POLITECNICO DI TORINO

Master degree course in Electronic Systems

Master Degree Thesis

Development of demo designs for VE2302-DK based on AMD Versal Edge adaptive SoC VE2302

Politecnico di Torino

Supervisor
Prof. Mario Roberto Casu

Candidate
Francesca Franzese
Student ID: 315092

Co-Supervisor
Ing. Maurizio Cignetti

April 2025

Summary

The AMD Versal™ architecture is a heterogeneous compute platform that combines programmable logic (PL), processing system (PS), and AI Engines-ML (AIE-ML), along with leading-edge memory and interfacing technologies. The **Versal™ adaptive SoCs** family delivers powerful acceleration to support a variety of applications in embedded systems.

The work presented in this thesis aims to develop a demo design for **VE2302 Development Kit** by Avnet, which is a board based on AMD Versal Edge adaptive SoC VE2302. This design consists in a Linux-based system running on AI Engine, PS, and PL. The design implements a simple image processing core based on the bilinear interpolation algorithm, which can be used, for example, as a pre-processing stage for AI models.

The AI Engine domain contains a graph consisting of a bilinear interpolation kernel. The PL domain includes data movers between the global memory and the AI Engine-ML array, along with an RTL IP that pre-processes image pixels before they are elaborated by the AI Engine kernel. The PS domain contains a host application that controls the entire system. Despite the final target device being the VE2302-DK, the platform on which this system was targeted and tested is the AMD Versal™ AI Edge Series VEK280 Evaluation Kit. Indeed, the VEK280 board is based on a SoC of the same Versal family (the VE2802), which has the same architecture and a larger number of available resources.

In the first part of the thesis, the two involved platforms are described, and a brief analysis of the architecture of the Versal SoC is provided. This part is followed by a description of the bilinear interpolation algorithm, to explain the implementation choices. The second part describes the individual system components: AI Engine-ML kernels, HLS kernels, and RTL IP in programmable logic, and host application. The focus of this

section is mainly on AI Engine programming. This part also illustrates the integration of an RTL IP in this heterogeneous environment, along with the interfacing rules that must be followed. The interaction between the AI Engine-ML devices and the RTL IP is analyzed by considering two different design approaches.

In the first case, IP and AIE-ML operate sequentially, without direct interaction. The IP fully processes the input frame and writes the output data to DDR before the AIE execution begins. In this approach, the AIE-ML and RTL kernel exchange data using buffers in global memory.

In the second design, the IP and AIE-ML work concurrently. The IP provides data to the AIE array in packets via an AXI-Stream interface, and the AIE processes data between the reception of each packet.

The outcomes obtained from hardware emulation on an x86 host machine and from synthesis and testing on the target hardware are presented. The results are described in terms of performance and resource utilization.

Furthermore, an alternative implementation using only AI Engine-ML is explored, demonstrating the feasibility of a purely AI Engine-based approach and also the limits of this choice.

Overall, this thesis demonstrates the implementation of a bilinear interpolation system on the AMD Versal™ AI Edge platform, integrating AI Engine-ML, programmable logic, and the processing system, considering different design approaches. This work becomes a starting point for the implementation of an image processing system on Versal: the result is a reference design that can be used by the final users to explore the platform's heterogeneity and the powerful performance of the Versal devices.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

In recent years, unprecedented growth in the artificial intelligence market has been witnessed by the investments of millions of companies seeking innovation in various fields by this route. Since 1957, when the first Perceptron algorithm was implemented, the Machine Learning approach has captured the attention of the scientific and engineering world. In particular, Machine Learning aims to obtain better performance from ultra-parallel architectures, such as GPUs, Vector processors, and FPGAs.

Responding to the demands of this market means designing and producing hardware that is able to offer very high computing power and performance and support the increasing complexity of the applications into which artificial intelligence is being integrated. Important are embedded applications, where the system can benefit from various features improved and supported by AI algorithms such as imaging techniques, real-time management of multi-sensor systems, or predictive analysis that make many applications more secure and reliable.

In this scenario, AMD stands as one of the leaders in producing solutions capable of supporting the implementation of AI in various embedded applications. In particular, AMD's response to these requirements is the Versal™ AI Edge series of products, with a portfolio of systems capable of offering more computational power than a simple GPU while maintaining adequate power consumption for embedded applications.

The platforms belonging to the Versal™ AI Edge series are adaptive SoCs (System-on-Chips); The real strength of AMD's solution lies in its heterogeneity, and thus in the presence of:

- Programmable Logic (FPGA), for sensor fusion and maximum flexibility;
- Intelligent Engines, i.e. devices optimized for processing by AI/ML algorithms;
- Scalar Devices, for real-time control in critical applications.

However, new applications exploiting the Versal Edge SoC potential could not be designed without development boards equipped with all the resources necessary to access the chip's capabilities. The development kit consists of an SOM (System-On-Module) and a carrier card with peripheral support. The **VE2302-DK** development kit produced by Avnet Embedded is designed exactly for this purpose: the VE2302 SOM offers developers the flexibility and versatility needed to realize projects with the AMD Versal™ AI Edge series [1]. In particular, the purpose of the work presented in this thesis is the realization of a **reference design** to provide the users a guide to explore and exploit the capability of the hardware.

In this section are presented the elements underlying the development of the reference design, considering the entire hardware and software environment:

- Vitis-AI, the development platform provided by AMD-Xilinx;
- The Versal™-AI Edge series of adaptive SoC, with its special features;
- The VE2302 board, at the center of this project, based on the Versal™ Edge XCVE2302 chip, composed of a SoM and a carrier board.

1.2 Motivations

This thesis's aim is the development of the reference design for the VE2302-DK development kit, describing the realization process. A reference design for an electronic device is an example design provided by the manufacturer to show how to use the device in question. The advantage is that engineers and designers have a tested and optimized starting point to develop their applications. Some advantages that demonstrate the importance of using a reference design are:

- **Reduction of development time**: starting from a working schema, designers do not have to start from scratch but can use the reference design as a basic model for their devices, saving time in the design phase.

- **Performance optimization**: reference designs are usually created by experts who know the component inside out. These designs are optimized for maximum performance and often include recommendations for the best energy utilization, heat management, and minimization of electromagnetic interference.
- **Reduction of errors**: using a design that has already been tested reduces the risk of errors, which can be costly in terms of time and resources during development and production.
- **Compatibility**: a reference design often shows how to integrate different components to ensure compatibility between them, such as connectors, memories, sensors, and communication modules.

In this specific case, considering the complexity and heterogeneity of the Versal SoC components, the presence of a reference design becomes vital for the future designer to ensure a full understanding of the product's potential.

1.3 SoC programming tools

The SoC represents a heterogeneous environment in which different elements work together. Therefore, for the target board programming, several tools are required, combined in the **Unified Vitis™ environment**:

- To program the processing system (PS), the PetaLinux tools are required to generate the Linux image; the Vitis environment is used for the embedded software development flow, compiling the host application that runs on the Arm processor. Also, simulators such as QEMU are present, providing debugging possibilities.
- To configure the programmable logic (PL), Vivado's IP integrator is used to create the block design according to requirements. This can include either classic RTL blocks or HLS components described in C/C++. The Vivado simulator is used for functional verification of the PL behavior.
- To program the AI Engines, the Vitis platform (which includes the AI Engine tools) is used to compile the projects into a graph object. The AIE simulator is used to simulate the graph before running on hardware.

Chapter 2

The VE2302 Development Kit

The purpose of this chapter is the introduction of the VE2302 development kit architecture, which is the target platform of the reference design. The board is presented with a focus on its main parts, i.e. the System-On-Module and the Carrier Board.

2.1 VE2302 SoM

The **VE2302 SOM** offers developers the flexibility and versatility to realize projects with the AMD Versal™ AI Edge series. The SOM is based on the Versal AI Edge SoC XCVE2302. In table 2.1 are summarized the main features and also the interfaces with the carrier card.

2.1.1 AMD Xilinx Versal AI Edge XCVE2302

As mentioned, the SOM is based on the Versal AI Edge SoC XCVE2302-1LSESFVA784-E. As can be read on the official website, the product aims at "high performance, low latency AI inference for intelligence in automated driving, predictive factory, and healthcare systems, multi-mission payloads in aerospace and defense, and a breadth of other applications"[2]. It is, therefore, a product that aims not only to accelerate the AI part but also the monitoring of sensors and real-time controls, speeding up the whole application, while maintaining safety and reliability requirements in critical applications. The characteristics of the SoC are summarised in the table 2.2,

and a block diagram is provided in figure 2.1.

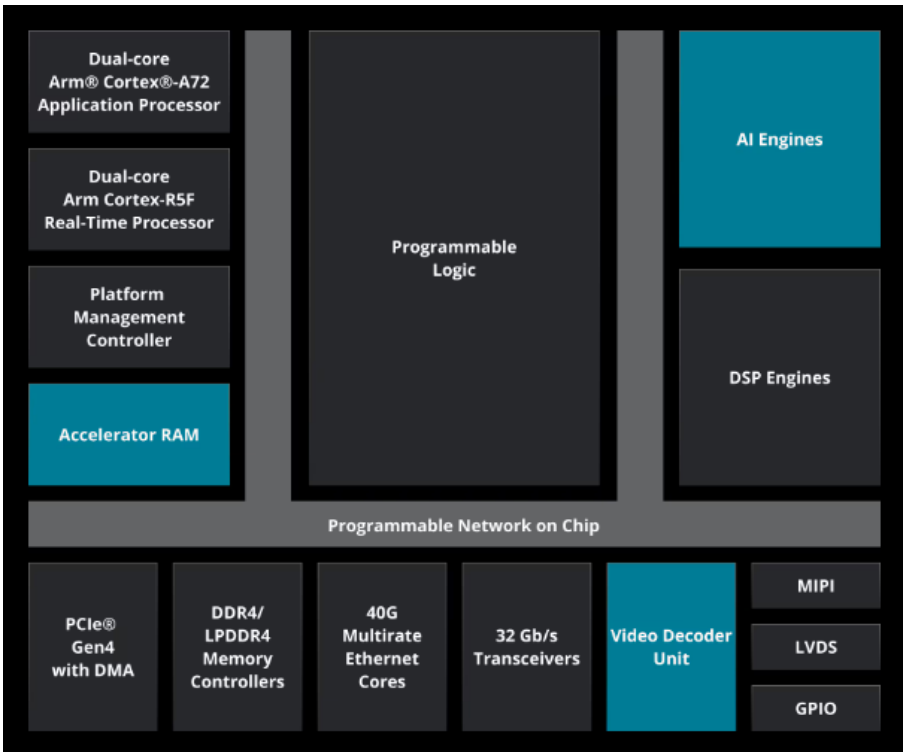

Figure 2.1. Versal AI Edge XCVE2302 Block Diagram[2]

The processing system of the device is based on a **dual-core Arm Cortex-A72**, an application processor used to execute Linux applications, and on a **dual-core Arm Cortex-R5F** (real-time processor) to execute critical code fragments. Flexibility is enhanced by the programmable logic (PL) which allows the integration of different sensors and interfaces. Edge applications are supported by **AI engines-ML** (AIE-ML) and Digital Signal Processors (DSPs), arranged according to an architecture based on an array of vector processors and distributed memories. As can be seen from the diagram, there is also the presence of a RAM accelerator, which has 4 MB of on-chip memory, accessible from all computation engines. This is crucial for AI inference, as it removes the dependency on external memory and enhances performance per watt. Finally, the Versal SoC's programmable I/O allows the connection of different sensors, offering a wide range of speeds and sensors to adapt to both traditional and next-generation standards.
In terms of AI/ML performance, the Versal SoC is capable of achieving the levels summarised in table 2.3.

2.1.2 Platform Management Controller

The Platform Management Controller (PMC) contains several I/O banks that can be connected to various peripherals via the MIO pins of the versal chip. The specific connections of each MIO bank of the PMC are explained below.

PMC MIO Bank 500

This bank consists of 26 MIO pins (MIO[25:0]). The voltage of the bank is +1.8 V. There are three interfaces implemented on the pins of this bank:

- Octal SPI Flash, used for primary boot functionality;
- USB2.0 ULPI, whose physical connector is on the Carrier Card;
- Interrupt general purpose PMC.

PMC MIO Bank 501

This bank also consists of 26 MIO pins (MIO[51:26]), with a supply voltage of 1.8 V. With these pins, various interfaces are implemented:

- SD3.0 for microSD card, to the carrier card, used for both boot and storage;
- eMMC x8 Flash, also to be used as a secondary boot or for storage;
- I2C MAC EEPROM (2Kb);
- I2C 8-Bit I/O Expander (low power);
- I2C 2-Channel Switch/MUX;
- I2C Interface to the Carrier Card: master/slave communication is possible with the I2C devices connected to the carrier card;
- PMBus I2C Interface;
- PMC User LEDs;

LPD MIO Bank 502

Also for this bank there are 26 MIO pins, MIO[25:0]. There are MIO pins of Bank 502 dedicated to implementation. The pins are dedicated to various uses:

- Part of them are dedicated to the implementation of a Gigabit Ethernet interface for a PHY RGMII;
- The rest of the Bank 502 pins are routed to the JX connectors, to enable proper implementation on the Versal AI Edge Carrier Card of different interfaces, such as UART, I2C, CAN, SPI...

PMC Configuration Bank 503

The 503 PMC Configuration bank consists of JTAG, RESET, reference clock input, BOOT MODE, RTC crystal input, and other associated configuration pins. Some of these signals are implemented on the Versal AI Edge SOM, the others are connected to a JX connector for implementation on the Versal AI Edge Carrier Card. The following pins include:

- Pin LED DONE: a BLUE LED indicating that the configuration is complete;
- Pin LED ERROR OUT: turns red in case of problems.
- Some pins of the SOM are used to implement a JTAG interface;
- BOOT Mode pins: there is a small 4-position DIP switch for the pins that sets the boot mode.
- Real Time Clock (RTC): a 32.768 kHz crystal is connected to two of the pins of this bank, and also makes use of the backup battery;
- RESET structure of the SOM. The facility is also equipped with a power line monitoring system, to check when the minimum voltage values have been reached. In addition, there is also a button to manually generate the reset signal for the SOM.
- Reference Clock: one of the pins of the bank is dedicated to providing the SOM with a 33.333MHz (+/-20ppm), single-ended, reference clock input at 1.8 V voltage.

2.1.3 Programmable logic I/O banks

The programmable logic portion of the design consists of several I/O banks that can be connected to various peripheral devices via GPIO programmable logic. In detail, the following connections can be mentioned:

- XPIO Banks 700-701-702: LPDDR4 interface. The SOM is equipped with 4 GB of LPDDR4 memory, whose pins connected to the above-mentioned banks operate at 1.1 V.
- XPIO Bank 702: contains also ab interface to a JX connector (only 4 of the XPIO pins of Bank 702 are used by the LPDDR4 interface);
- Bank HDIO 302: interface towards JX connector;
- Banks 103-104: GTYP Transceiver. The SOM is equipped with 8 GTYP transceivers, supporting line rates from 1.25 Gbps to 32.75 Gbps. They are highly configurable and tightly integrated with the programmable logic resources of the Versal architecture. [5]

2.1.4 Micro Header JX connectors

The Versal AI Edge SOM uses 3 micro header connectors to provide connections to the Versal AI Edge Carrier Card. These connectors have a pitch of 0.635 mm and are rated to support data transmission up to 56 Gbps, with a current rating of 1.4 A per pin.

2.2 AI Edge IO Carrier Card

2.2.1 Main features and block diagram

The Versal Edge Carrier Card is designed to support a single Versal AI Edge SoM. The features are summarised in the table 2.4, while the block diagram can be found in figure 2.2.

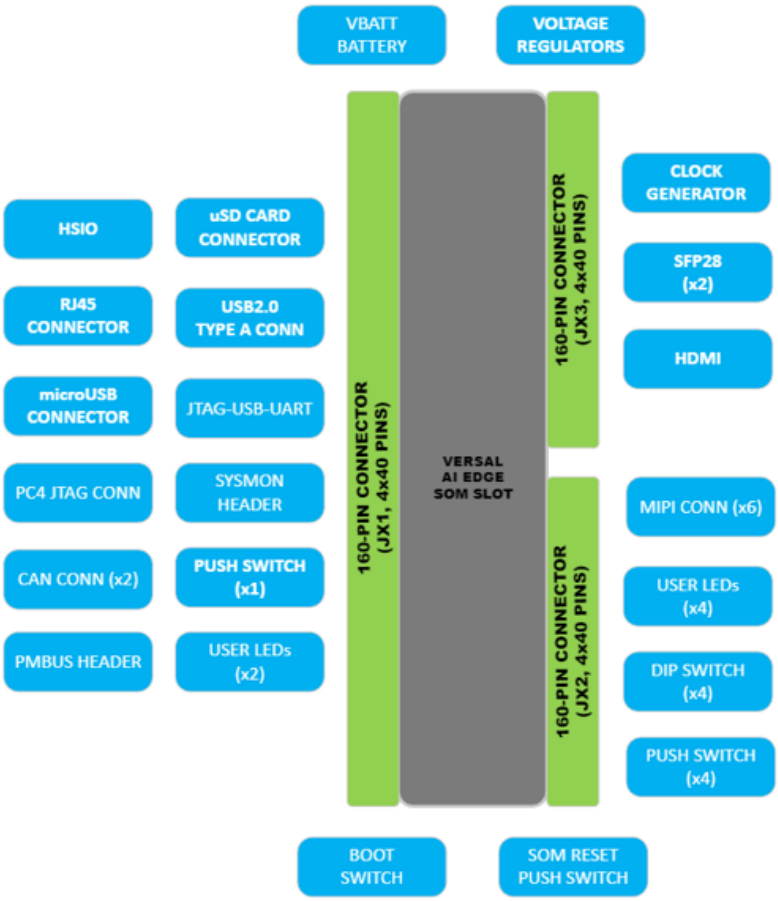

Figure 2.2. Versal AI Edge Carrier Card Block Diagram

SOM Features	AMD Xilinx Versal AI Edge XCVE2302-1LSESFVA784-E Micron LPDDR4 SDRAM (4GB, 2x32) Micron OSPI Flash (64MB up to 256MB) Micron eMMC Flash (32GB up to 64GB) Gigabit Ethernet RGMII PHY USB 2.0 ULPI PHY I2C MAC EEPROM I2C 8-Bit I/O Expander 2-Channel I2C Switch/Mux Reference Clock Real Time Clock Carrier Card JTAG and UART Debug Interface TDK µPOL™ Voltage Regulators 3 Micro-Header Connectors JX1/JX2/JX3 (4x40-pin) Connections to the Carrier Card 104 User XPIO Pins
SOM-to-Carrier Interfaces	22 User HDIO Pins 12 User LPD MIO Pins 13 User PMC MIO Pins 8 GTYP Transceiver 8 GTYP Reference Clock Inputs SYSMON Interface USB 2.0 Connector Interface Gigabit Ethernet RJ45 Connector Interface PMBus Interface Carrier Card I2C Interface SOM VCC-BATT Battery Input SOM Reset Input Carrier Card Reset Output SOM Power Good Output SOM to Carrier Card Ground Pins SOM Input Voltages and Output Sense Pins

Table 2.1. Main features and interfaces of the VE2302 SOM

XCVE2302 SoC features	
AI Engine-ML Tiles	34
DSP Engines	464
System Logic Cells (K)	329
LUTs	150272
Programmable NoC Ports	5
Application Processing Unit	Dual-core Arm® Cortex®-A72
Real-Time Processing Unit	Dual-core Arm Cortex-R5F
Total AI Compute (INT8)	23 TOPS
GTYP Transceivers	8^2
PCIe (PL PCIE4)	1 x Gen4x8
40G Multirate Ethernet MAC	1

Table 2.2. Versal AI Edge XCVE2302 Adaptive SoC features

Operation type	TOPS
AI Engine - INT8x4	45
AI Engine - INT8	23
DSP Engine - INT8	3.2
Programmable Logic - INT4	19
Programmable Logic - INT8	5

Table 2.3. Versal AI Edge XCVE2302 - Performance AI/ML

Versal AI Edge Carrier Card Components	Versal AI Edge SOM Slot 2x SFP28 interfaces 6x 22-pin MIPI-CSI connectors for camera/display HSIO GTYP / HDIO Connector HDMI RX/TX Interface 2x Industrial CAN Header Connectors RJ45 Connector USB 2.0 Connector (Type-A) XPIO Push Buttons XPIO LEDs PMC Buttons GPIO LEDs (connected via I2C) microSD Card Connector microUSB-UART-JTAG Interface PC4 JTAG Header Differential Clock Generator PMBus Header VBATT Battery Connector SOM Reset Button 3x 160-Pin JX Micro-Header Connectors

Table 2.4. Versal AI Edge Carrier Card Features

Chapter 3

The VEK280 Evaluation Kit

The **VEK280 Evaluation Kit** is an evaluation platform based on the Versal AI Edge VE2802 series device. This board is designed for AIE-ML applications, offering higher computing performance, low latency and high level of integration. Its main objective is to enable the evaluation and development of applications based on the Versal AI Edge series [5]. The SoC XCVE2802, which is the device core, belongs to the same Versal AI Edge family of the XCVE2302 presented in the precedent chapter.
Having the physical availability of using this board, the designs presented below have been tested on this board. In fact, the SoC architecture is completely the same: the difference between XCVE2302 and XCVE2802 regards only the number of resources. Therefore, given the compatibility ensured by the fact that the two chips belong to the same family, a design running on VEK280 can also be considered compatible with VE2302 as long as it does not require more resources than those offered by the XCVE2302 SoC. Figure 3.1 shows the VEK280 Evaluation Kit.

