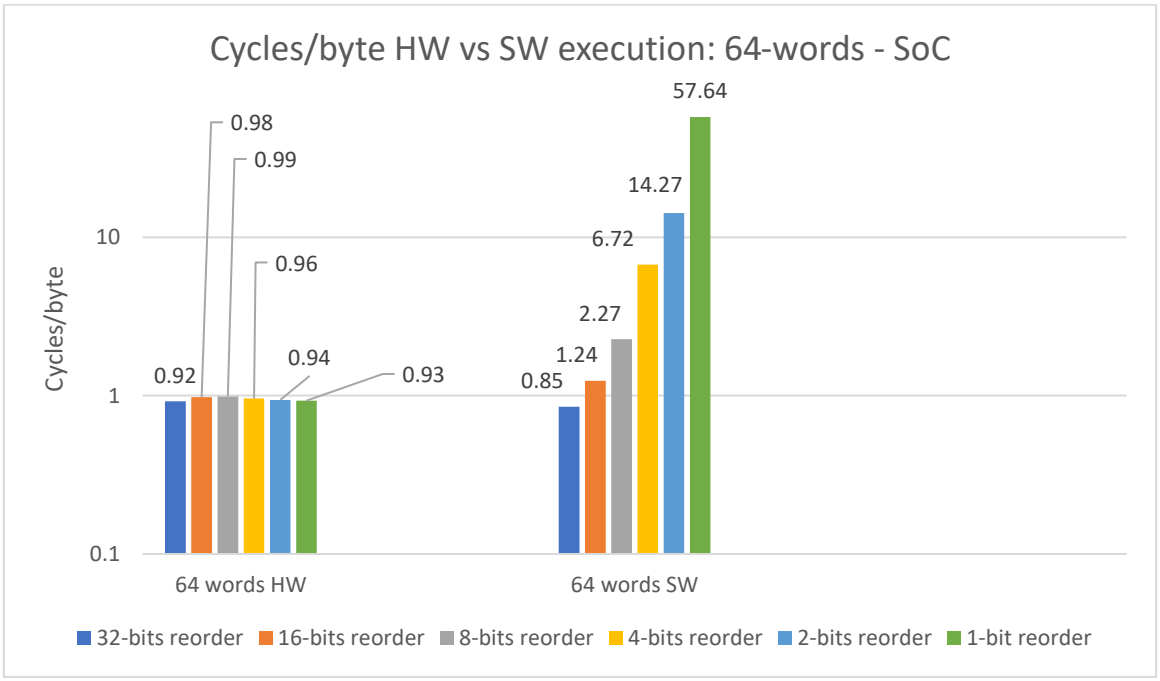

Figure 48: Comparison between HW and SW executions of a 64-words transfer in the SoC