3.1 Board features

As anticipated, the board is based on the Versal AI Edge XCVE2802 Adaptive SoC, whose features are summarized in table 3.1. This table provides also a comparison with XCVE2302 features.

The board is equipped with a large amount of resources described in the following:

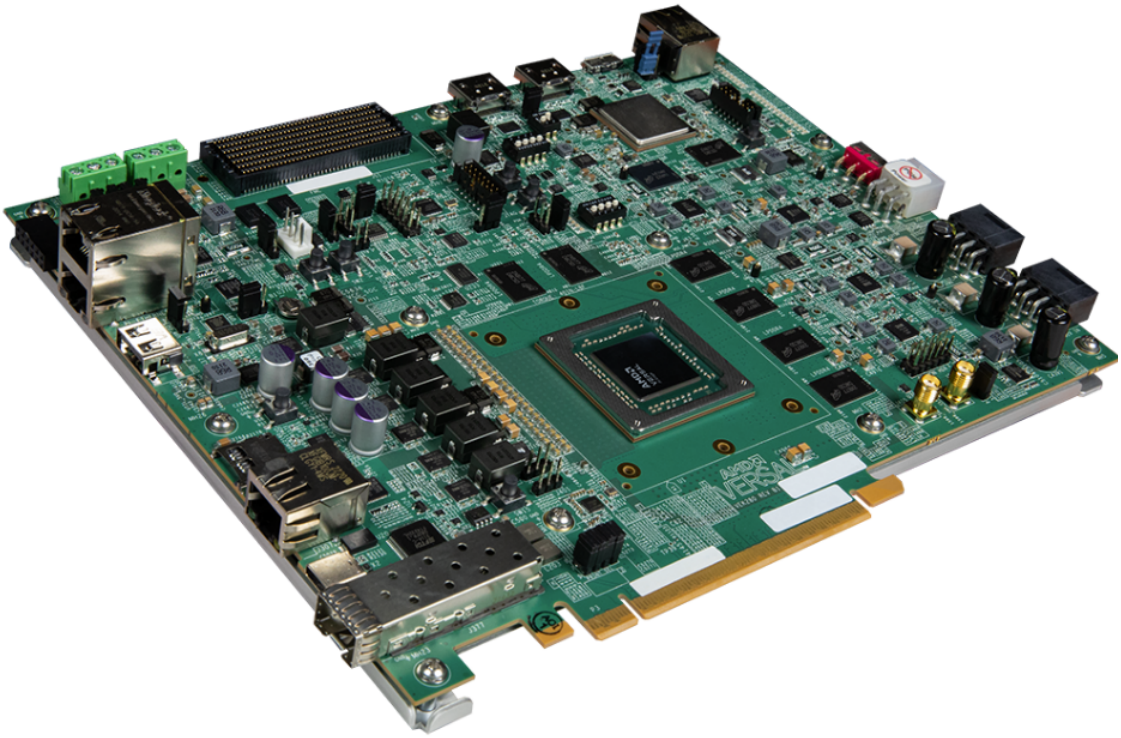

Figure 3.1. The VEK280 Evaluation Kit [7]

- XCVE2802(-VSVH2802 package) SoC ;
- Onboard configuration block, consisting of:
  - USB-to-JTAG bridge;
  - JTAG pod 2 mm 2x7 flat cable connector;
  - Quad SPI (QSPI)/eMMC;
  - microSD card;
  - OSPI (octal serial peripheral interface).
- Different clocks:
  - Versal device bank 702/5/6 RC21008A SYS_CLK_0/1/2 (DIMM) 200 MHz;

Feature	XCVE2302	XCVE2802
AI Engine-ML Tiles	34	304
DSP Engines	464	1312
System Logic Cells (K)	329	1139
LUTs	150,272	520,704
Programmable NoC Ports	5	21
Application Processing Unit	Dual-core Arm® Cortex®-A72	Dual-core Arm® Cortex®-A72
Real-Time Processing Unit	Dual-core Arm Cortex-R5F	Dual-core Arm Cortex-R5F
Total AI Compute (INT8)	23 TOPS	228 TOPS
GTYP Transceivers	8^2	32^2
PCIe Support	1 x Gen4x8 (PL PCIE4)	1 x Gen4x16 (CPM), 4 x Gen4x8
40G Multirate Ethernet MAC	1	2
Video Decoder Engines	–	4

Table 3.1. Comparison of key features: Versal AI Edge XCVE2302 vs XCVE2802 Adaptive SoC

  - Versal device bank GTY205/6 RC21008A_GTCLK1_OUT6/7 100 MHz;
  - Versal device bank GTY106 RC21008A RC21008A_GTCLK1_OUT8 156.25 MHz;
  - Versal device bank GTY106 626L15625 HSDP_156_25_REFCLK 156.25 MHz;
  - Versal device bank GTY204 8T49N241 HDMI_8T49N241_OUT design dependent;
  - Versal device bank GTY204 TMDS1204 HDMI_RCLK_OUT design dependent;
  - Versal device bank 503 RC21008A PS_REF_CLK 33.3333 MHz;
  - Versal device bank 503 RTC Xtal 32.768 kHz.
- Three pin-efficient mode LPDDR4 interfaces (2x32-bit 4 GB components each);

- PL FMCP HSPC (FMC+) connectPL GPIO connections
- PL GPIO connections;
  - PL UART1 to FTDI;
  - PL GPIO DIP switch;
  - PL GPIO LEDs (4 LEDs in total);
  - PL GPIO pushbuttons (2 pushbuttons);
  - PL SYSCTLR_GPIO[0:7];
  - PL 1588_GPIO[0:7, SMA_CLK I/O].
- 32 PL GTYP transceivers (8 quads), used for example for USB-C, PCIe Gen 4, SFP28...
- PS PMC MIO connectivity;
- Security: PSBATT button battery backup;
- SYSMON (System Monitor) header;
- Operational switches (power on-off, boot mode DIP switch...);
- Operational status LEDs (INIT, DONE, PS STATUS, PGOOD)
- Power management;
- System controller (XCZU4EG); [6]

The block diagram of the VEK280 board can be seen in figure 3.2.

3.2 AMD Versal™-AI Edge SoC XCVE2802

The core element of this board is the XCVE2802 SoC, which combines a powerful processing system (PS), programmable logic (PL) and engines for AI/ML application in the same device.
The Versal AI Edge series offers high-performance and low-latency AI inference, as required in various applications such as automated driving, or in factory predictive systems. It is an **adaptive computing acceleration platform** because it combines several processing technologies in a single device, allowing developers to optimise performance according to the specific needs of their applications.

Adaptive acceleration is achieved through resource heterogeneity, as it can be seen on the SoC overview in figure 3.3:

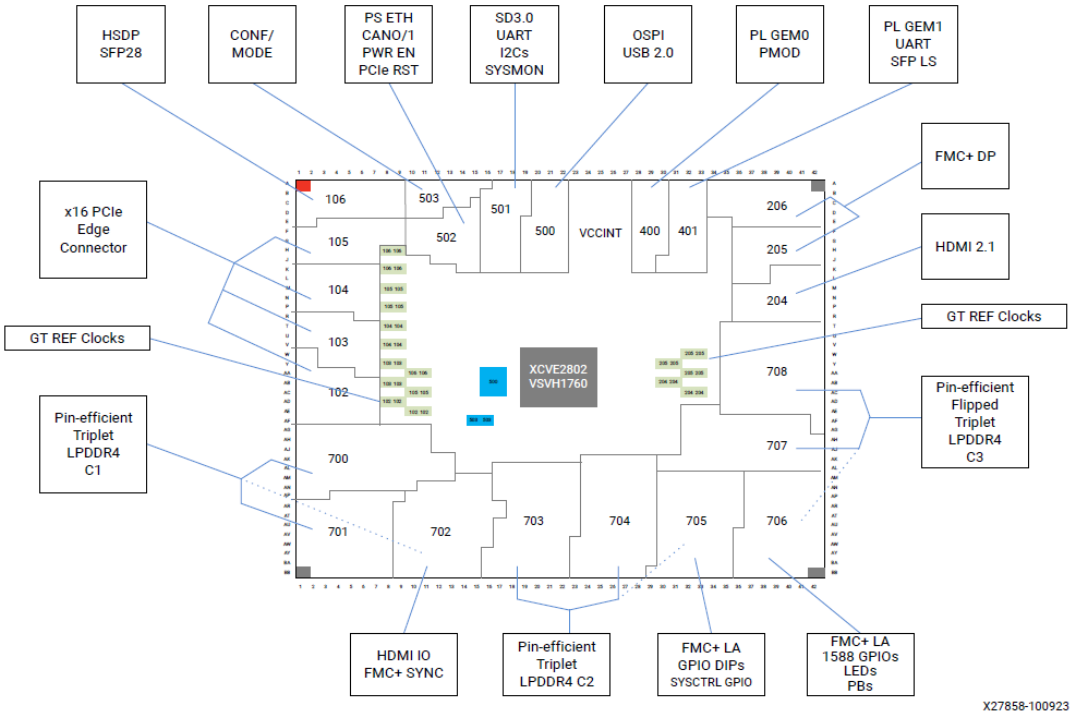

Figure 3.2. Block diagram of the VEK280 board [6]

- **Processor system** (Scalar Engines): the Versal device's processing system is based on the Arm® dual-core Cortex®-A72 processor and the dual-core real-time Cortex-R5F processor. The former is ideal for running Linux applications, while the latter is exploited for running critical code, making it suitable for real-time applications. These processors are commonly used to manage the loading of the various components of the Versal device and to monitor their status.
  The application processing unit, based on the Arm Cortex-A72 processor, is equipped with a system memory management unit, coherent cache interconnect, and interface channels for the remaining systems and peripherals. The maximum working frequency is 1.7 GHz, and is ideal for running applications, with support for Linux or bare-metal environment.
  The real-time processing unit is based on the dual-core Arm Cortex-R5F processor. The main features are low latency, determinism and real-time control.
- **Programmable Logic** (Adaptable Engines): the SoC has millions of programmable logic cells allowing maximum flexibility. They can be

programmed runtime, offering support for pipelined, parallel or hybrid architectures. The system is completed by the presence of several memory cells (LUTRAM, Block-RAM, UltraRAM) and their memory interfaces.

- **AI Engines-ML** (Intelligent Engines): this is the real special feature of Versal SoCs, and will be explored in more detail in section 3.2.1.

The entire technology is based on the TSMC 7 nm technology process.

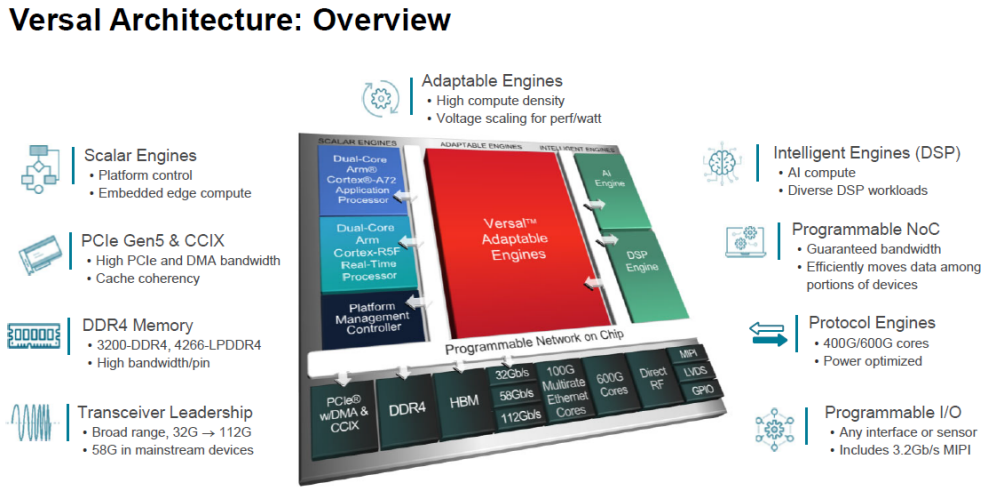

Figure 3.3. Versal Adaptive SoC: overview [3]

The system is also equipped with all the necessary hardware for connectivity and data exchange. The programmable NOC (Network On Chip) ensures adequate bandwidth and high efficiency when moving data through the different parts of the device. As can be seen from figures 3.2 and 3.3, the SoC includes the presence of different programmable I/O peripherals, a DDR4 memory with its respective memory controller, and protocol engines that allow optimization and acceleration of different communication protocols. In addition, high-performance serial transceivers enable data transfer rates from 32 to 112 gigabits per second.

3.2.1 AI Engines-ML

As mentioned above, the special feature of the Versal AI Edge series is the presence of **AI Engines-ML** (AIE-ML), which provide significant speed-ups

for a wide range of applications. The AIE-ML Engines are blocks that offer different levels of parallelism, at the instruction and data level. They are organised in a two-dimensional matrix structure where the AIE-ML tile is the basic unit. In figure 3.4 it is possible to see the diagram of the Versal Adaptive SoC, where the AI Engine-ML array is shown.

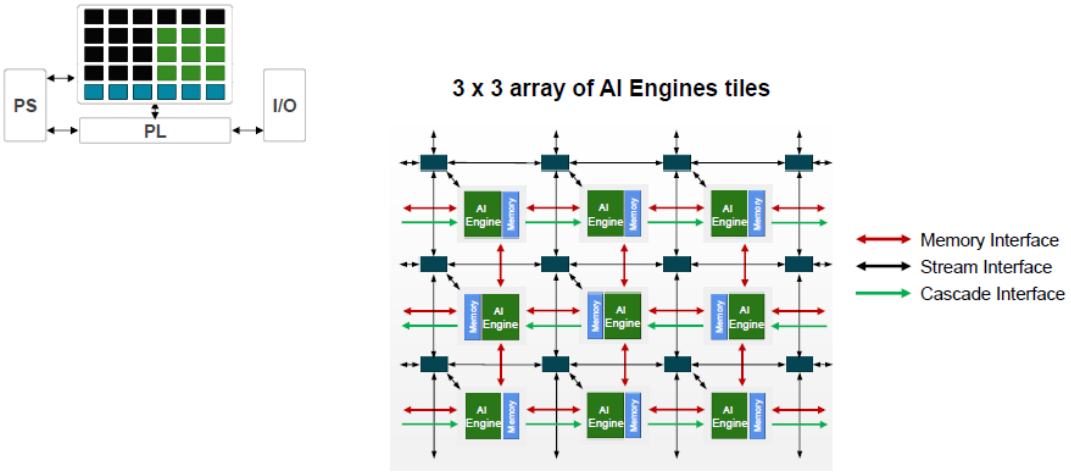

Figure 3.4. AI Engine-ML array [9]

The AI Engine-ML array consists of a 2D array of AI Engine-ML tiles. Each AI Engine-ML tile contains an AI Engine-ML accelerator, memory module, and tile interconnect module. Figure 3.5 shows the top-level block diagram of the Versal device, where it is possible to see the main three component of the SoC: the processor system (PS), the programmable logic (PL) and the AI Engines-ML array.

It also can be noted the presence of a separate functional block, which is the **memory tile**. Each memory tile has 512 KB data memory and is used to reduce PL resources as LUTs and URAMs utilisation in machine learning applications.
In figure 3.4, each black or green box represents an individual AI Engine tile, while the blue boxes represent the interface tiles of the AIE-ML Engine matrix, including interfaces with PL, PS, and peripherals. It is a modular architecture, which means it can scale to include more tiles to meet higher computing needs.

Overall, each AIE-ML tile consists of 3 high-level modules, as can be seen in figure 3.6:

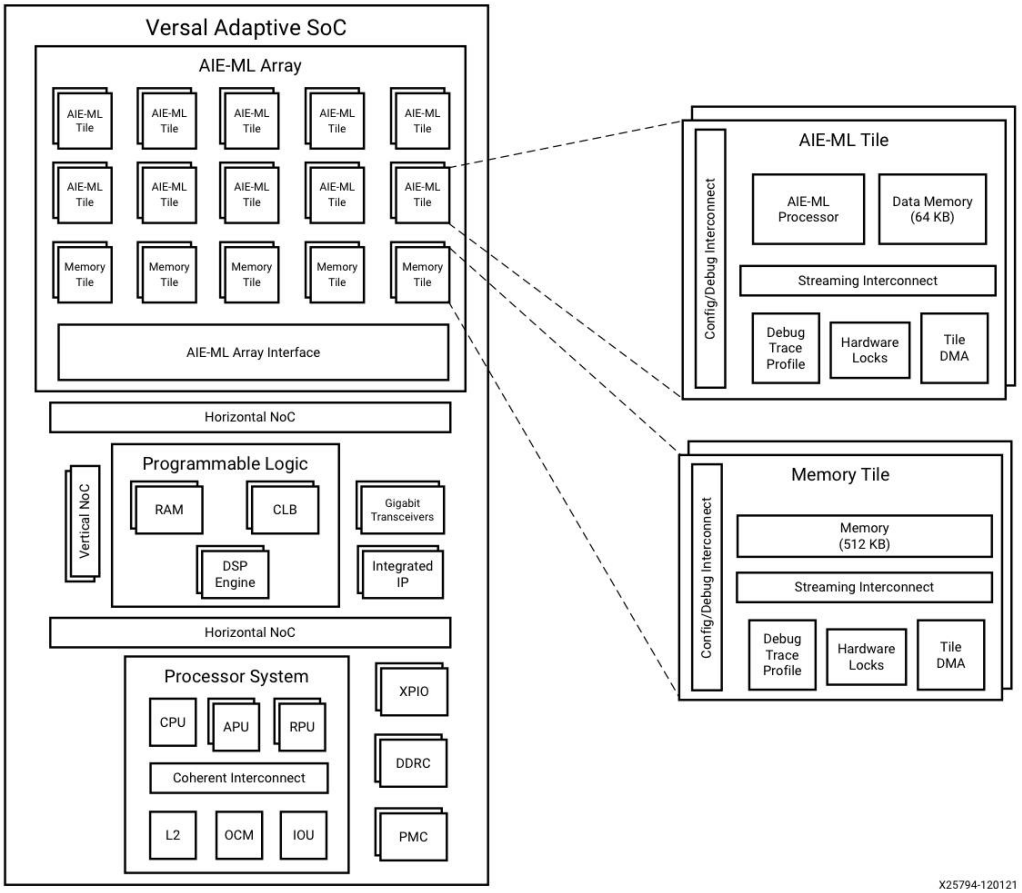

Figure 3.5. AI Engine-ML tile [9]

- The AIE-ML Engine, which is the **ISA-based instruction processor**;
- The **block interconnect module**, handling input/output traffic on AXI streams, memory mapped streams or from/to cascade interconnection;
- The **memory module**: each tile has 64 KB of data memory, divided into eight memory banks, with a memory interface, a DMA for data transfer, and a locks block. The data memory of each tile is shared between its north, south, and west tiles. Therefore, each AI Engine-ML tile can access also the memory module of the neighboring tiles. The connection is provided by an AXI4-Stream switch that is a fully programmable 32-bit AXI4-Stream crossbar (black arrow in figure 3.6).

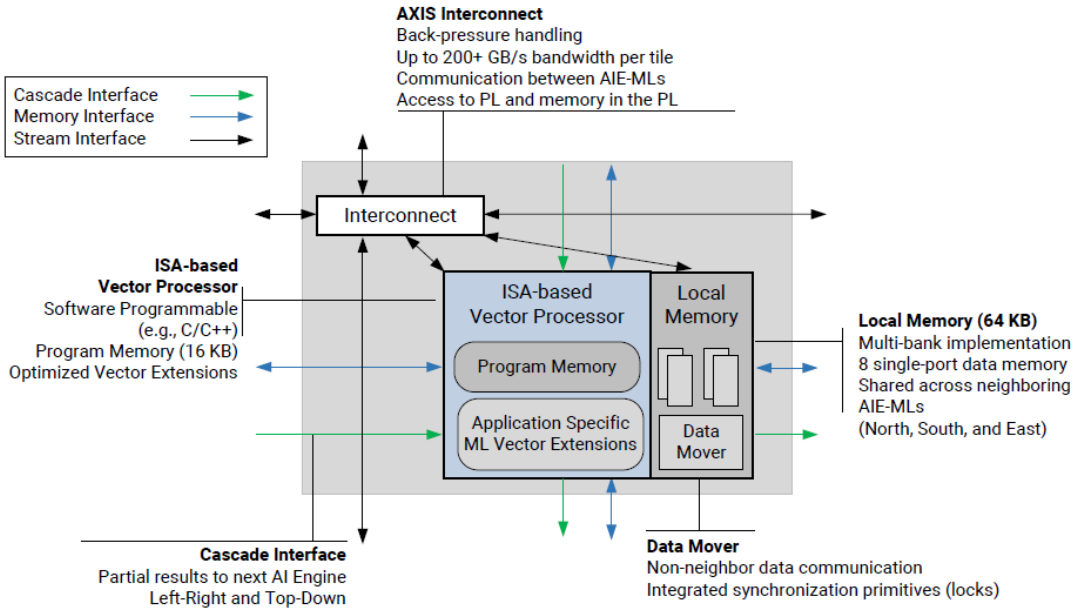

Figure 3.6. AIE-ML Engine tile [3]

By looking inside the engine structure (figure 3.7), it is possible to see that the AI Engine-ML is a highly-optimized processor, **single-instruction multiple-data** (SIMD) and **very long instruction word** (VLIW). It contains:

- A scalar unit, i.e. a 32-bit scalar RISC processor, consisting of scalar registers, special registers, 32-bit multiplier, 32-bit adder/subtractor, and ALU operators;
- A 512 bit vector unit, which supports fixed point and floating point operations. It also supports operations such as FFT and sparsity for ML inference applications, with fixed point and floating point numbers;
- Two 256-bit load units;
- A single 256-bit store unit;
- An istruction fetch and decode unit.

This means that each VLIW instruction support a maximum of two loads, one store, one scalar operation, one vector operation (fixed-point or bfloat16),

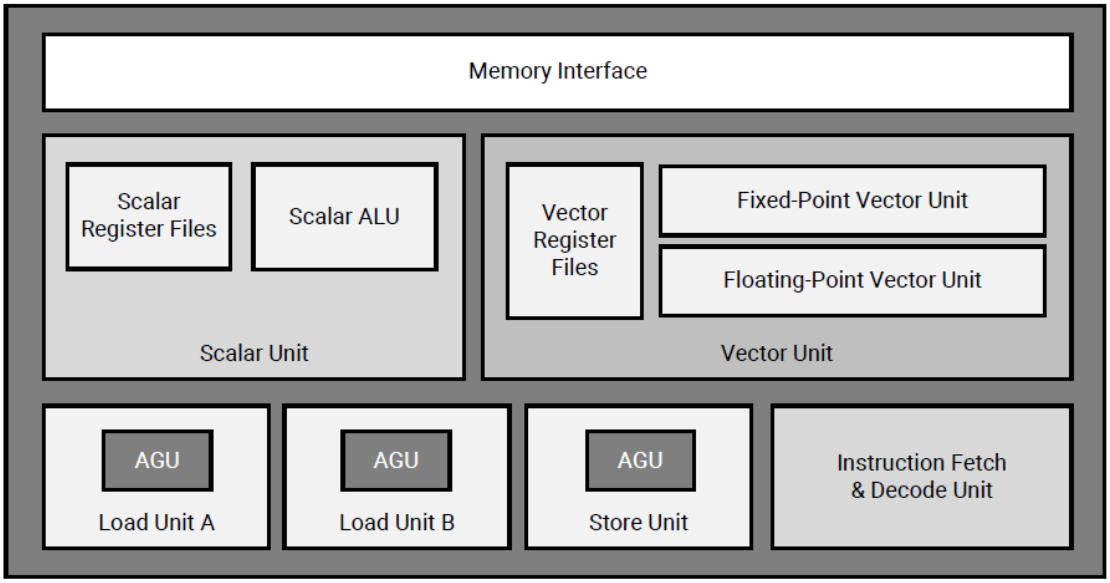

Figure 3.7. AIE-ML Tile [3]

and one move instruction. Therefore, for each clock cycle, a single VLIW instruction corresponding to a maximum of seven operations can be executed.

In addition, each engine has a 16 KB code memory, a stall management unit, a FIFO to accumulate data from the stream, and a debug/trace unit.

In Vitis IDE, there are special Engine tools for compiling and simulating projects involving AI-Engines. The **AI Engine compiler** (aie-compiler) compiles the user code into graphs that describe the algorithm executed on the AIE-ML array and how the AI Engines interacts with PL elements. It also compiles the code executed on each AI engine tile, generating an ELF file, which will be loaded into the code memory of the assigned tile. The **AI Engine simulator** (aie-simulator) simulates the AI Engine kernels running in the AI Engine array. The simulator also includes global memory (DDR) and on-chip network modeling in addition to the AI Engine array, for detailed simulation.

Chapter 4

Implementation of bilinear interpolation algorithm

In this chapter, the realization of the reference design platform has been described, considering all the hardware components. The design of the hardware platform includes:

- Realization of the board definition file (BDF);
- Design of the Vitis platform, which includes the kernels defined in Vivado, the software (baremetal or with Linux operating system support), and the definition of all other resources (such as clock, reset, connectivity...);
- Design of the Vitis Overlay, i.e. the accelerators that will be implemented and executed on the platform, such as the bilinear interpolation kernel on AI Engines.

The fundamental idea is the implementation of a **bilinear interpolation algorithm**, which is an interpolation method for functions of two variables. It is used not only for image processing as in this particular case, but also for finite element analysis, computer graphics, and much more. In the following, an analysis of the algorithm is proposed, which is useful for understanding the implementation choices in this case study.

4.1 The bilinear interpolation algorithm

Bilinear interpolation is an algorithm often used in image processing to improve the quality of images after performing certain operations on them or

to achieve transformations such as rescaling and cropping. A common class of image processing operations is spatial transformations, which redefine the arrangement of pixels on the image plane. These are, for instance, rotations, zooms, and corrections of geometric image defects, such as perspective or radial distortion caused by the lens, as illustrated in figure 4.1.

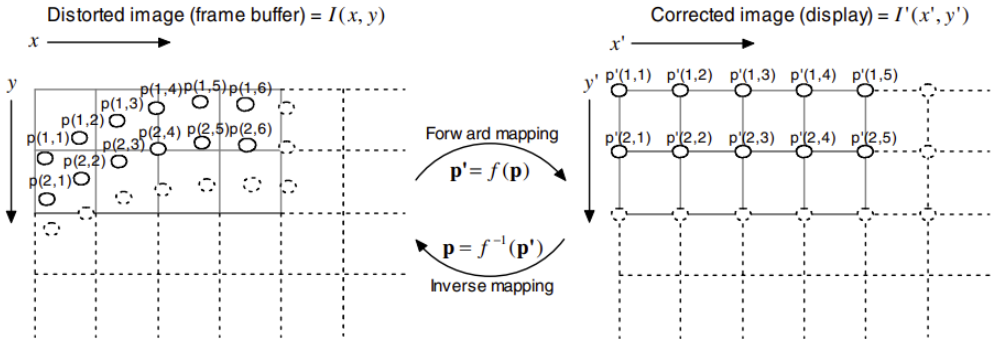

Figure 4.1. Bilinear Intepolation [8]

A spatial transformation can be conceived as a mapping function defined by the following equation:

$$\mathbf{p}' = f(\mathbf{p}) \tag{4.1}$$

where

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \end{bmatrix} \mathbf{p} = \begin{bmatrix} x \\ y \end{bmatrix}$$

p' is the matrix of distorted coordinates, while **p** is the matrix of coordinates of the original pixels. The equation 4.1 is a direct mapping, which calculates the coordinates of pixels in the undistorted image as a function of those in the distorted image. For correction purposes, the inverse mapping function is required to determine which pixel of the distorted image should be output in the corrected grid, f^{-1}. In practice, each pixel in the corrected image is obtained from the corresponding point in the original image.

The coordinates calculated by the inverse mapping function, which are the coordinates of the output image, rarely are integer values: usually, the new positions lie 'between' the pixels of the original image. Simplistic methods such as rounding off or truncation can be used. Still, they introduce significant errors in the pixel position, distorting lines and producing artifacts with jagged edges. Therefore, methods such as bilinear interpolation are used to deal with this problem.

Although bilinear interpolation can introduce artifacts such as blurring or aliasing, it is still widely used in favor of more advanced methods. It is true that algorithms such as bicubic or spline interpolation can produce smoother and more accurate results, but they are also more computationally expensive. Instead, bilinear interpolation is an extremely simple and fast method that represents a good compromise between computational cost and image quality.

From a mathematical point of view, bilinear interpolation is a method for interpolating functions of two variables using **repeated linear interpolations**. The algorithm calculates the value of the 'new' pixel as a weighted sum of the pixel values of the four nearest neighbors surrounding the calculated position, as shown in figure 4.2.

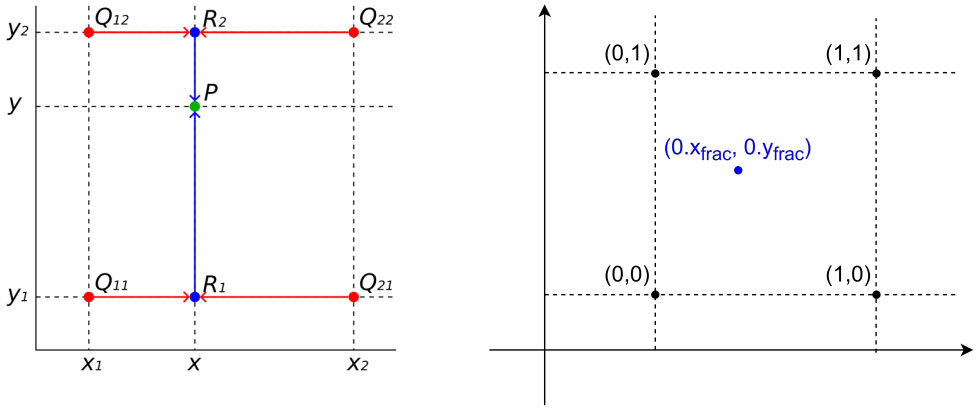

Figure 4.2. Bilinear Intepolation [6]

The calculations involved are very simple. First, interpolation is applied in x-direction, obtaining the equations:

$$R_1 = f(x, y_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \tag{4.2}$$

$$R_2 = f(x, y_2) = \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \tag{4.3}$$

Then, interpolation is applied on y-direction:

$$P = f(x_q, y_q) = \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \tag{4.4}$$

By substituting equations 4.2 and 4.3 in 4.4, the interpolated point is obtained:

$$f(x,y) = \frac{1}{(x_2-x_1)(y_2-y_1)}(f(Q_{11})(x_2-x)(y_2-y) + f(Q_{21})(x-x_1)(y_2-y) + f(Q_{12})(x_2-x)(y-y_1) + f(Q_{22})(x-x_1)(y-y_1)) \tag{4.5}$$

Or, in matricial form:

$$f(x_q,y_q) = \frac{1}{(x_2-x_1)(y_2-y_1)} \begin{bmatrix} x_2-x & x-x_1 \end{bmatrix} \begin{bmatrix} f(Q_{11}) f(Q_{12}) \\ f(Q_{21}) f(Q_{22}) \end{bmatrix} \begin{bmatrix} y_2-y \\ y-y_1 \end{bmatrix} \tag{4.6}$$

In the case of this reference design, the aim is to use this algorithm to interpolate an image of a certain resolution x_{res}, y_{res}. The pixels of the image are assumed to be uniform, as the grid in figure 4.3.

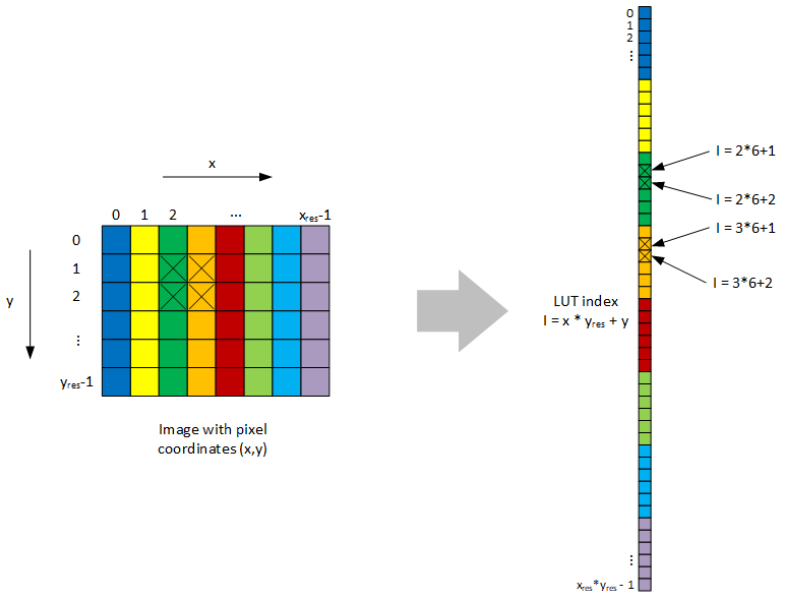

Figure 4.3. Pixel grid [7]

Also it can be assumed that the original image pixels are stored in a LUT by column. Assuming the image is greyscale, individual pixels have values in the range [0,255]. The coordinates of the distorted grid (i.e. the grid corresponding to the new image to be obtained) are expressed as floating-point numbers, in single precision, having an integer and fractional

parts:

$$x_q = x_{int}.x_{frac}$$
$$y_q = y_{int}.y_{frac}$$

In fact, it was said earlier that the coordinates of the new pixels are located "between" the coordinates of the original image.

For processing purposes, only the integer part of each new coordinate (x_q, y_q) can be used to extrapolate from the LUT the 4 points required for the interpolation of each point of interest, according to the equations:

$$\begin{aligned} P_{11} = f(x_1, y_1) &= LUT(x_{int} \cdot y_{res} + y_{int}) \\ P_{12} = f(x_1, y_2) &= LUT(x_{int} \cdot y_{res} + y_{int} + 1) \\ P_{21} = f(x_2, y_1) &= LUT((x_{int} + 1) \cdot y_{res} + y_{int}) \\ P_{22} = f(x_1, y_1) &= LUT((x_{int} + 1) \cdot y_{res} + y_{int} + 1) \end{aligned} \tag{4.7}$$

Once the four points surrounding the computing position have been extrapolated from the original image, only the fractional part of (x_q, y_q) is needed for the calculation. Indeed, the four points define the region within the new pixel value is calculated. The equations 4.2 and 4.3 useful for calculating the point of interest are rewritten by considering only the fractional part of the coordinates (x_q, y_q):

$$\begin{aligned} f(x, y_1) &= x_{frac} f(P_{21}) + f(P_{11}) - x_{frac} f(P_{11}) \\ f(x, y_2) &= x_{frac} f(P_{22}) + f(P_{12}) - x_{frac} f(P_{12}) \end{aligned} \tag{4.8}$$

Same can be done for equation 4.9:

$$f(x, y) = y_{frac} f(x, y_2) + f(x, y_1) - y_{frac} f(x, y_1) \tag{4.9}$$

It can be seen that each of the equations required for the bilinear interpolation of a pixel is a **MAC** (multiply and accumulate) followed by a **MSC** (multiply and subtract).

From the analysis of the algorithm, it is now possible to define all the operations required for pixel interpolation. In summary, for each coordinate point (x_q, y_q) of the output image grid, the calculation of the interpolated pixel value requires two steps:

1. The **identification of the 4 closest pixels** (of the original image) to the one to be interpolated. This operation requires retrieving data from memory and separating the integer and fractional parts of floating-point numbers. These operations suit better implementation in programmable logic.

2. The **interpolation of the point**, by calculating 3 MACs and 3 MSCs. In this case, AI engines can be programmed to perform the calculation efficiently, also taking advantage of the vectorization of multiple operations.

In addition, it is required the implementation of interfaces to allow communication from/to global memory, and the presence of a host application controlling the system. A functional block diagram of the design is provided in figure 4.4.

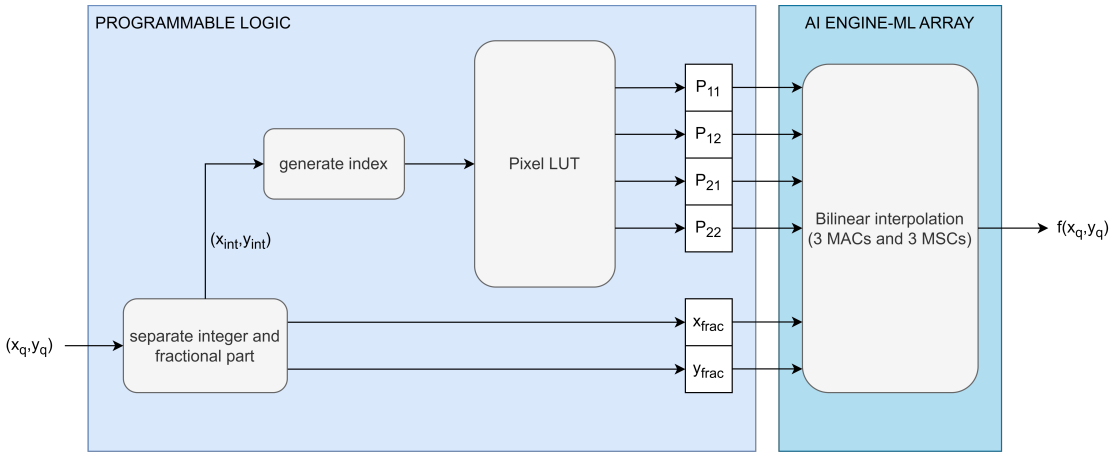

Figure 4.4. Functional block diagram

As an example of a chosen transformation, this reference design uses the bilinear interpolation algorithm on an input image to scale its dimensions. A **scaling factor of 2** is applied to an image of 1024x1024 resolution to obtain an image of 512x512 resolution.

4.2 Bilinear interpolation kernel on AI Engine-ML

In this section, a brief introduction about the fundamental concept of AI Engine-ML programming is provided, to better understand design choices and the elements involved. Then, the code of the bilinear interpolation kernel is presented.

4.2.1 Introduction to AI Engine-ML programming

In section 3.2.1, the AI Engine-ML array architecture has been described. Instead, the purpose of this section is to describe how this structure can be programmed with the tools available in Vitis.
The AI Engine-ML programming is based on **adaptable data flow graph specification** (ADF) written in C++, which can be compiled by the AI Engine compiler. The basic idea is to describe the operations executed on tiles and the data movement in the array using a DFG (Data Flow Graph) representation written in C++ language. The ADF graph application consists of nodes and edges where nodes represent compute kernel functions, and edges represent data connections. The conceptual view of the ADF programming consists of the following items:

- The **AI Engine-ML tile architecture**, introduced before;
- The **AI Engine Kernel**: is a C/C++ program written using the AI Engine API that targets the VLIW scalar and vector processors of an AIE-ML tile. The kernel code is compiled using the aie-compiler, included in the AMD Vitis™ development kit.
- The **AI Engine graph**, described above. An ADF graph is a Kahn process network [1] with a single or multiple AI Engine kernels connected by data streams and/or buffers.

The ADF graph includes interactions not only between AIE-ML tiles but describes also the interactions between the array and the PL, the PS and the global memory. Each interaction requires specific constructs:

- **PLIO** port attribute: used to make stream connections to or from the programmable logic;
- **GMIO** port attribute: used to make external memory-mapped connections to/from the global memory;
- **RTP** (runtime parameters): used by the PS to configure at runtime the graph execution.

[1] A Kahn process network is a distributed model of computation where a group of deterministic sequential processes communicate through unbounded first in - first out channels.

Starting from these considerations, it is possible to describe the AI Engine programming for the bilinear interpolation kernel case. Figure 4.5 illustrates the graph in this specific application from a functional point of view, considering the kernel and also the interaction with programmable logic through PLIO ports.

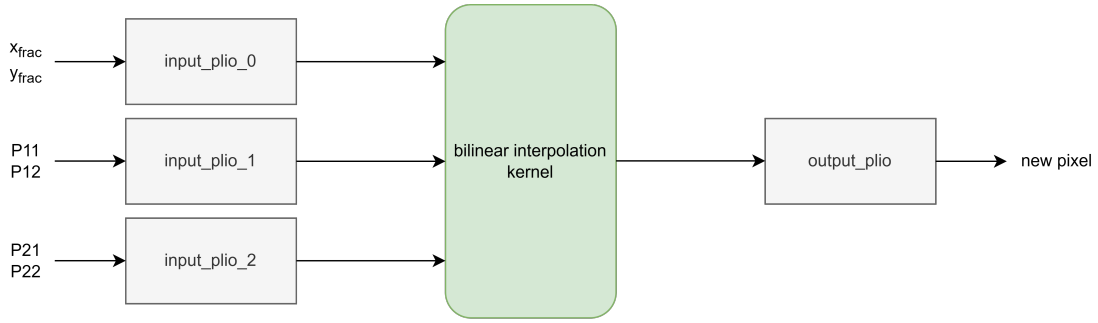

Figure 4.5. Bilinear kernel graph: functional representation

4.2.2 Bilinear kernel graph code

The following code was written starting from a version available in the AMD Xilinx GitHub repository[12], for execution on the VEK190 board. The code has been adapted to run on the VEK280, which is equipped with AIE-ML engines. The original version can be found in Xilinx repository [7]. In fact, the code cannot be compiled on the VEK280 or VE2302 board as it is but requires modifications due to the different architecture of the AI engines on the VEK190 board, the target hardware of these codes. The latter is equipped with AI Engines, which differ from the AI Engines-ML presented above. Therefore, the first part of the work has been studying the APIs supported by the AIE-MLs. The modified code for compatibility with the AIE-ML compilation is presented in this section.

As seen in figure 4.4, the bilinear interpolation kernel requires 6 values per pixel (x_{frac}, y_{frac}, p_{11}, p_{12}, p_{21}, p_{22}), represented as floating point values, coming from programmable logic. For this reason, PLIO interfaces are used to support data transfer between PL and AI Engine array.