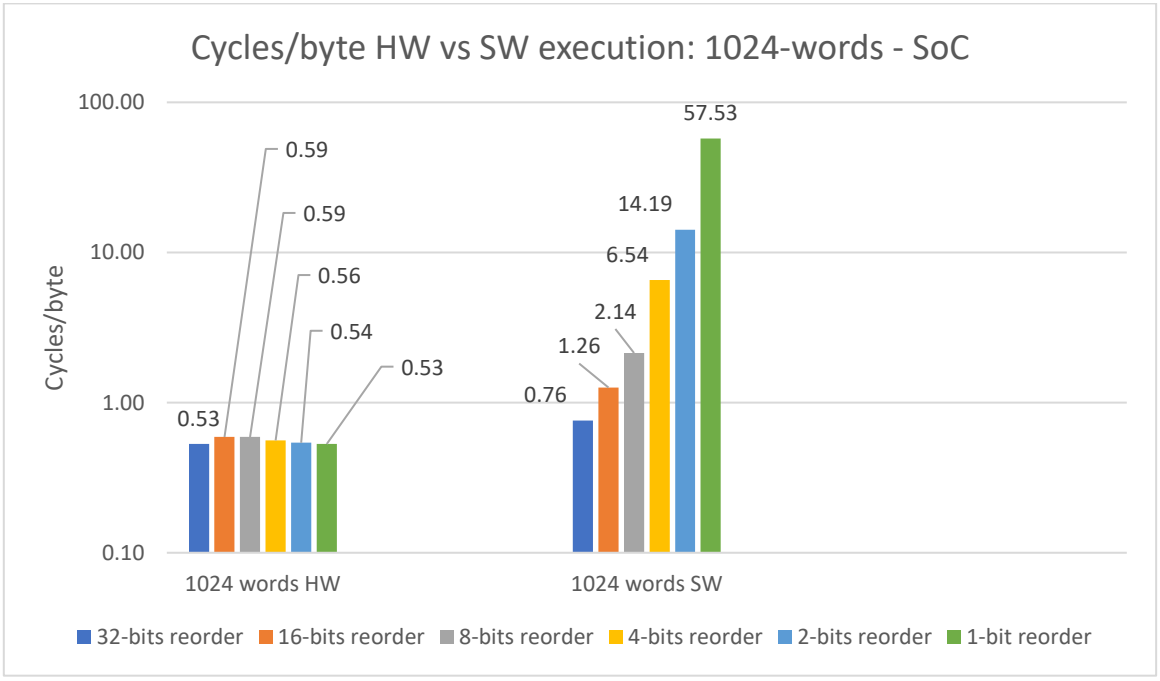

Figure 49: Comparison between HW and SW executions of a 1024-words transfer in the SoC

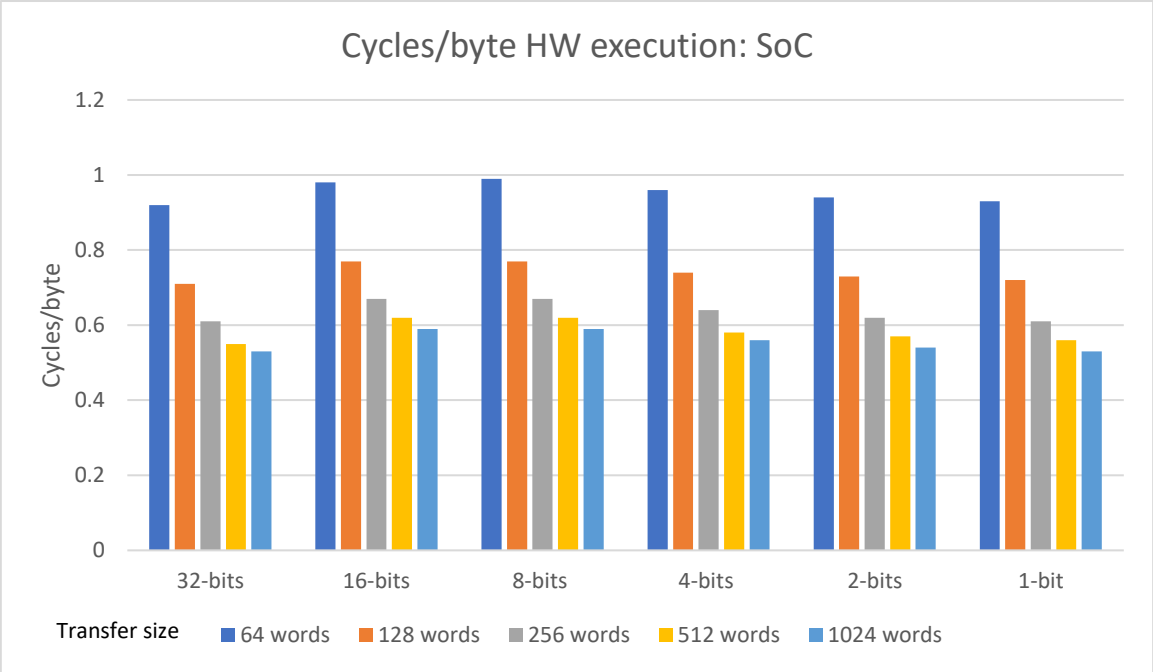

Figure 50: Overall HW performances

By comparing this graph with Fig. 47, it can be observed that, while the worst cluster performances stood around the 1,4 cycles/byte mark for the SoC the results are slightly better, with the worst performance always being lower than 1 cycles/byte.

4.2: Putting it all together: The Darkside SoC

In this last section, some results regarding the Darkside cluster are shown. In Fig. 51, the results in terms of area occupation for the cluster are illustrated.