Each PLIO interface supports transfer rates of 32 bits per cycle, meaning that one floating-point value per cycle can be transferred. Since for each interpolation are required 6 floating-point values, the design choice is to use 3 PLIO-ports 64-bit wide (2 values per stream), in order to match the

computational efficiency. In this way, ideally are required to 2 cycles to get the required data for each interpolation.

The kernel graph code is shown in listing 4.1.

```
#include <adf.h>
#include "bilinear_kernel.h"
#include "buffers.h"

using namespace adf;

class bilinear_graph : public adf::graph {
private:
  kernel bli_krnl;

public:
  std::array<input_plio, NCORE> iplio_A, iplio_B, iplio_C;
  std::array<output_plio, NCORE> oplio;

  bilinear_graph()
  {
      //...

      iplio_A = input_plio::create(iplio_A_name, plio_64_bits,
      iplio_A_file, 156.25);
      iplio_B = input_plio::create(iplio_B_name, plio_64_bits,
        iplio_B_file, 156.25);
      iplio_C = input_plio::create(iplio_C_name, plio_64_bits,
        iplio_C_file, 156.25);
      oplio = output_plio::create(oplio_name, plio_64_bits,
        oplio_file, 156.25);

      bli_krnl = kernel::create_object<bilinear_kernel>();

      connect(iplio_A.out[0], bli_krnl.in[0]);
      connect(iplio_B.out[0], bli_krnl.in[1]);
      connect(iplio_C.out[0], bli_krnl.in[2]);
      connect(bli_krnl.out[0], oplio.in[0]);

      source(bli_krnl) = "src/bilinear_kernel.cpp";

      runtime<ratio>(bli_krnl) = 0.9;
  }
};
```

Listing 4.1. Bilinear kernel graph code

In the code, the graph class is instantiated by using the objects defined in the **adaptive data flow** (`adf`) namespace. All user graphs are derived

from the `adf::graph` class, which contains kernels and interface definitions. It is possible to see that kernels and interfaces are instantiated using `kernel::create()` and `input_plio::create()` function of the class graph. Also, the connectivity information (i.e. the nets of the ADF graphs) are present. The last parameter is the **runtime ratio**: it is a user-specified constraint that allows the AI Engine-ML tools to put multiple AI Engine kernels into a single AI Engine-ML, if their total runtime ratio is less than one. The runtime ratio of a kernel is defined as:

$$\text{runtime ratio} = \frac{\text{cycles for one run of the kernel}}{\text{cycle budget}} \tag{4.10}$$

The cycle budget is the number of cycles allowed to run one invocation of the kernel, which depends on the system throughput requirement.

4.2.3 Bilinear Kernel algorithm code

The kernel code is the program executed on the VLIW processor. It is transformed into an ELF file by the aie-compiker and it is loaded into the program memory of the tile. The first part of the code is shown in listing 4.2.

```
#include <adf.h>
#include "aie_api/accum.hpp"
#include "aie_api/vector.hpp"
#include <aie_api/aie.hpp>

using namespace adf;

void bilinear_kernel::interp(
input_buffer<int32, extents<BUFFER_SIZE_IN>>& __restrict in_A,
input_buffer<int32, extents<BUFFER_SIZE_IN>>& __restrict in_B,
input_buffer<int32, extents<BUFFER_SIZE_IN>>& __restrict in_C,
output_buffer<int32, extents<BUFFER_SIZE_OUT>>& __restrict out)
{
    // kernel code
}
```

Listing 4.2. Bilinear kernel code

The first thing to notice is that the kernel uses buffered I/O for input and output. Input and output buffers are blocks of data stored contiguously on the tile's physical data memory, accessible by the kernel code. The origin of this data can be either other kernels, or data coming from the PL (as in this

specific case). The interaction between the kernel and the buffer port is the following:

- At input side, a kernel waits for the input buffer to be fully available before starting the execution. This means that the buffer must be entirely filled by data, then the AI Engine processor starts to elaborate.

- At the output, the kernel can write a block of data to the local memory that can be used by other kernels after it has finished execution.

The main advantage of using buffer is a **more efficient VLIW parallelism**, at the cost of an increased initial latency. Another possibility (not exploited in this design), is to use input/output stream of data.

Buffer ports are declared in the prototype of the kernel function by using specific type. For each port, the data type and the dimension of the buffer is specified. In this way, the compiler will place input and output buffers in the tile data memory of size specified by `BUFFER_SIZE_IN/BUFFER_SIZE_OUT` parameters.

The algorithm code is present in listings 4.3 and 4.4, and shows the use of the **AI Engine-ML API** [14]. It is a portable AI Engine kernel programming interface targeting AI Engine-ML architectures. The interface provides parameterizable data types that enable generic programming and implements the most common operations in elaboration algorithms (such as MAC).

Read and write operations on buffer ports are supported by **iterators objects**, as shown in code 4.3. They can iterate over data in a buffer and provide access to individual data. Both scalar and vector iterators are supported in buffer ports. In this case, vector iterators have been used because **vectorization** is exploited to speed up the execution of the interpolation. A vector iterator takes eight elements from the buffer at a time so that eight new pixels are interpolated at each iteration of the for loop. The basic vector registers in hardware are 256-bit wide, so a vector of 8 elements is used.

```
1    // iterators for input & output buffers
2    auto pInA = aie::begin_vector<8>(in_A);
3    auto pInB = aie::begin_vector<8>(in_B);
4    auto pInC = aie::begin_vector<8>(in_C);
5    auto pOut = aie::begin_vector<8>(out);
```

Listing 4.3. Bilinear kernel code - iterators

Each kernel invocation processes a total number of PXLPERGRP pixels, 8 samples at a time, as shown in code 4.4. `chess_prepare_for_pipelining` is a compiler pragma that tells the kernel compiler to do pipelining for the loop. As can be seen, the AI Engine API supports basic arithmetic operations on scalars and vectors, such as addition or subtraction on an accumulator. In this code, the MAC and MSC operations are performed using the function provided by the AIE API. Additional functions are used to convert vectors to accumulators (and vice versa) to ensure function type compatibility.

```
for (unsigned i = 0; i < PXLPERGRP/8; i++)
    chess_prepare_for_pipelining
    chess_loop_count(PXLPERGRP/8)
{
    // get data for first x interpolation
    aie::vector<float, 8> xfrac = (*pInA++).cast_to<float>();
    aie::vector<float, 8> p11 = (*pInB++).cast_to<float>();
    aie::vector<float, 8> p21 = (*pInC++).cast_to<float>();

    aie::accum<accfloat, 8> p11_acc;
    p11_acc.from_vector(p11);

    // compute first x interpolation
    aie::accum<accfloat, 8> tempy1 = aie::mac(p11_acc,xfrac, p21);
    p11= p11_acc.to_vector();
    aie::accum<accfloat, 8> pxy1 = aie::msc(tempy1, xfrac, p11);

    // ...

    // write interpolated pixels to output
    *pOut++ = aie::vector_cast<int32>(pxy.to_vector());
}
```

Listing 4.4. Bilinear kernel code - for loop

An important point to stress is that the AI Engine-ML tile does not support direct floating-point operations. Instead, the AI Engine-ML hardware supports floating-point operations through emulation using the **bfloat16** data type (Brain Floating Point 16-bit).[9] Bfloat16 is a floating-point format widely used in machine learning and AI applications. It is a truncated version of IEEE 754 single-precision (FP32) format, which has a 7-bit mantissa instead of 23-bit in FP32. Bfloat16 allows training with almost no loss in accuracy while being significantly faster than FP32, and is used in applications that can tolerate lower precision in computation. The AI Engine API supports conversion between floating-point accumulator registers and bfloat16 vector registers. If the conversion is not performed and FP32 is

maintained (as in this case), each floating-point MAC has two cycles (instead of one) latency.[2]

In order to run the example, a simple application code is used. It is shown in listing 4.5.

```
#include "bilinear_graph.h"
#include "config.h"

bilinear_graph blint;

int main(void)
{
    blint.init();
    blint.run(NRUN);
    blint.end();

  return 0;
}
```

Listing 4.5. Main application code

This application is used to run the AI Engine graph on the host machine for x86 simulation, using a SystemC based simulator.

- `blint.init()` is used to initialize the graph;
- `blint.run(NRUN)` executes the graph for the number of iteration indicated by the parameter `NRUN`;
- `blint.end()` is used to wait until graph iteration is completed.

The complete code of each presented part of the ADF graph is provided in appendix 9.1.

4.2.4 Bilinear kernel graph compilation

To execute the **aie-compiler**, the following command is used:

```
v++ -c -mode aie -target hw -config ./config.cfg blint.cpp
```

[2] An analysis made considering the bilinear interpolation algorithm with float32 and bfloat16 format is provided in the GitHub repository: `https://github.com/FraancescaFranzese/Vitis-Tutorials/tree/2024.2/AI_Engine_Development/AIE-ML/Design_Tutorials/11_Bilinear_Interpolation`

The aie-compiler allocates the necessary locks, memory buffers, DMA channels, descriptors, and generates routing information for mapping the graph onto the AI Engine-ML array. For each AI Engine core, the compiler creates a main program to schedule all the kernels to be executed on the cores and implements the necessary locking mechanism and data copy among buffers. The C/C++ program for each core is compiled to produce loadable ELF files. In addition, the AI Engine compiler generates control APIs for graph initialization, execution, and termination.

The aie-compilation also produces a summary that can be viewed using the **Vitis analyzer tool** in the Vitis IDE. This summary contains a collection of reports and diagrams showing the compilation results in the AI Engine-ML array. For example, the graph view gives a diagram of the compiled graph, showing allocated buffers and connectivity (see figure 4.6).

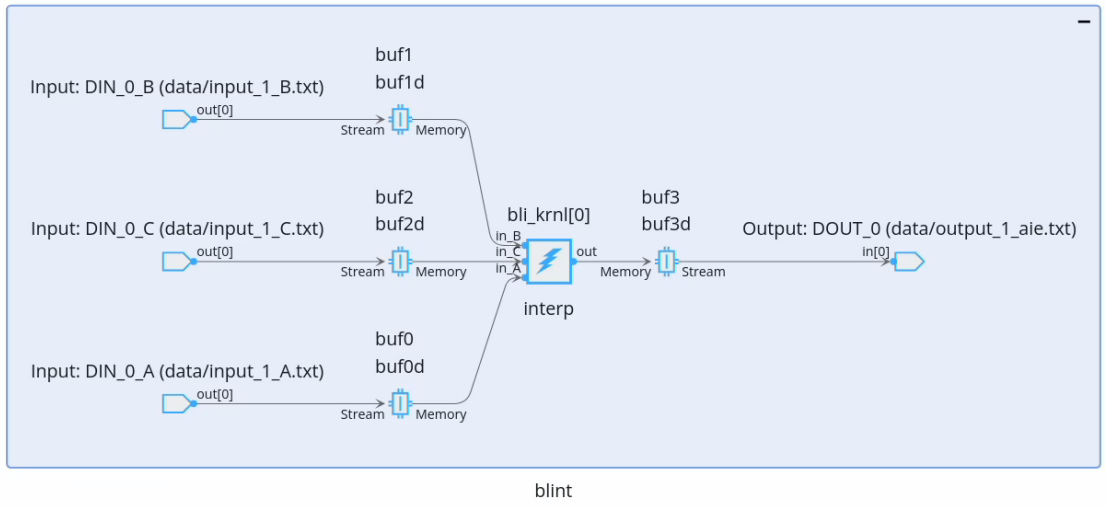

Figure 4.6. AI Engine-ML graph after compilation

It is also interesting to see the **array view** (figure 4.7), illustrating how the design has been mapped on the available physical resources. From this representation, it is possible to see the tiles that will be used for kernel execution and also the allocated buffers.

Looking deeply at the array (figure 4.8), it is possible to observe that tile [18,0] has been allocated for bilinear kernel execution. The buffers, as expected, are allocated in the local memory of the tile, but the data memories of neighboring tiles are also used. At the interface between the AIE-ML array and the PL, four PLIO ports are present.

Figure 4.7. AI Engine-ML array after compilaition

Figure 4.8. AI Engine-ML array with bilinear interpolation kernel - zoom

In figure 4.9, it is possible to see that for each buffer instantiated in the code, a couple of two physical buffer is allocated. In addition, the compiler inserted a couple of **ping-pong buffers** to speed up the algorithm execution.

The arrows in the array are helpful to see data movement.

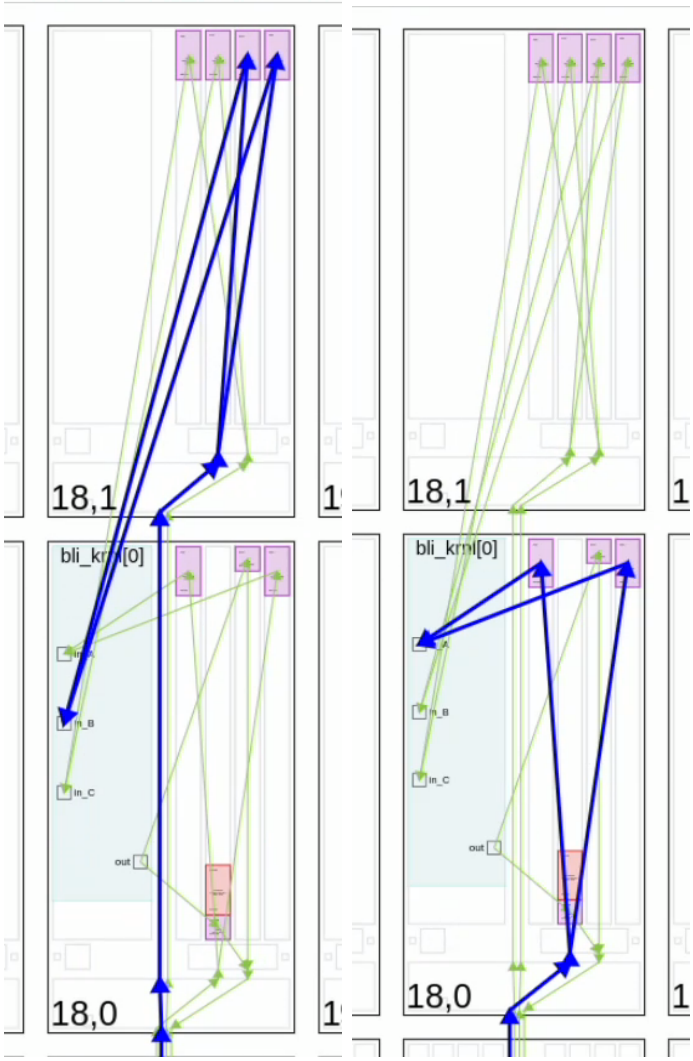

Figure 4.9. AI Engine-ML array - data movement

Graph simulation exploits the SystemC-based simulator integrated into Vitis tool. The control program is the main application already described in listing 4.5. During the simulation, it is possible to collect trace data to display in the Vitis Analyzer, which helps inspect the timing execution of all the graph components. Figure 4.10 shows the trace in this specific case. The **trace view** is useful also to evaluate the performance of the system, measuring the execution time of the kernel, or for debug purposes.

Figure 4.10. Bilinear interpolation kernel trace

4.3 PL kernels requirements

In the Vitis application acceleration development flow, **PL kernels** are the processing elements executing in the programmable logic region of the Xilinx device. The Vitis development flow supports two different kernel types:

- **HLS kernels** written in C/C++ and compiled in Vitis;
- **RTL IPs** designed and packaged in the Vivado Design Suite.

In both cases, kernels are compiled Xilinx object (.xo) files. Therefore, the only difference between the two is how they are designed. Indeed, regardless of the source language, all PL kernels have the same properties and must adhere to the same requirements.

In general, PL kernels can be divided into **software-controllable** and **non-software-controllable**. This means that the kernel is controlled through software, such as the host application, or is unmanaged by software and instead data-driven.

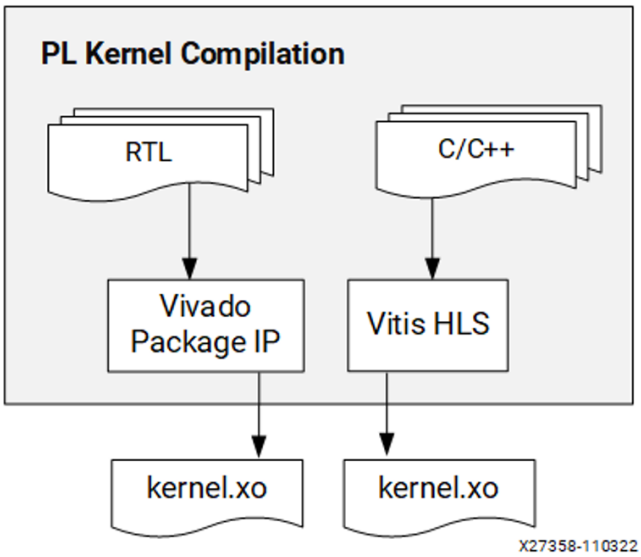

Figure 4.11. Vitis PL kernels development flow

Software-controllable kernels expose a programmable register interface to the processing system, allowing a host software application to interact with kernels through register reads and writes. These are the most common and widely applicable types of kernels. It is recommended that the designer

only develop kernels belonging to this category. Instead, data-driven kernels are automatically instantiated by the compiler where they are needed but are not directly accessible by the software application. They do not have a programmable register interface and must have at least one AXI4-Stream interface. In this design, only software-controllable kernels have been used. They are divided into two categories:

- **XRT-managed kernels**. In this case, the software application communicates with the XRT-managed kernel using higher-level commands such as set_arg, run, and wait;
- **User-managed kernels**. The software application communicates with the user-managed kernel using atomic register reads and writes through the AXI4-Lite interface.

Consequently, XRT-managed kernels are recommended for C++ developers, where the user does not need to know the low-level details of the programmable registers and kernel execution protocols. Alternatively, user-managed kernels can support many different user-defined execution protocols as found in existing Vivado RTL IP and so are a better fit for RTL designers described in Packaging RTL Kernels. The main differences between the two are summarized in table 4.1.

XRT-managed kernels	**User-managed kernels**
• object class is `xrt::kernel` • software application communicates with high level commands • no need to know the low-level details of the kernel • recommended for C++ developers	• object class is `xrt::ip` • software application communicates using atomic register reads and writes through the AXI4-Lite interface • low-level details knowledge required • better fit for RTL designers

Table 4.1. Software controllable kernels

RTL IP from the Vivado Design Suite can be packaged as kernels

(or compiled Xilinx object (.xo) files) that can be linked into an FPGA executable (.xclbin) as long as they adhere to Vivado IP Packaging guidelines.

An RTL kernel must adhere to specific requirements to enable the Vitis compiler to connect kernels to the target platform. Requirements include:

- Language: kernels can be developed using either HDL or C/C++;
- Hardware interfaces: a single AXI4-Lite slave interface, any number and combination of AXI4 memory mapped and AXI4-Stream interfaces, and clock and reset signals;

Hardware interface requirements are summarized in table 4.2.

Ports or interfaces	**Requirements**
clock	at least one clock input is required, and must be packaged as a bus interface.
reset	optional port. If present, must be associated to a clock signal.
interrupt	optional port. If present, active high.
s_axi	required, and only one is allowed.
m_axi	AXI4 Memory mapped interface. It is optional port. If present, it must have 64-bits addresses, and must not use Wrap or Fixed burst types.
axis	AXI4 stream interfaces, optional but necessary to transfer data between kernels.

Table 4.2. Software controllable kernels

In addition, if the IP is packaged as an xrt-managed kernel (instead of user-managed), the designer must provide the software with a specific register map. Indeed, the control interface is managed through the XRT API `xrt::ip` kernel class, which expects a certain register interface.
The kernels integrated within this specific reference design are:

- `mm2s` and `s2mm` interfaces, HLS components handled as **XRT-managed kernels**. They are used to provide exchange of data between PL and AIE-ML array;
- `pixel_reorder` RTL IP, a **user-managed kernel** which provides pre-elaboration of the original image to create, for each pixel interpolation, the set of data x_{frac}, y_{frac}, p_{11}, p_{12}, p_{21}, p_{22}.

4.3.1 HLS interfaces from/to global memory

Two ways can be exploited to provide data to the AI engine array: the **PL streaming interface** or GMIO (i.e., external memory-mapped connections to or from the global memory). The GMIO interface has a low throughput and is, for this reason, ideal for configuration. An efficient way to provide data streaming to AIE-ML array is to implement additional interfaces in programmable logic:

- **mm2s**: Memory Map to Stream HLS kernel to feed input data from DDR to AI Engine kernel via the PL DMA;
- **s2mm**: Stream to Memory Map HLS kernel to feed output result data from AI Engine kernel to DDR via the PL DMA.

Vitis library provides some example code that can be used to include easily HLS kernel in user designs. The code of mm2s interface is provided in listing 4.6.

```
1  /*
2  Copyright (C) 2023, Advanced Micro Devices, Inc. All rights reserved.
3  SPDX-License-Identifier: X11
4  */
5
6  #include <ap_int.h>
7  #include <hls_stream.h>
8  #include <ap_axi_sdata.h>
9
10 extern "C" {
11
12 void mm2s(ap_int<32>* mem, hls::stream<ap_axis<32, 0, 0, 0> >& s,
13           int size) {
14
15 #pragma HLS INTERFACE m_axi port=mem offset=slave bundle=gmem
16
17 #pragma HLS interface axis port=s
```

```
#pragma HLS INTERFACE s_axilite port=mem bundle=control
#pragma HLS INTERFACE s_axilite port=size bundle=control
#pragma HLS interface s_axilite port=return bundle=control

    for(int i = 0; i < size; i++) {
        #pragma HLS PIPELINE II=1
        ap_axis<32, 0, 0, 0> x;
        x.data = mem[i];
        s.write(x);
    }
}
}
```

Listing 4.6. mm2s HLS code [10]

From the code, it is possible to describe some relevant information:

- The header `<hls_stream.h>` provides the `hls::stream<>` class, used for AXI4-Stream communication.`ap_axi_sdata.h` define `ap_axis<>`, a structure used to send data over an AXI4-Stream;
- In the function declaration, `extern "C"` ensures C-style linkage so that the function can be called from PL, although it is a non-C++ environment.
- There are three function arguments. The first is a pointer to 32-bit integer memory, representing the input data; the AXI4-Stream output, which will send to the AI Engine stream the data read from memory; the "size", i.e. number of elements to read from memory and to send on the stream.
- `#pragma HLS` directives give the compiler specific instructions for hardware generation, starting from the C code.
- A for loop is used to read size elements from memory `mem[i]`. Each value is stored in an `ap_axis<32, 0, 0, 0> structure`. Ultimately, the value contained in this structure is written in the AXI4-Stream (s.write(x)).
- `#pragma HLS PIPELINE II=1` force the compiler to implement a one-stage pipeline, ensuring one iteration per clock cycle.

A similar code is used to implement s2mm interface, to write data from AI Engine output stream to the global memory. The code is provided in appendix 9.2.

4.3.2 Integration of RTL IP in programmable logic

From the algorithm analysis, it is clear that the bilinear interpolation of a pixel requires a set of 6 data: x_{frac}, y_{frac}, p_{11}, p_{12}, p_{21}, p_{22}. A **user-managed RTL IP** is used as a pre-processing block, implementing the first part of the bilinear interpolation operation. This kernel corresponds to the block represented in the programmable logic area of the functional block diagram in figure 4.4.

Assuming that the coordinate grid of the output image and the matrix of the original image pixels are stored in global memory, the implemented algorithm is the following:

1. The first row of the output coordinate matrix is read from global memory through the AXI bus and stored in a FIFO after a read request of the AXI-master interface. This means that, for each iteration, `Y_RES_OUT` (output image vertical resolution) values are read from memory. Each (x_q, y_q) value is the starting point of the pre-elaboration. Notice that a DMA mechanism has been implemented to read and write data from/to the global memory.

2. First (x_q, y_q) query coordinate is read from the FIFO. For the sake of simplicity, floating point coordinates are converted into fixed point values, as it is easier to separate integer and fractional parts.

3. `x_int, y_int` are used to calculate the memory address corresponding to the rows of the original image where p11, p12, p21, and p22 are located.

4. The selected rows are read from memory and stored in two line buffers in programmable logic. Two consecutive lines are read because the square of points of interest lies on two adjacent rows of the image. For this reason, two line buffers are present in the datapath to store the two lines for a total of `YRES*2` floating point values for each line.

5. The fractional parts `x_frac, y_frac` are used to calculate the index of the four neighboring pixels inside the two line buffers.

6. Once the four neighboring pixels have been obtained, they are temporarily stored inside some output FIFOs. They will then be sent to the global memory or directly to the AI Engine array PLIO ports.

7. When all the (x_q, y_q) coordinates in the input FIFO have been processed, getting the four neighboring pixel sets for each of them, this process repeats until the entire grid has been pre-elaborated.

This kernel has been designed in two versions to fit into two different designs that will be proposed in the next chapter as possible implementations of the bilinear interpolation algorithm. In both cases, each set of data is stored inside 3 FIFOs in PL, and then:

- In the first version, the pre-elaborated data stored in the output FIFOs are stored in buffers instantiated in global memory using DMA write operations. These store operations occur at the end of the processing of each full row of the output image grid - that is, once all six data sets for the `Y_RES_OUT` pixels have been selected. The top-level datapath of this first IP is provided in figure 4.12.
- In the second version, the data in the FIFOs is directly provided to the AI Engine PLIO ports thanks to three AXI4-Stream interfaces. Each data transmission towards the AI Engine consists of a 256 data packet, sent as soon as the FIFO has the required amount and if the AI Engine is ready to receive. The datapath is provided in figure 4.13, where it is possible to notice the additional presence of AXI4-Stream interfaces. [3]

In both datapaths, it is possible to notice the presence of the **Pixel Reorder** block, which is the RTL part performing the pixels pre-processing.

Pre-processing pixel reorder kernel datapath

The datapath of the pixel reorder kernel is shown in figure 4.14. Some relevant elements can be described.

- `FIFO`. The input FIFO is used to store the pixel coordinates (x_q, y_q) of the output image loaded from memory. The output FIFOs are used to store selected pixels. The first fifo contains `x_frac, y_frac`, the second `P_11, P_12` and the third stores `P_21, P_22`. An important

[3] AXI4-Stream is a protocol designed for arbitrary unidirectional data exchange. In AXI4-Stream, `TDATA` bits are transferred per clock cycle. The transfer starts when the producer sends the `TVALID` signal, and the consumer responds by sending the `TREADY` signal (once the initial TDATA has been consumed). At this point, the producer will start sending `TDATA` and `TVALID`. The consumer keeps consuming the incoming data until `TLAST` signal is asserted.

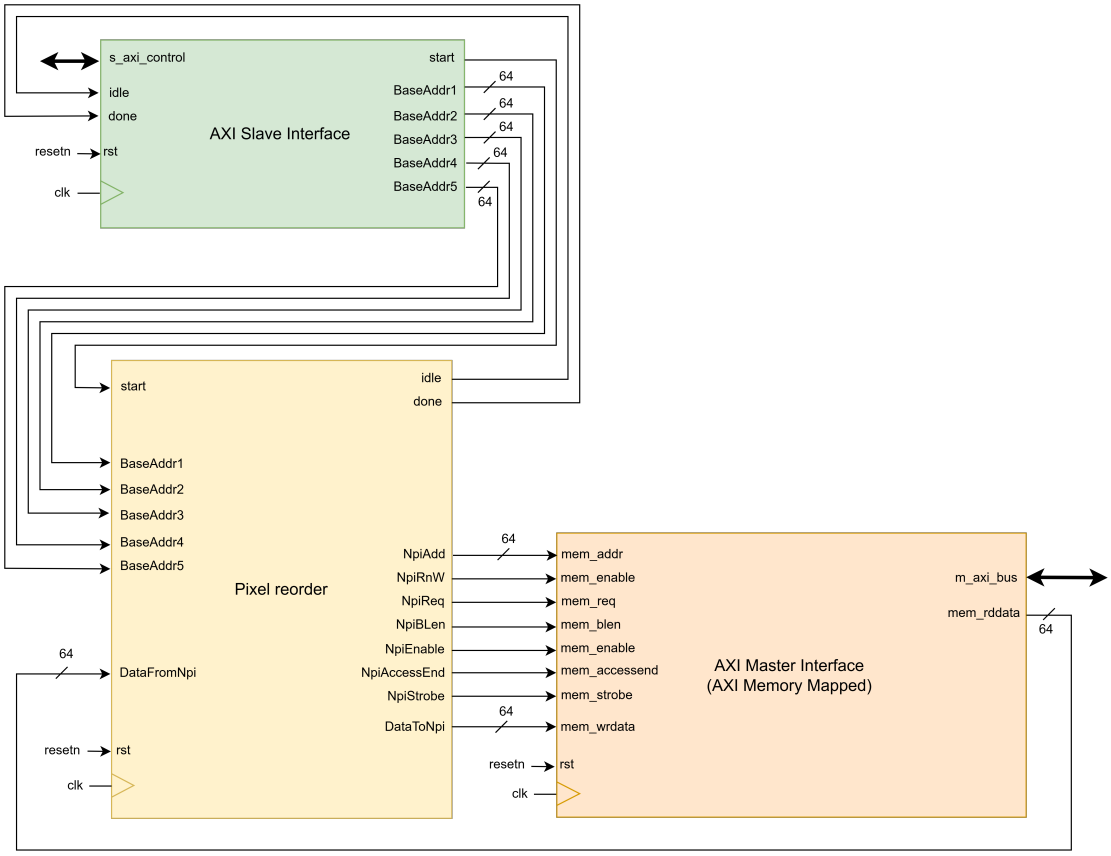

Figure 4.12. Design 1 - top level datapath

point to notice is that the data stored in the FIFOs are arranged to consider the vector calculation within the AI Engine. Indeed, the bilinear interpolation kernel expects to receive vectors of 8 consecutive data to perform vector operations. Figure 4.15 shows the data placement in the output FIFOs.

- `BRAM`. Two Block RAMs used as line buffers store the two consecutive rows of the original image, selected from memory according to the (x_q, y_q) values. Each block has been dimensioned to contain one row, i.e. 1024 elements of 32 bits each. Input data is 64 bits in order to match the AXI-bus width. For this reason, each received data contains two pixels.

- `Memory pointer generator`: this block generates the memory addresses to retrieve the image rows from DDR, (`MemPtrImg`), or generates the address to read the next set of (x_q, y_q) values from memory

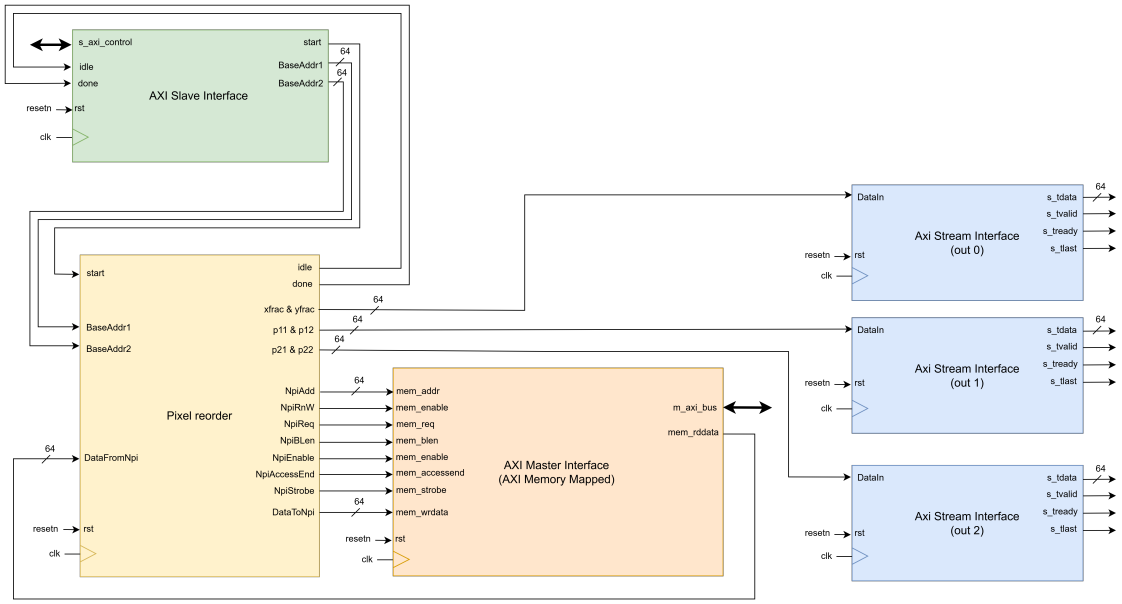

Figure 4.13. Design 2 - top level datapath

(`MemPtrXqYq`).

- `Splitter`: this block separates integer and fractional part of (x_q, y_q).
- `Fixed to float`. This block uses the Xilinx Floating-Point operator, an IP capable of being configured to provide a range of different floating-point operations. In this case, the IP has been set to convert fixed point 16.16 numbers into single precision floating point numbers on 32 bits.
- `X counter`: counts the output image's processed rows. The FSM uses this value to decide when the entire image has been pre-processed.
- `Y counter`: counts the processed pixels in a row of the input image. The FSM uses this value to determine when new data must be read from memory.
- `Burst counter`: counts the number of bursts received from DDR. This is necessary because the maximum number of 64-bit elements allowed for each burst transfer is 256. This means that more than one burst is needed to receive an entire line of the image. In this case, AXI Bus is 64 bits wide, so two elements are transferred each cycle. Since each row has

1024 pixels, two burst transfers are required to receive an entire image line from memory.

- `FSM`. A state machine coordinates read/write operations to/from memory and generates the commands required by the datapath elements.

Pre-processing pixel reorder kernel FSM

The FSM block consists of two different Moore's finite state machines:

- The first FSM is mainly used to generate the commands required by the datapath elements of the pixel reorder kernel. Figure 4.16 illustrates the FSM flow chart for the first IP version. Figure 4.17 provides the flow chart for the second case. The state machines are very similar; the only difference is that the first design also requires the management of writing operations of the obtained data to the buffers in the global memory. This part is not present in the FSM of the second design since the data is sent directly to the AI Engine via AXI4-Stream.
- The second FSM is mainly used to interact with the AXI-Master interface, generating the commands required for DDR reads and writes. This second machine starts the execution when `start_read` or `start_write` signals are set to one by the first FSM. The flow chart is shown in figure 4.18.