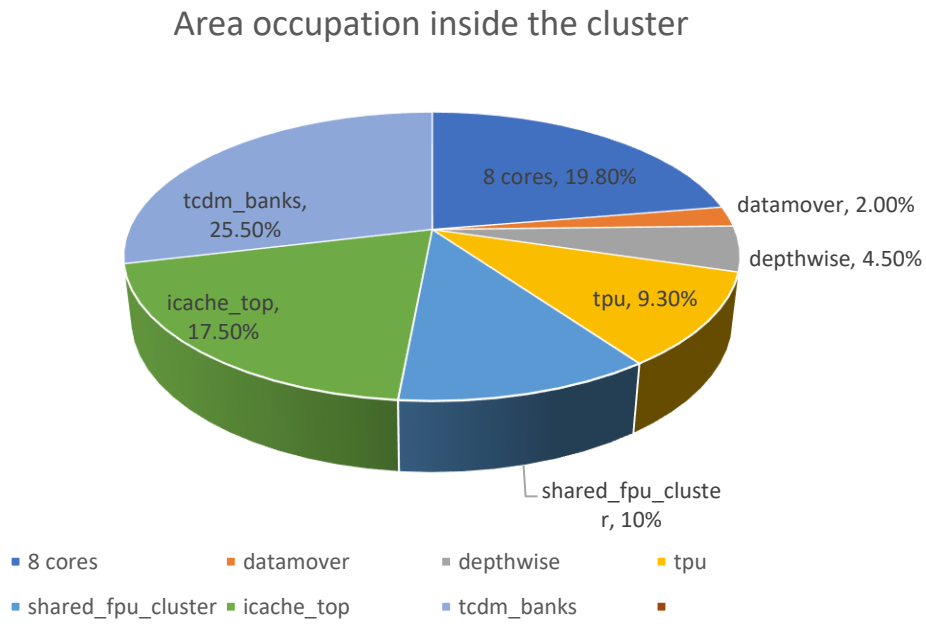

Figure 51: Cluster area occupation

Most of the area occupation is due to the tcdm memory banks, the cache and the eight RISCV cores. Also, the areas of the three accelerators are considered in the cluster, with the Datamover accounting for only 2% of the total area, being slightly smaller than a single core. These results were obtained by synthesizing the design with a 4.5 ns clock period, a supply voltage of 1.08 V and a temperature of 125 °C. The tapeout of the whole chip will happen shortly, with further results in terms of power and timing.

5: Conclusions

In the end, the purpose of designing a small and simple accelerator to perform data marshaling tasks and reordering operations on sub-byte widths has been successful. The strengths of this accelerator surely lie in the ease of configuration and the significant speedup obtained when compared to a not overly optimized software emulation. After removing the tcdm overhead, the accelerator is outperforming the software emulation in almost every supported configuration. Only when choosing to transfer 64 words and not to perform any reordering the software emulation is slightly faster. This is likely due to the configuration overhead of the accelerator.

Besides this, the possibility for 3D stridden data access and the flexible architecture, complete a simple and efficient design for tensor manipulation and layout conversion to be used in image processing applications. These features make the Datamover a valid hardware component for augmenting extreme-edge processing capabilities on IoT nodes.

6: Bibliography

1. Mr. Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing – Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, Luca Benini – 2019.
2. An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics – Francesco Conti, Robert Schilling, Pasquale Davide Schiavone, Antonio Pullini, Davide Rossi, Frank Kagan Gurkaynak, Michael Muehlberghuber, Michael Gautschi, Igor Loi, Germain Haugou, Stefan Mangard, Luca Benini – 2017.
3. A Mixed-Precision RISC-V Processor for Extreme-Edge DNN Inference – Gianmarco Ottavi, Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Luca Benini, Davide Rossi – 2020.
4. ENVISION: A 0.26-to-10TOPS/W Subword-Parallel-Dynamic-Voltage-Accuracy-Frequency-Scalable Convolutional Neural Network Processor in 28nm FDSOI – Bert Moons, Roel Uytterhoeven, Wim Dehaene, Marian Verhelst – 2017.
5. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks – Yu-Hsin Chen, Tushar Krishna, Joel Emer, Vivienne Sze – 2016.
6. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs – Liangzhen Lai, Naveen Suda, Vikas Chandra – 2018.
7. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications – Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, Luca Benini - 2017.
8. Near-Threshold RISC-V Core with DSP Extensions for Scalable IoT Endpoint Devices – Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gurkaynak, Luca Benini – 2017.
9. A Fully-Synthesizable Single-Cycle Interconnection Network for Shared-L1 Processor Clusters – Abbas Rahimi, Igor Loi, Mohammad Reza Kakoee, Luca Benini – 2011.
10. µDMA: An autonomous I/O subsystem for IoT end-nodes – Antonio Pullini, Davide Rossi, Germain Haugou, Luca Benini – 2017.
11. Ultra-Low-Latency Lightweight DMA for Tightly Coupled Multi-Core Clusters – Davide Rossi, Igor Loi, Germain Haugou, Luca Benini – 2014.
12. The Quest for Energy-Efficient I$ Design in Ultra-Low-Power Clustered Many-Cores – Igor Loi, Alessandro Capotondi, Davide Rossi, Andrea Marongiu, Luca Benini.

13. DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs – Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, Francesco Conti- 2020.
14. PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors – Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, Luca Benini – 2019.
15. Scalable Heterogeneous L1 Memory Interconnect for Smart Accelerator Coupling in Ultra-Low Power Multicores – Tobias Riedner – 2020.