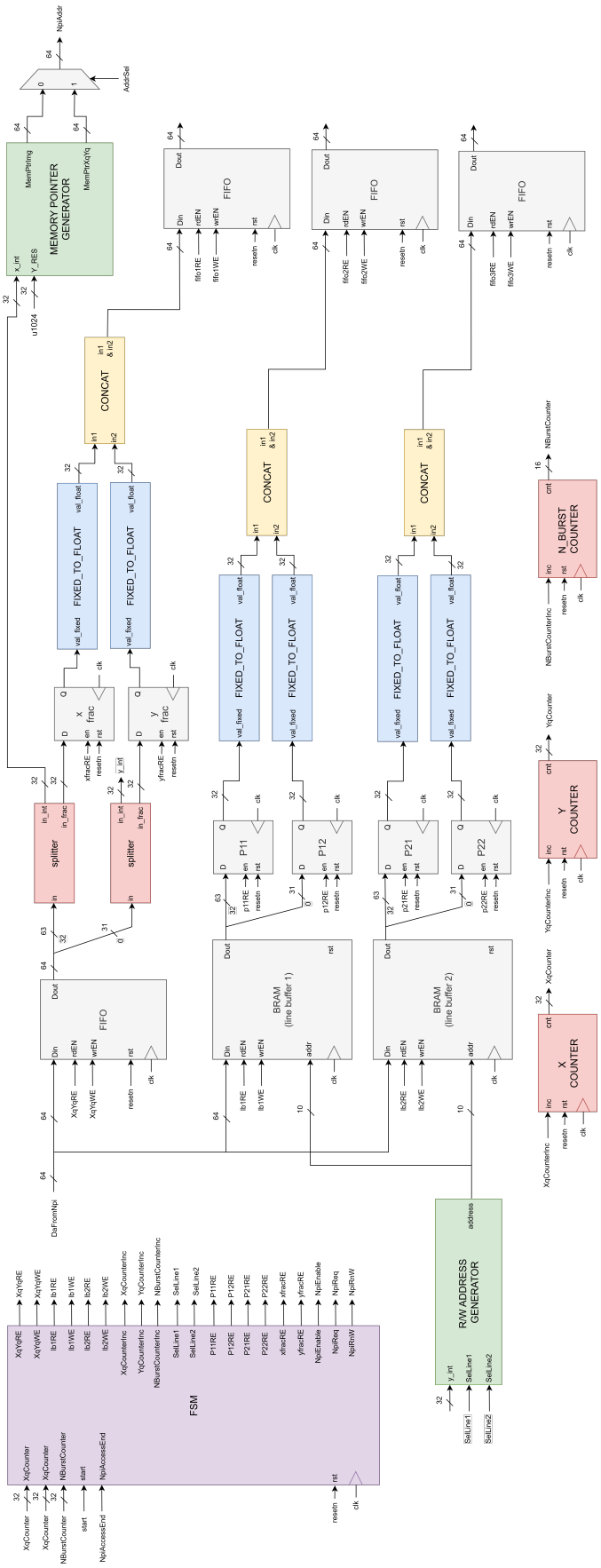

Figure 4.14. Pixel reorder RTL kernel datapath

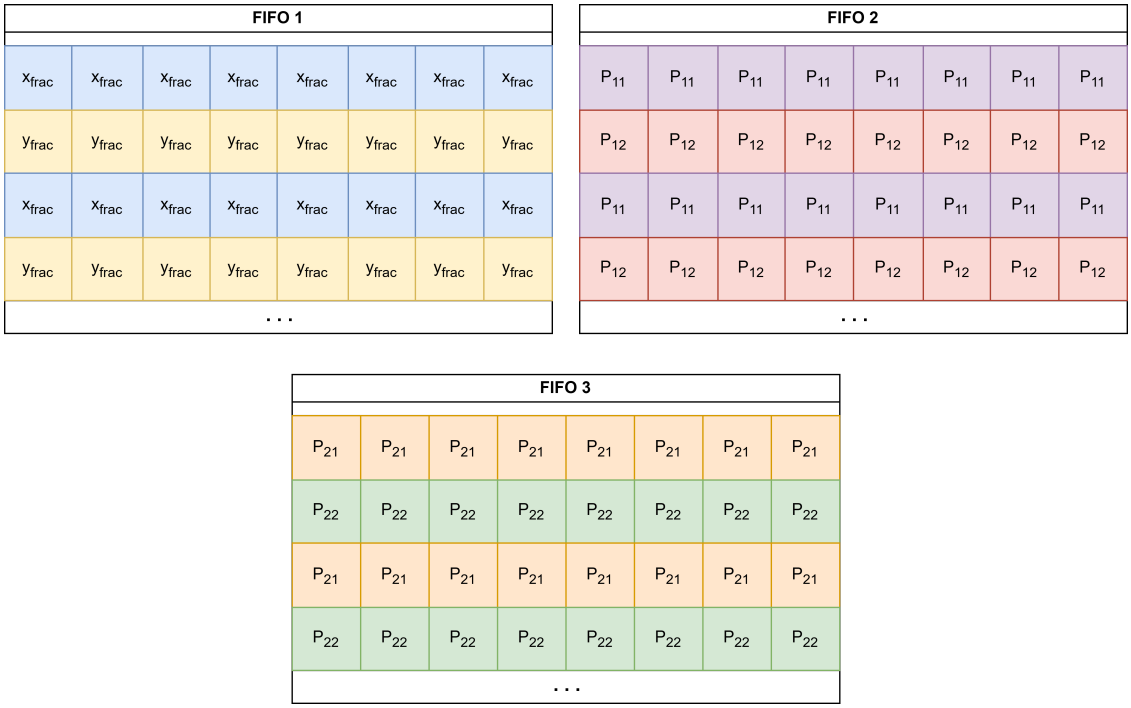

Figure 4.15. Representation of output data order in FIFOs

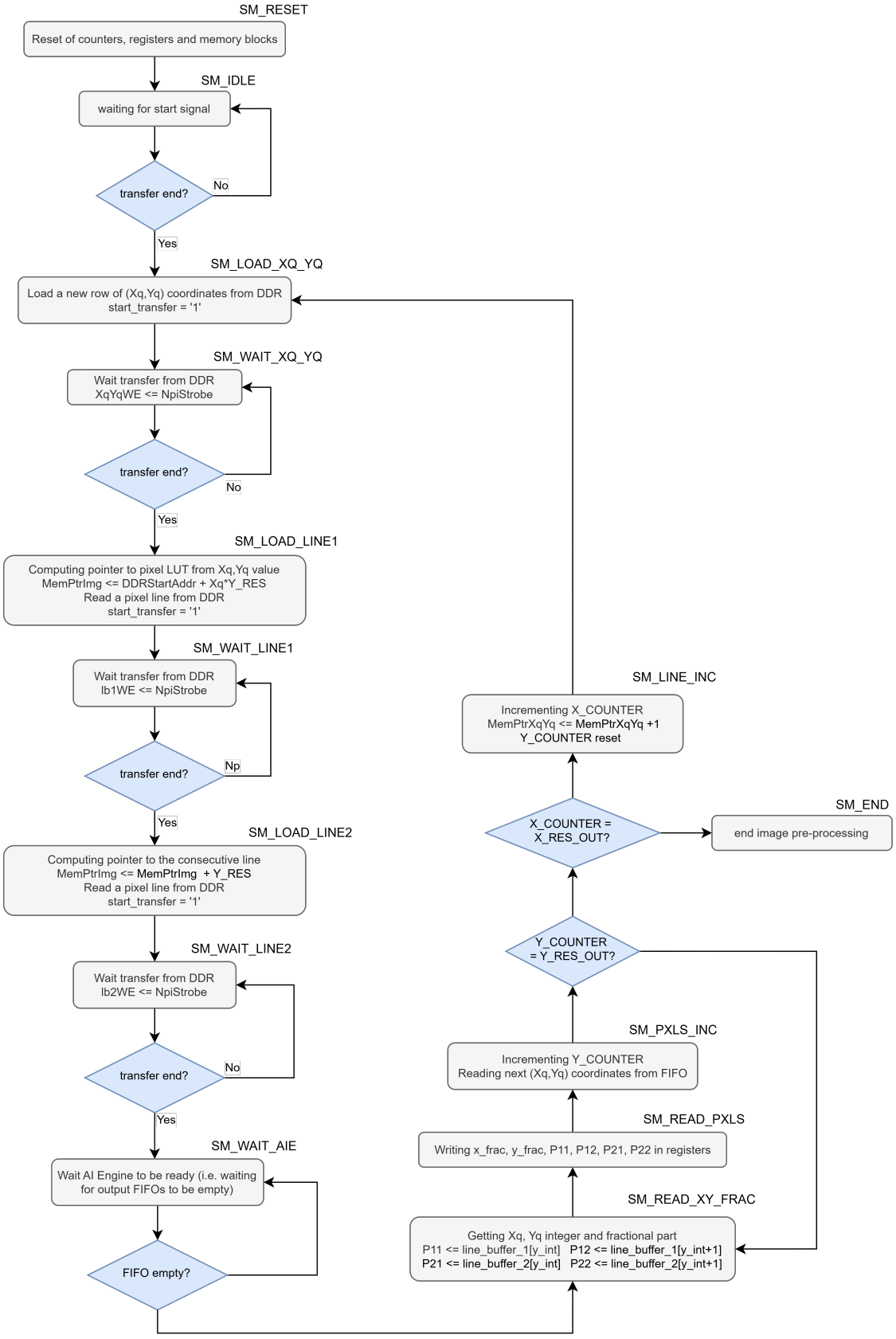

Figure 4.16. Algorithm finite state machine with AXI Stream interface

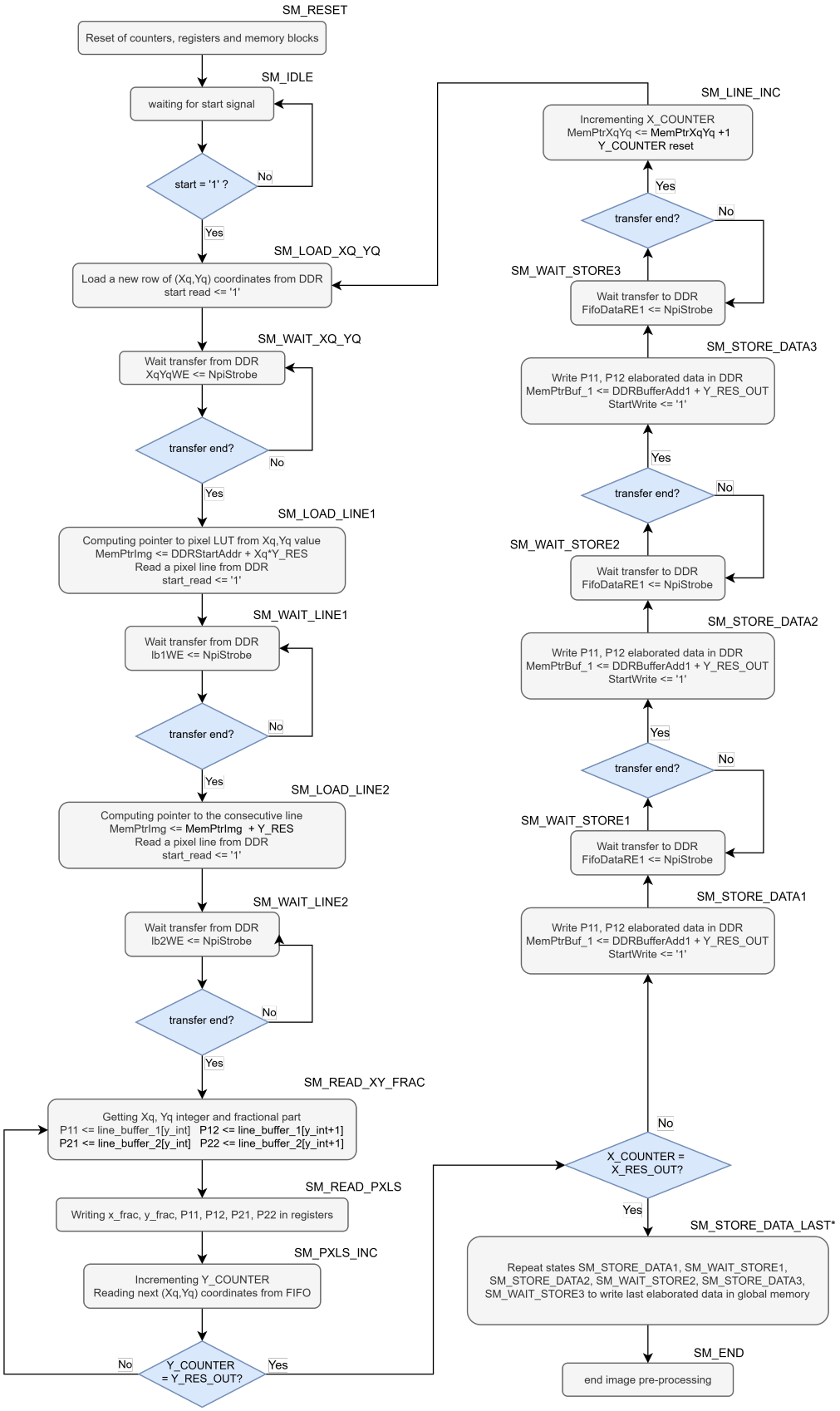

Figure 4.17. Algorithm finite state machine with output data written in DDR

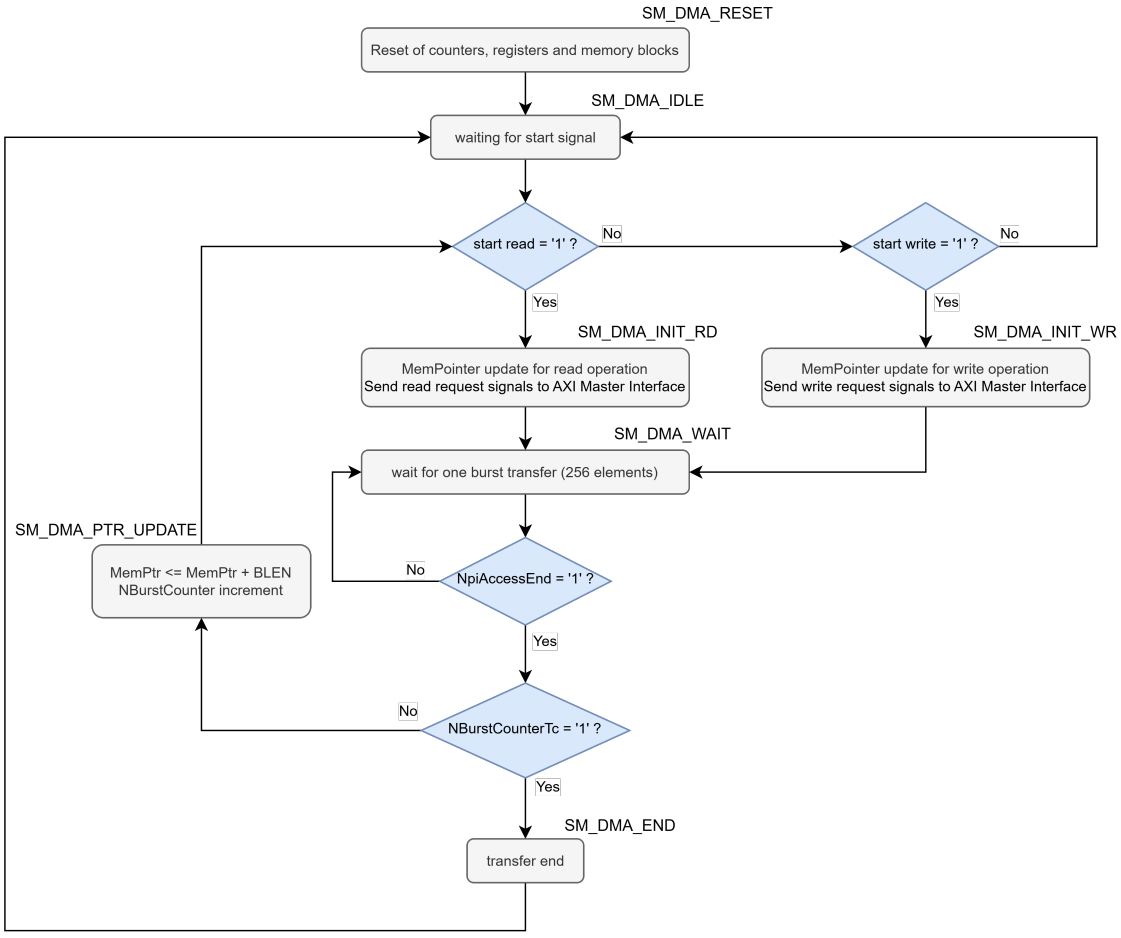

Figure 4.18. DMA management finite state machine

4.4 Generation of input and golden data

Input data to the system and golden reference data are generated using Matlab scripts. In particular:

- A first Matlab script extracts the pixels of the original image used for testing from a PNG file and generates the reference data for comparison;
- A second Matlab script generates the coordinate grid of the output image, i.e., the matrix of coordinates to interpolate, depending on the desired transformation (scaling, translation, rotation, crop, etc.). In this case, a scaling by half of the original image is implemented. The coordinates are generated in fixed point format and converted into floating point values inside the IP.
- A third Matlab script is also used to verify the correctness of the output data from the algorithm running on the board. This script is not provided since it is a simple for loop that compares golden data to the data obtained during hardware emulation and the algorithm's run.

Matlab scripts are provided in appendix 9.3.

All the blocks of the RTL IP are organized with parametric values, so the user can easily modify source codes to set different input and output image resolutions. Also, using a variable coordinate grid makes changing the type of image transformation simple: a different coordinate grid corresponds to a different transformation performed by the system exploiting the bilinear interpolation algorithm.

As mentioned earlier, in this reference design, the bilinear interpolation algorithm was used to **scale an input image**. The input resolution is 1024x1024 pixels, while the output resolution is 512x512. Therefore, a scaling factor 2 is applied to the x and y-directions.

Figure 4.19 shows one of the images used for the test.

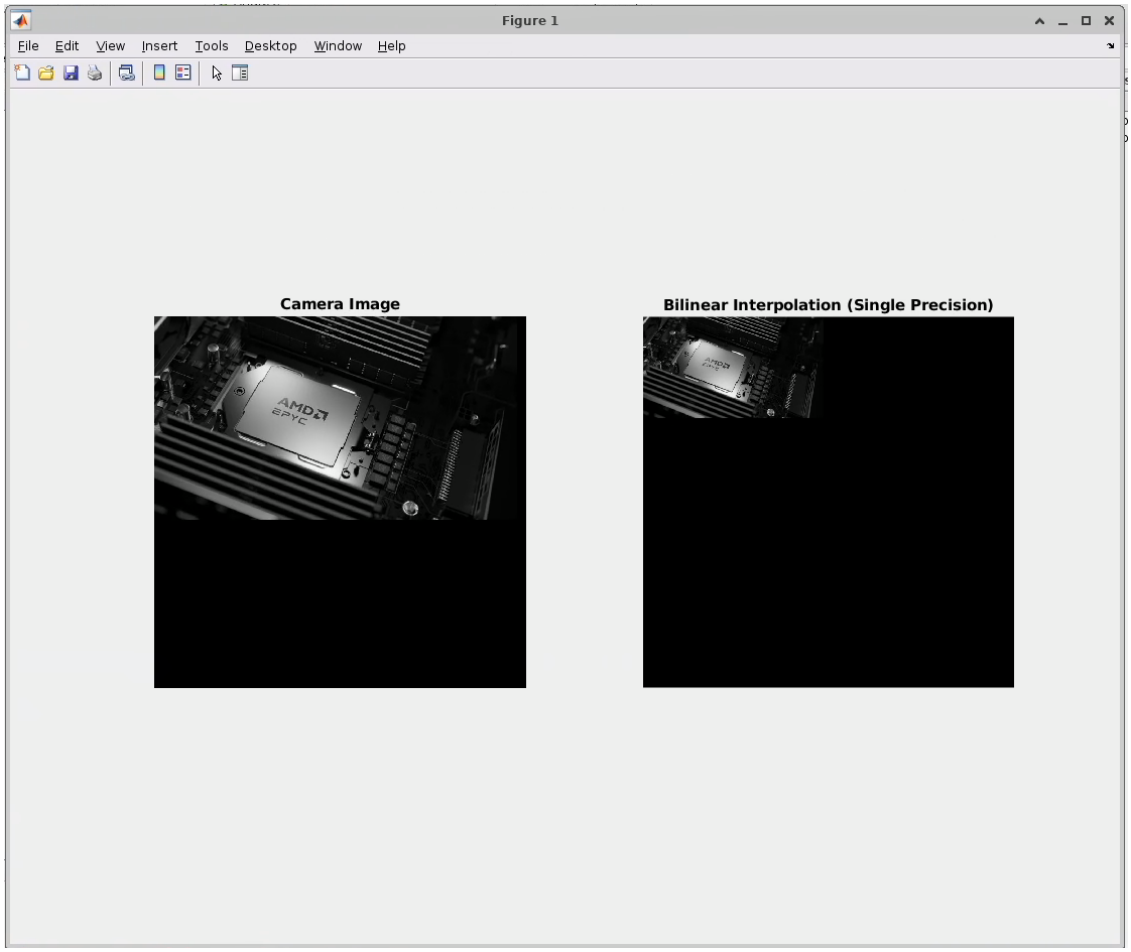

Figure 4.19. Scaling of an input image of resolution 1024x1024 by factor 2

Chapter 5

System Integration

5.1 Introduction to Versal Integration

The following chapter presents the integration of the components presented before, including AI Engines and PL kernels. Different solutions have been experimented with during the development of the VE2302 reference design. For this reason, this chapter presents three different designs:

- **Design 1**. The RTL IP and the AIE operate **sequentially** in the first system without direct interaction. The IP fully processes the input frame and writes the output data to DDR before the AIE execution begins. In this approach, the IP and RTL kernel exchange data using buffers in global memory. The functional block diagram of this design is provided in figure 5.1.
- **Design 2**. In the second design, the IP and AIE work **concurrently**. The IP provides data to the AIE in packets via an AXI4-Stream interface, and the AIE processes data between the reception of each packet. The functional block diagram of this design is provided in figure 5.2.
- **Design 3**. The third design uses a different approach based on AI Engines **without programmable logic kernels**. The image pre-processing part, carried out by the RTL IP in the two previous cases, is shifted to an AI Engine kernel. This case is presented in the next chapter.

This chapter mainly describes the first two designs, in which the integration of PL, PS, and AIE-ML components of the Versal SoC takes place. In both

cases, the AI Engine kernel's graph and code are the same. Instead, the RTL IP has been slightly modified to manage elaborated data with the different interfaces required by the two designs. Indeed, the two solutions require the implementation of different interfaces between RTL IP and AI Engine kernel:

- In the first case, the IP has been packaged with the addition of an AXI4 Memory-Mapped (AXI-MM) interface to read input pixels and query coordinates from the global memory and write elaborated data in three different buffers of the DDR. For this reason, three mm2s HLS interfaces provide input data to the AI Engine by reading them from DDR.
- In the second case, the IP has been packaged with an AXI4 Memory-Mapped interface to get input from DDR but also three AXI4 stream interfaces to send the elaborated data to the AI Engine PLIOs. In this case, mm2s interfaces are not required because the stream directly provides the data exchanges.

In both cases, the AI Engine kernel uses an s2mm interface to store interpolated pixels from the output buffer and the global memory.

5.2 Design linking

Vitis development flow for Versal devices allows the integration of PL and AI Engines kernel to create a complex design. There is the possibility to develop the single components with different tools provided by the Unified Vitis™ IDE and then link together heterogeneous parts. In figure 5.3, the integration process in this case is represented:

- The AIE compiler in Vitis compiles the ADF graph related to the bilinear interpolation kernel. This process generates the graph object and the library components necessary for execution on hardware;
- The Vitis HLS compiler compiles the HLS kernels; the synthesis result is a kernel object file.
- The RTL IP is designed and compiled in Vivado and synthesized as a Vitis kernel object.

The linker uses the output of each compilation stage to generate the hardware platform (an xsa file). An XSA (Xilinx Support Archive) file is a hardware design archive containing information about the created hardware platform, including:

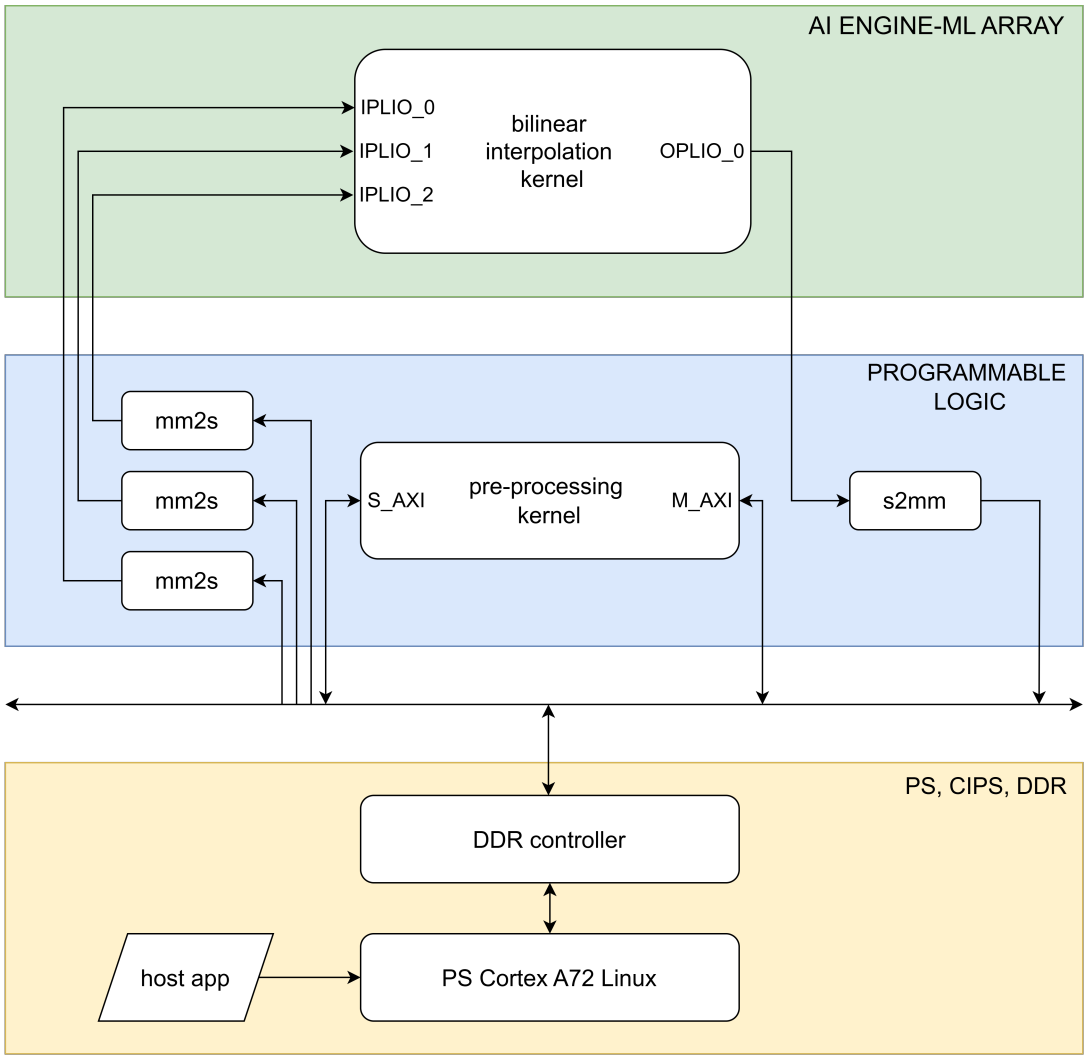

Figure 5.1. Functional block diagram of system 1

- FPGA bitstream;
- Block design (BD) information;
- AXI interconnect details;
- Peripheral configurations;
- Processor system setup.

As it is possible to see in figure 5.3, a configuration file is used during hardware linking. This system.cfg file provides an organized way of passing

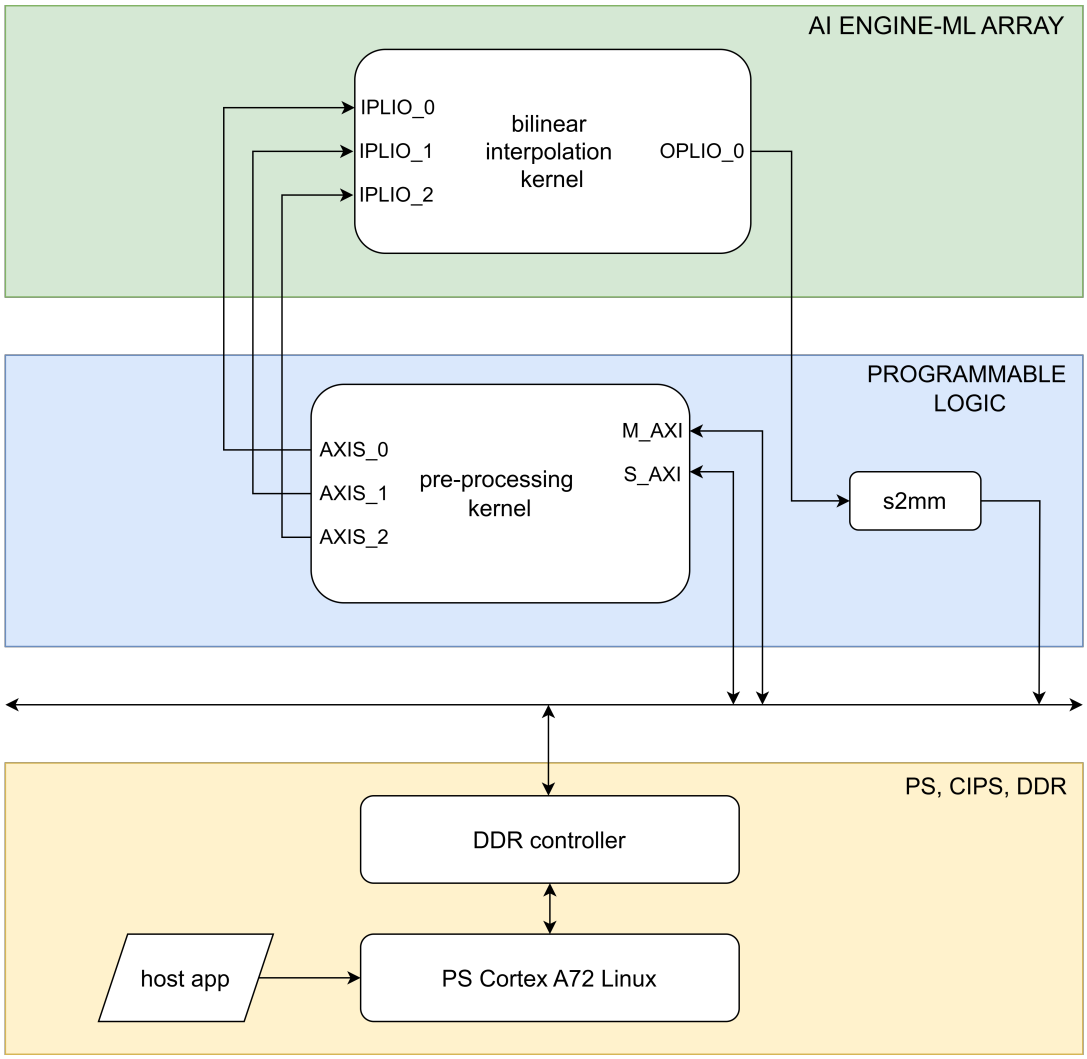

Figure 5.2. Functional block diagram of system 2

options to the tools creating reusable configuration files. Different commands are supported regarding information given to the v++ linker about the clock, package options, and debugging. The configuration files for the two designs are shown in listings 5.1 and 5.2, respectively.

```
1 debug=1
2 save-temps=1
3 temp_dir=binary_container_1
4 report_dir=binary_container_1/reports
5 log_dir=binary_container_1/logs
6
```

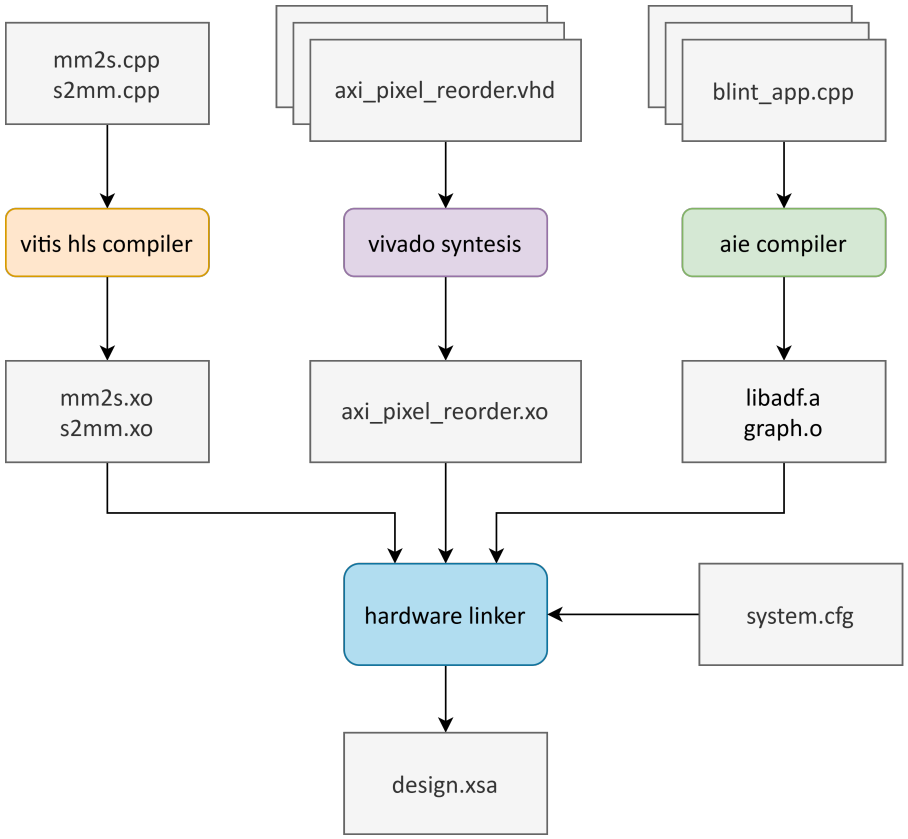

Figure 5.3. Integration of PL and AIE kernels

```
[profile]
data=all:all:all

[advanced]
misc=solution_name=binary_container_1
param=compiler.addOutputTypes=hw_export

[connectivity]
nk=s2mm:1:s2mm_1
nk=mm2s:3:mm2s_1.mm2s_2.mm2s_3
stream_connect=mm2s_1.s:ai_engine_0.DIN_0_A
stream_connect=mm2s_2.s:ai_engine_0.DIN_0_B
stream_connect=mm2s_3.s:ai_engine_0.DIN_0_C
stream_connect=ai_engine_0.DOUT_0:s2mm_1.s
nk=axi_pixel_reorder:1:axi_pixel_reorder_1
```

Listing 5.1. system.cfg file for design 1

```
debug=true
```

```
save-temps=1
temp_dir=binary_container
report_dir=binary_container/reports
log_dir=binary_container/logs

[profile]
data=all:all:all

[advanced]
misc=solution_name=binary_container
param=compiler.addOutputTypes=hw_export

[connectivity]
nk=s2mm:1:s2mm_1
stream_connect=axi_pixel_reorder_1.out0:ai_engine_0.DIN_0_A
stream_connect=axi_pixel_reorder_1.out1:ai_engine_0.DIN_0_B
stream_connect=axi_pixel_reorder_1.out2:ai_engine_0.DIN_0_C
stream_connect=ai_engine_0.DOUT_0:s2mm_1.s
nk=axi_pixel_reorder:1:axi_pixel_reorder_1
```

Listing 5.2. system.cfg file for design 2

In particular, the option `param=compiler.addOutputTypes=hw_export` tells the compiler to generate the XSA file during the linking stage for hardware export. A relevant part concerns connectivity, providing the linker with information about the number of instances of the same core and their connection.

The linking command is the following:

```
v++ -l -t hw s2mm.xo mm2s.xo libadf.a -config system.cfg -o
                        design.xsa
```

One of the linking process's output files is the hardware platform's Vivado project, where the result of this step can be seen. Figure 5.4 shows the block design of the linked platform. In particular, it is possible to notice the presence of the AI Engine block, connected to the Control Interface and Processing System IP (CIPS) and the **Vitis Region**.

Expanding the Vitis region in the Vivado project of the first system (figure 5.5) makes it possible to see the PL kernels included in the design. As expected, the s2mm/mm2s interfaces are connected to the pre-processing RTL IP. It is possible to notice that all these blocks, and also the inputs of the AI Engine, are connected to the NoC, which provides a bridge to the memory.

In the Vitis region for the second design (figure 5.6), it is possible to see the presence of the s2mm interface, which provides communication between the

AIE output and the memory. As expected, the stream connection between the AIE inputs and the reorder kernel also directly provides data to the interpolation kernel.

Inspecting the Vitis project is also useful to check the correctness of the linked design before proceeding with the packaging.

5.3 Software host application

A **software application** running on the PS is required to control the algorithm's execution on hardware. For this aim, Vitis development flow provides the **XRT API**, allowing programmers to develop their own applications in C, C++, and Python. The **Xilinx Runtime library** (XRT) is an open-source software stack that facilitates the management and usage of Versal and other AMD devices. Developers can write software using familiar programming languages like C/C++ to write host code, and the XRT API provides an easy way of interacting with FPGA/ACAP devices. To use the native XRT APIs, the compiler must link the `xrt_coreutil` library to the host application.

This library provides different class objects to support all the elements of the heterogeneous environment presented, summarized in table 5.1.

Concept	**XRT Class**	**Header File**
Device	`xrt::device`	`#include <xrt/xrt_device.h>`
XCLBIN	`xrt::xclbin`	`#include <experimental/xrt_xclbin.h>`
Buffer	`xrt::bo`	`#include <xrt/xrt_bo.h>`
Kernel	`xrt::kernel`	`#include <xrt/xrt_kernel.h>`
Run	`xrt::run`	`#include <xrt/xrt_kernel.h>`
User-managed Kernel	`xrt::ip`	`#include <experimental/xrt_ip.h>`
Graph	`xrt::graph`	`#include <experimental/aie.h>` `#include <experimental/graph.h>`

Table 5.1. Software controllable kernels

These common steps characterize the typical host code flow:

- Open Xilinx Device and load the XCLBIN file;
- Create Buffer objects to transfer data to kernel inputs and outputs;
- Use the Buffer class member functions for the data transfer between host and device (before and after the kernel execution);
- Use Kernel and Run objects to offload and manage the compute-intensive tasks on FPGA.

An **XCLBIN file** is a Xilinx Binary Container that packages the compiled hardware design for execution on AMD/Xilinx platforms. The key components of an xclbin file are:

- **Bitstream**, which is the actual configuration data for the programmable logic;
- **Kernel metadata**, i.e., information about kernel arguments, memory mappings, and compute units;
- **ADF graph metadata**, if AI Engine kernels are included;
- **Connectivity information**, describing how the hardware kernels are connected to global memory and other system components;
- **Clocking and resource configurations**, defining the frequency settings and the resource usage.

5.3.1 Host application code

The host application code of the second system is described as an example to show the typical structure and use of XRT API. In listing 5.3, the first part of the application code is presented. It is possible to notice the use of class objects `xrt::xclbin` and `xrt::device`, performing device opening and xclbin loading.

```
1 int main(int argc, char* argv[]) {
2
3 // Open device and load xclbin
4
5 char* xclbinFilename = argv[1];
6 unsigned dev_index = 0;
```

```
auto my_device = xrt::device(dev_index);

// ...
```

Listing 5.3. host code: loading xclbin and device opening (system 2)

The next step is buffer allocation in global memory. Buffers are mainly used to transfer the data between the host and the device, but they can also be used to transfer data between RTL IP and AI engines, as in the first system. The class constructor `xrt::bo` is used primarily to allocate a buffer object 4K aligned.

```
// allocating input and output memory

//input buffer for RTL IP
auto in_bo1 = xrt::bo(my_device, DDR1_BUFFSIZE_I_BYTES,
                  xrt::bo::flags::host_only, 0);
auto in_bo1_mapped = in_bo1.map<uint64_t*>();
memcpy(in_bo1_mapped, img_pxl, DDR1_BUFFSIZE_I_BYTES);
in_bo1.sync(XCL_BO_SYNC_BO_TO_DEVICE);

auto in_bo2 = xrt::bo(my_device, DDR2_BUFFSIZE_I_BYTES,
                  xrt::bo::flags::host_only, 0);
auto in_bo2_mapped = in_bo2.map<uint64_t*>();
memcpy(in_bo2_mapped, coords, DDR2_BUFFSIZE_I_BYTES);
in_bo2.sync(XCL_BO_SYNC_BO_TO_DEVICE);

//ouput buffer for AIE results
auto out_bo = xrt::bo(my_device, DDR_BUFFSIZE_O_BYTES, 0);
auto out_bo_mapped = out_bo.map<uint32_t*>();
memset(out_bo_mapped, 0x0000, DDR_BUFFSIZE_O_BYTES);
```

Listing 5.4. host code: buffers allocation (system 2)

In design 1, three additional buffer objects are allocated to store intermediate data generated by the RTL IP and used as inputs for the AI engine. The next step consists of the allocation of the RTL IP kernel. The class object `xrt::ip` is used since it is a user-managed IP (listing 5.5). In addition, the addresses of the input buffers are passed to the IP using the provided register interface.

```
// allocating RTL IP kerneL
auto ip1 = xrt::ip(my_device, xclbin_uuid,
               "axi_pixel_reorder:{axi_pixel_reorder_1}");

// setting up RTL IP registers
uint64_t addr1;
uint64_t addr2;
```

```
addr1 = in_bo1.address();
addr2 = in_bo2.address();

ip1.write_register(BASE_ADDR1, addr1 >> 32);
ip1.write_register(BASE_ADDR1 + 4, addr1);
ip1.write_register(BASE_ADDR2, addr2 >> 32);
ip1.write_register(BASE_ADDR2 + 4, addr2 );

uint32_t reg1 = ip1.read_register(BASE_ADDR1);
uint32_t reg2 = ip1.read_register(BASE_ADDR1+4);
uint32_t reg3 = ip1.read_register(BASE_ADDR2);
uint32_t reg4 = ip1.read_register(BASE_ADDR2+4);

ip1.write_register(PRESCALER_REG, PRESCALER_VALUE);
```

Listing 5.5. RTL IP allocation and register setup (design 2)

The allocation of s2mm and mm2s interfaces instead is provided by the class object `xrt::kernel`. These HLS kernels are configured as xrt-managed kernels; for this reason, high-level calls as `xrt::run()` and `xrt::wait()` are used, as shown in the code 5.6.

```
// start output kernel
auto s2mm_k = xrt::kernel(my_device, xclbin_uuid, "s2mm:{s2mm_1}");
auto s2mm_r = s2mm_k(out_bo, nullptr, DDR_BUFFSIZE_O_BYTES/sizeof(int64));
std::cout << "s2mm run started" << std::endl;
```

Listing 5.6. RTL IP allocation and register setup (design 2)

Instead, the user must manage the RTL kernel run through atomic register read/write operations from the host application. The RTL IP has been designed to start when bit 0 of register 0x10 is set to 1. In the case of design 1, the IP processes the whole image, and the AIE graph executes only after the pre-processing ends. For this reason, a while loop implements polling on bit 2 of register 0x10, corresponding to a "done" signal (listing 5.7). Instead, in design 2, the elaboration of AIE and RTL IP happens concurrently, and no polling is required.

```
// start RTL kernel
uint32_t axi_ctrl = IP_START;
ip1.write_register(CTRL_ADDR_REG, axi_ctrl);

while( axi_ctrl != IP_END ){

    axi_ctrl = ip1.read_register(CTRL_ADDR_REG);
    if(axi_ctrl == IP_START){
        ip1.write_register(CTRL_ADDR_REG, IP_NONE);
    }
```

```
}
```

Listing 5.7. RTL IP allocation and register setup (design 1)

The last step is the allocation of the AI Engine graph, using class object `xrt::graph`, as shown in code 5.8. The API call `my_graph.run(NRUN)` is used to execute the graph for `NRUN` times, and the function `my_graph.end()` waits until the graph execution is completed.

```
// Load AIE graph
auto my_graph = xrt::graph(my_device, xclbin_uuid, "blint");

// Run AIE graph
my_graph.reset();
my_graph.run(NRUN);
my_graph.end();
```

Listing 5.8. RTL IP allocation and register setup (design 2)

The complete codes of the host application for both design 1 and design 2 is provided in appendix 9.4. In the complete code, it can be seen that, at the end of the processing, the host app also writes the interpolated pixels to a text file ("output.txt") for verification with Matlab and performs a runtime check with golden data to detect errors. In fact, during the application execution, the console output will report any mismatch with the reference data and the maximum absolute error.

5.4 Design packaging

The last step in getting a design targeting the board is platform creation with **packaging**. Indeed, the platform is a package that contains:

- the **HPFM** (Hardware platform) i.e., the XSA file describing the hardware components;
- the **SPFM** (Software components), including the common image, Linux kernel, rootfs, device tree, and boot components.

These two components are the input of the AMD Vitis™ IDE tool that will generate the platform. The whole process is illustrated in figure 5.7. The Vitis command:

```
v++ -package -config package.cfg
```

generates SD card and other Flash images required for booting the system, in addition to the .xclbin device binary from the .xsa generated for Versal devices. In the configuration file `package.cfg`, it is possible to specify packaging flags, input files, and different options, which are summarized in table 5.2.

Option	Description
`rootfs`	Root filesystem, contains the Linux operating system and essential user-space utilities for the target platform.
`image_format`	Format of the Image
`boot_mode`	How the design is going to be run (e.g. QSPI, SD card).
`kernel_image`	Image file created by PetaLinux.
`defer_aie_run`	The AI Engine will not automatically start when the system is booted, the host application control it.
`sd_file`	Tells the packager what file is to be packaged in the `sd_card` directory. This must be specified multiple times for all the files to be packaged.

Table 5.2. Packaging options

After the creation of the platform (including hardware and software), the user can generate the **Programmable Device Image** (PDI) and a package for SD card boot. The PDI contains executables, bitstreams, and configurations of every device element, and it is used during the boot process for configuration. The packaged SD card directory contains everything to boot Linux and runs the generated host application on the created Vitis platform.

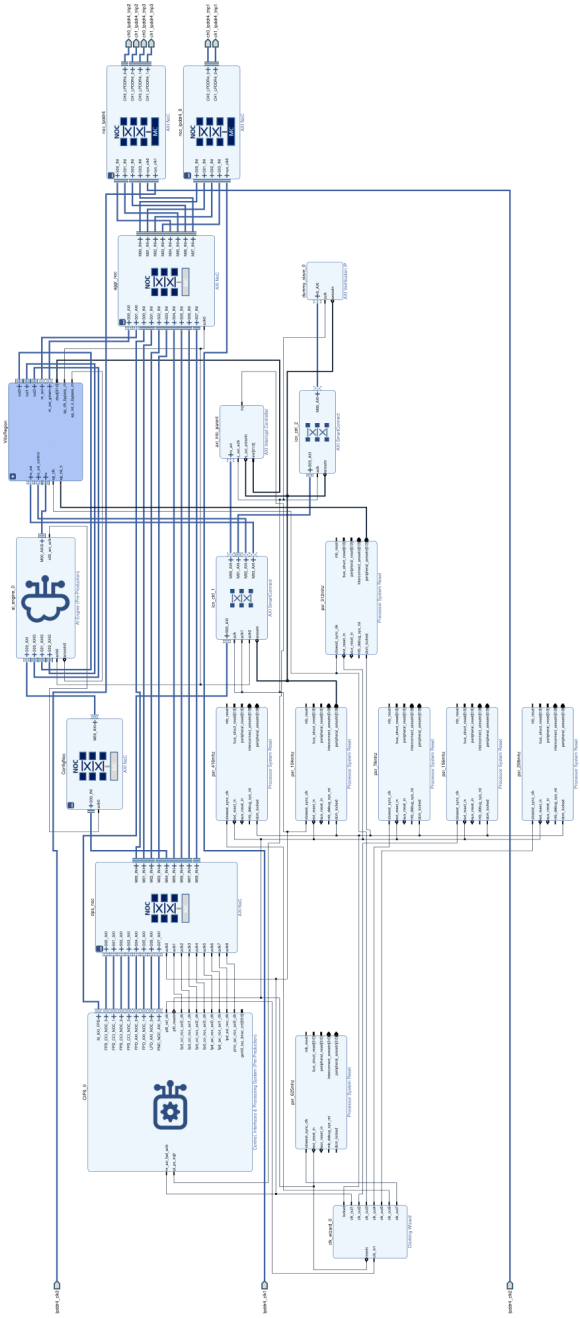

Figure 5.4. Platform block design

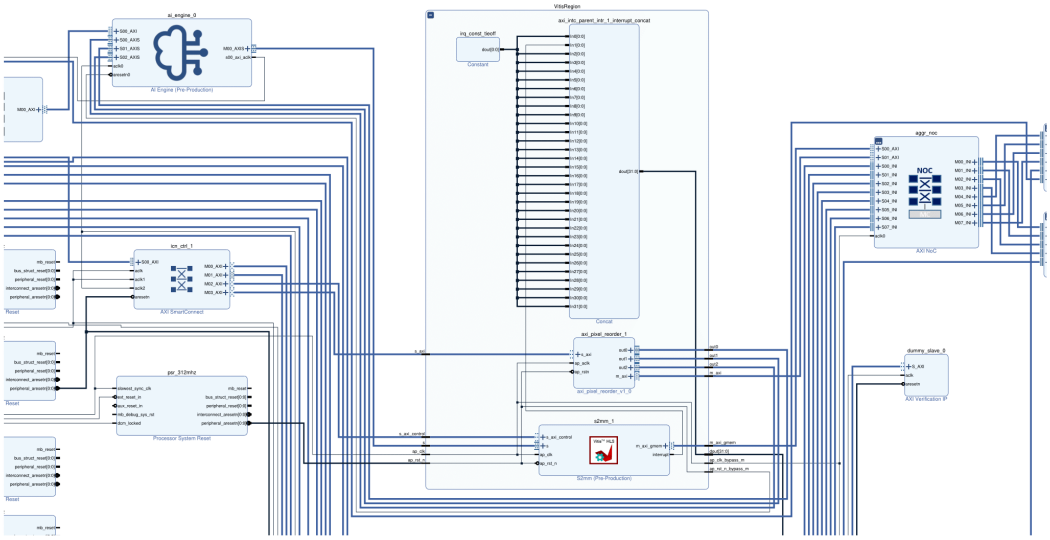

Figure 5.5. Platform block design - Vitis region of system 1

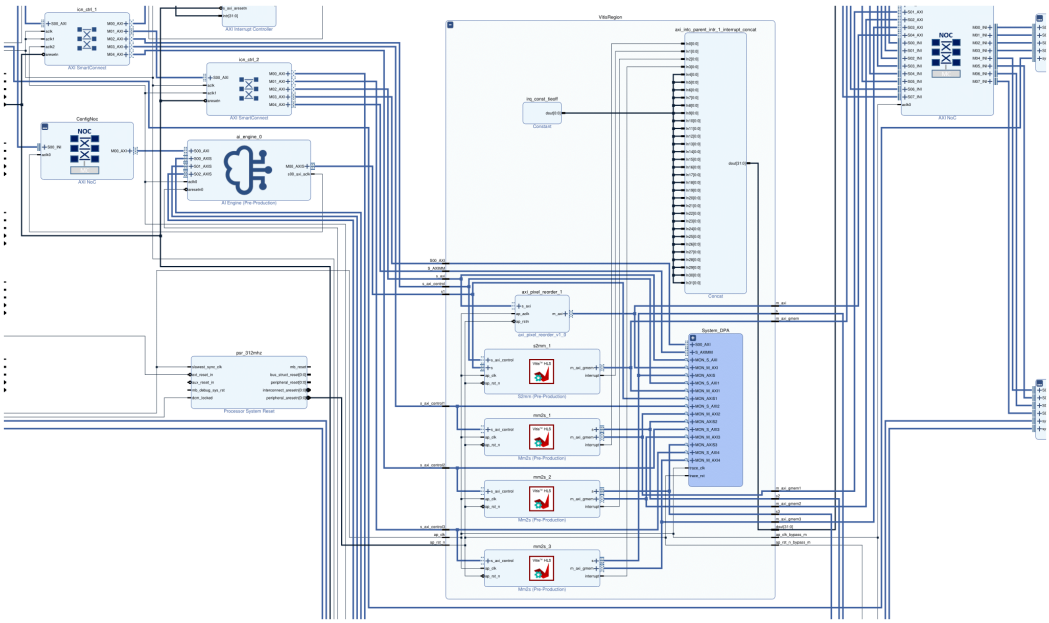

Figure 5.6. Platform block design - Vitis region of system 2

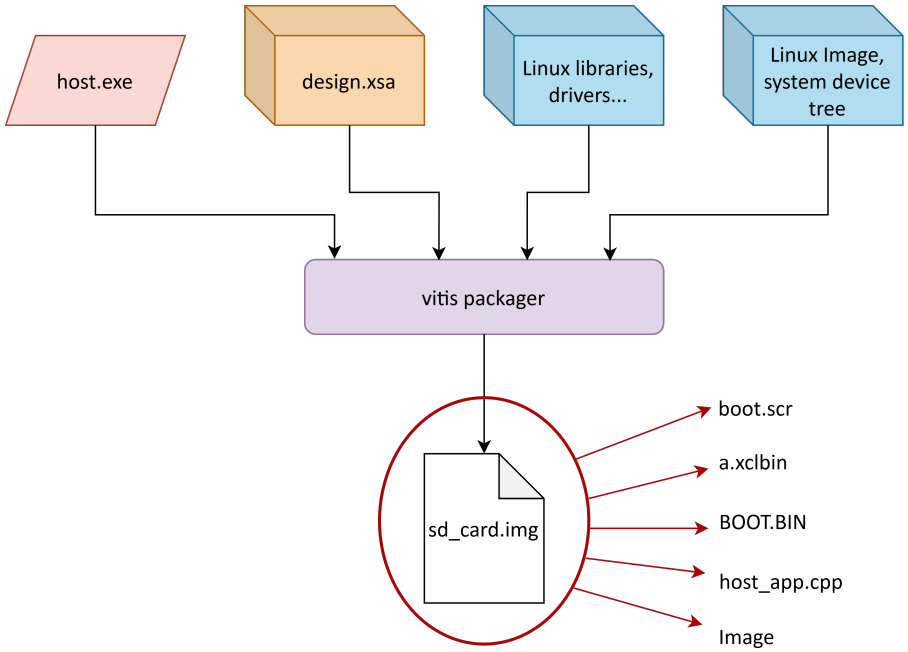

Figure 5.7. Packaging the design

Chapter 6

Bilinear Interpolation with AI Engine based approach

The last part of this project involves developing the bilinear interpolation algorithm using a complete **AI engine-based approach**. The idea is to evaluate the system's performance when the programmable logic is not involved, and the pre-processing part is moved to the AI Engine graph. This part aims to assess the advantages and disadvantages of a more software-oriented approach versus one that includes a deeper knowledge of hardware.

Figure 6.1 illustrates the graph in this specific approach from a functional point of view. In this case, the AI Engine graph consists in two different kernels:

- Bilinear interpolation kernel, which remains unchanged from what was described above;
- Pre-processing kernel, which implements the pixel pre-processing part that was previously the responsibility of the RTL IP in programmable logic.

In this case, **GMIO ports** have been used to transfer data from global memory to the AI Engine array. Three GMIO inputs provide the coordinates of the interpolation point (x_q, y_q) and the pixels of the original image. An output GMIO transfers data from the AI engine array to the global memory.

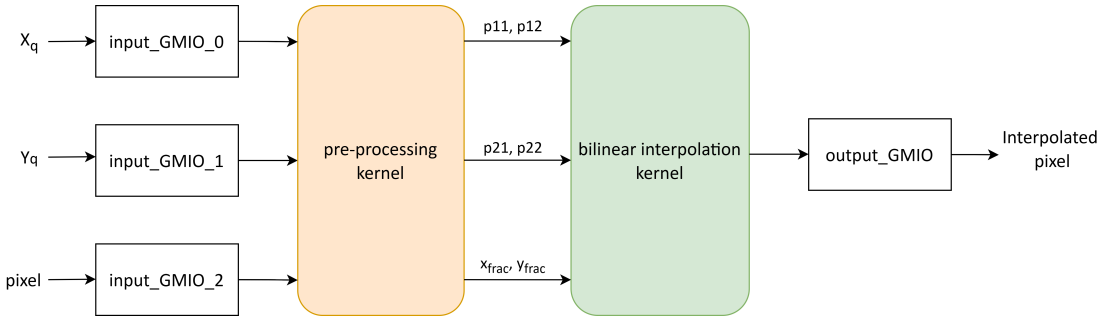

Figure 6.1. Bilinear kernel graph with pre-processing: functional representation

PLIO ports are not required since there isn't interaction with programmable logic.

6.1 AI Engine array programming

6.1.1 Bilinear kernel graph code

The code of the bilinear kernel graph is shown in listing 6.1. As can be seen from the code, four GMIO ports were created using the class object provided by the adaptive dataflow library. Each port attribute is characterized by the logical name, the size of the burst transfer (256 in this case), and the bandwidth, set to 1000 MB/s. The next part of the code shows the presence of a second kernel, which is used to substitute the RTL IP kernel. Then, as in the previous case, the graph code describes the connection between ports and kernels and provides information such as the runtime ratio and the source code for each kernel.

```
1  #pragma once
2  #include <adf.h>
3  #include "bilinear_kernel.h"
4  #include "reorder_kernel.h"
5  #include "buffers.h"
6
7  using namespace adf;
8
9  class bilinear_graph : public adf::graph {
10 private:
11
12     kernel bli_krnl;
13     kernel reorder_krnl;
```

```
14
15 public:
16
17     adf::input_gmio gmioIn1;
18     adf::input_gmio gmioIn2;
19     adf::input_gmio gmioIn3;
20     adf::output_gmio gmioOut;
21
22 bilinear_graph()
23 {
24
25     gmioOut = adf::output_gmio::create("gmioOut",256,1000);
26     gmioIn1 = adf::input_gmio::create("gmioIn1",256,1000);
27     gmioIn2 = adf::input_gmio::create("gmioIn2",256,1000);
28     gmioIn3 = adf::input_gmio::create("gmioIn3",256,1000);
29
30     reorder_krnl = kernel::create_object<reorder_kernel>();
31     bli_krnl = kernel::create_object<bilinear_kernel>();
32
33     connect(gmioIn1.out[0], reorder_krnl.in[0]);
34     connect(gmioIn2.out[0], reorder_krnl.in[1]);
35     connect(gmioIn3.out[0], reorder_krnl.in[2]);
36
37     connect(reorder_krnl.out[0], bli_krnl.in[0]);
38     connect(reorder_krnl.out[1], bli_krnl.in[1]);
39     connect(reorder_krnl.out[2], bli_krnl.in[2]);
40
41     connect(bli_krnl.out[0], gmioOut.in[0]);
42
43     source(bli_krnl) = "src/bilinear_kernel.cpp";
44     source(reorder_krnl) = "src/reorder_kernel.cpp";
45
46     runtime<ratio>(bli_krnl) = 0.9;
47     runtime<ratio>(reorder_krnl) = 0.9;
48
49     }
50 };
```

Listing 6.1. Bilinear kernel graph code

6.1.2 Reorder kernel code

The bilinear kernel code remains the same as previously presented for the other designs. Instead, this section provides a description of the code used for pre-processing. The first part of the code is shown in listing 6.2.

```
#include <adf.h>
#include "buffers.h"
#include "reorder_kernel.h"
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include "aie_api/vector.hpp"

void reorder_kernel::reorder(input_buffer<int32 , adf::extents<BUFF_DIM_1>>
                          & __restrict in1,
                          input_buffer<int32, adf::extents<BUFF_DIM_2>>
                          & __restrict in2,
                          input_buffer<int32, adf::extents<BUFF_DIM_2>>
                          & __restrict in3,
                          output_buffer<int32, adf::extents<BUFF_DIM_INT>>
                          & __restrict out1,
                          output_buffer<int32, adf::extents<BUFF_DIM_INT>>
                          & __restrict out2,
                          output_buffer<int32, adf::extents<BUFF_DIM_INT>>
                          & __restrict out3)
{
    // iterators to access input and output buffer
    auto inPtrP=aie::begin_random_circular(in1); //random access for pixel lut
    auto inPtrXq=aie::begin(in2);
    auto inPtrYq=aie::begin(in3);
    auto outPtr1=aie::begin_vector<8>(out1);
    auto outPtr2=aie::begin_vector<8>(out2);
    auto outPtr3=aie::begin_vector<8>(out3);

    // declaration of test vector
    aie::vector<float, 8> xfrac_v = aie::zeros<float,8>();
    aie::vector<float, 8> yfrac_v = aie::zeros<float,8>();
    aie::vector<float, 8> p11_v = aie::zeros<float,8>();
    aie::vector<float, 8> p12_v = aie::zeros<float,8>();
    aie::vector<float, 8> p21_v = aie::zeros<float,8>();
    aie::vector<float, 8> p22_v = aie::zeros<float,8>();
```

Listing 6.2. Reorder kernel code - first part

Also in this case, the kernel uses buffered I/O for input and output, as shown by the presence of `input_buffer()` and `output_buffer()` in the function declaration. At the beginning of the kernel code, the iterators that are used to access the buffers are created. A thing to notice is that a circular random iterator is instantiated to access the pixel buffer. This iterator will not give ordered access to the buffer, as it needs to select the four neighboring pixels based on the value of the coordinate to be interpolated.

```
    int samp_cnt = 0;
```

```
2
3   /*iterate over query coordinates buffer*/
4       for (unsigned i = 0; i < YRES_OUT; i++)
5       chess_prepare_for_pipelining
6       chess_loop_count(YRES_OUT)
7   {
8       //get query and get integer part
9       auto xq = (*inPtrXq++);
10      auto yq = (*inPtrYq++);
11
12      float xq_fp = *reinterpret_cast<float*>(&xq);
13      float yq_fp = *reinterpret_cast<float*>(&yq);
14
15      auto xint = std::floor(xq_fp);
16      auto yint = std::floor(yq_fp);
17
18      auto xfrac = xq_fp - xint;
19      auto yfrac = yq_fp - yint;
20
21      //get index of p11, p12, p21, p22 in input buffer
22      uint32_t p11_idx = yint; //xint*YRES+yint;
23      uint32_t p12_idx = yint+1; //xint*YRES+yint+1;
24      uint32_t p21_idx = YRES+yint; //(xint+1)*YRES+yint;
25      uint32_t p22_idx = YRES+yint+1; //(xint+1)*YRES+yint+1;
26
27      inPtrP += p11_idx;
28      auto p11 = *inPtrP;
29      float p11_fp = *reinterpret_cast<float*>(&p11);
30
31      inPtrP += 1; //p12 index is p11_index + 1
32      auto p12 = *inPtrP;
33      float p12_fp = *reinterpret_cast<float*>(&p12);
34      inPtrP=aie::begin_random_circular(in1);
35
36      inPtrP += p21_idx;
37      auto p21= *inPtrP;
38      float p21_fp = *reinterpret_cast<float*>(&p21);
39
40      inPtrP += 1;
41      auto p22= *inPtrP;
42      float p22_fp = *reinterpret_cast<float*>(&p22);
43      inPtrP=aie::begin_random_circular(in1);
44
45      //insert element in 8-vector
46      xfrac_v.set(xfrac, samp_cnt);
47      yfrac_v.set(yfrac, samp_cnt);
```

```
48          p11_v.set(p11_fp, samp_cnt);
49          p12_v.set(p12_fp, samp_cnt);
50          p21_v.set(p21_fp, samp_cnt);
51          p22_v.set(p22_fp, samp_cnt);
52
53          if(samp_cnt == 7){
54
55              *outPtr1++ = aie::vector_cast<int32>(xfrac_v);
56              *outPtr1++ = aie::vector_cast<int32>(yfrac_v);
57
58              *outPtr2++ = aie::vector_cast<int32>(p11_v);
59              *outPtr2++ = aie::vector_cast<int32>(p12_v);
60
61              *outPtr3++ = aie::vector_cast<int32>(p21_v);
62              *outPtr3++ = aie::vector_cast<int32>(p22_v);
63
64              samp_cnt = 0;
65          }
66          else{
67              samp_cnt++;
68          }
69      }
70  }
```

Listing 6.3. Reorder kernel code - second part

The second part of the code (listing 6.3) is a for loop producing the set of data (`p_11, p_12, p_21, p_22, x_frac, y_frac`) required for the interpolation of each pixel. For each coordinate (x_q, y_q) in the buffer, the fractional and integer part is obtained, and the integer part is used to find the position of each required pixel inside the pixel buffer. The selected pixels are grouped into vectors of 8 elements (to allow vector calculation in the next kernel), and the process continues for all output image coordinates. Interestingly, the programmer does not have to worry about managing data transfer from global memory to the AI Engine array. Once the burst size and bandwidth are set, the GMIO port handles the data traffic depending on the available space in the AI Engines' local buffers.

The complete kernel code is provided in the appendix 9.1, listing **??**.

6.1.3 Bilinear graph compilation

The adaptive data flow graph obtained in this case after the aie-compilation is shown in figure 6.2.

The allocation of the graph in the AI Engine array is shown in the array

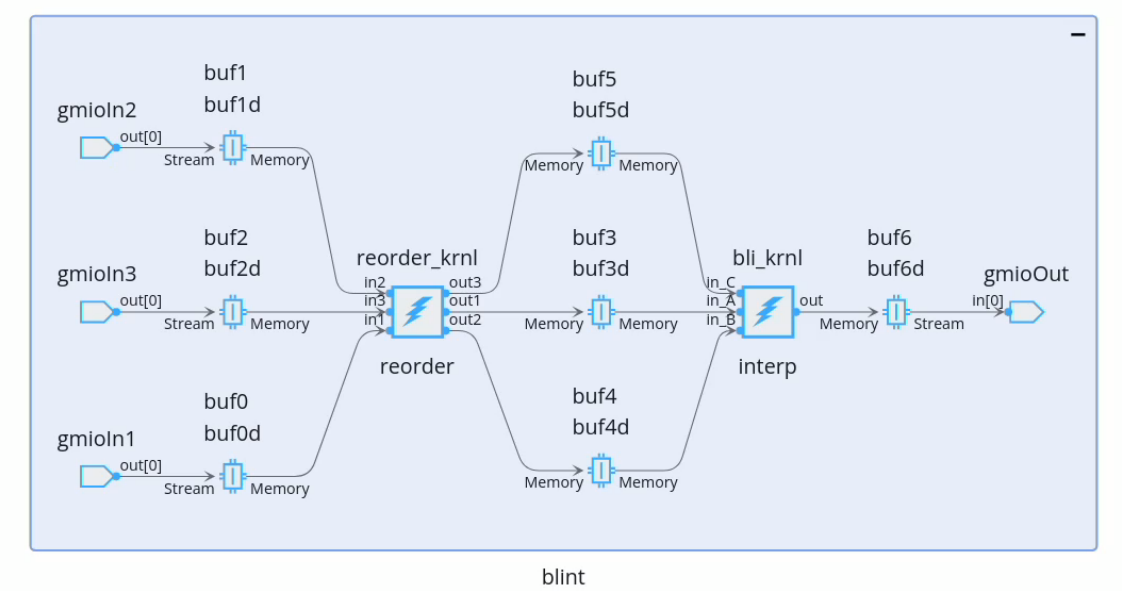

Figure 6.2. Bilinear kernel graph after compilation

view, figure 6.3 and 6.4.

6.2 Software host application

In this case, the presence of a host application is also necessary to manage the allocation of the AI Engine graph and the buffer in global memory and the system's evolution. The first part of the host application code is shown in listing 6.4. The operations performed in this phase are the same as those introduced previously, i.e., the loading of the xclbin file, the allocation and initialization of buffers in memory, and the allocation of the graph.

```
1  //
2  // include XRT libraries
3  // ...
4
5  #define NRUN 512
6  #define SCALE_FACTOR 2
7  #define YRES 1024
8
9  using namespace adf;
10 using namespace std;
11
```

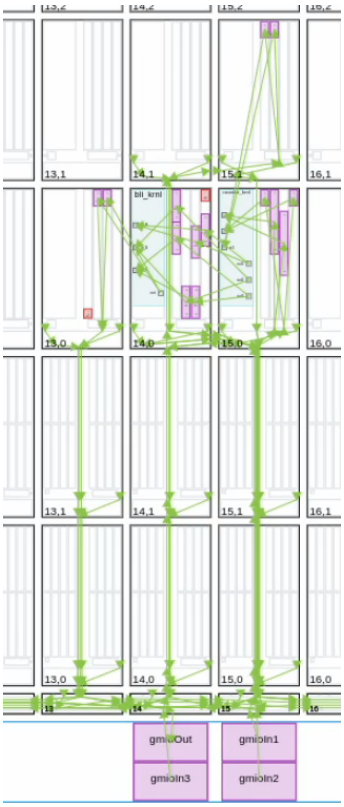

Figure 6.3. Bilinear kernel graph after compilation - array view

```
12 int main(int argc, char* argv[]) {
13
14     if (argc != 2) {
15         std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
16         return 1;
17     }
18
19     // Open device and load xclbin
20     char* xclbinFilename = argv[1];
21     unsigned dev_index = 0;
22     auto my_device = xrt::device(dev_index);
23     if(!my_device){
24         std::cout << "Device open error!" << std::endl;
25     }else{
26         std::cout << "Device open OK!" << std::endl;
27     }
28     auto xclbin_uuid = my_device.load_xclbin(xclbinFilename);
29
30     // allocating input and output memory
```

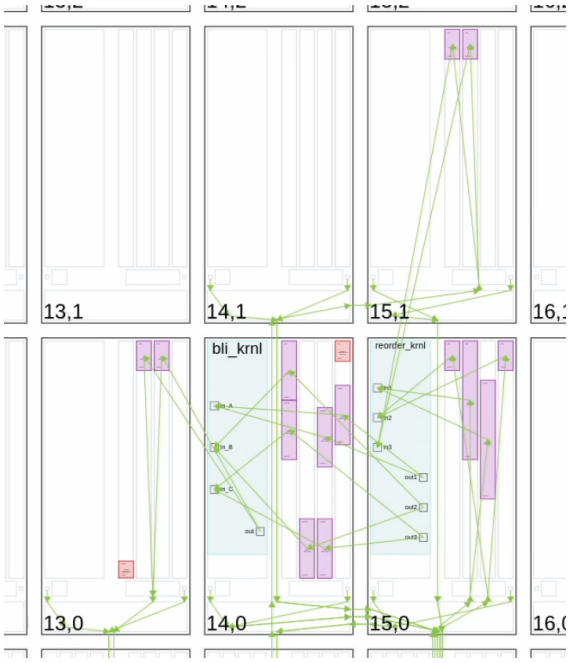

Figure 6.4. Bilinear kernel graph after compilation - zoom

```
31
32    auto din_xq_buffer = xrt::bo (my_device, BUFF_SIZE_XY,
33    xrt::bo::flags::normal, 0);
34    uint32_t* dinXqArray= din_xq_buffer.map<uint32_t*>();
35    memcpy(dinXqArray, xq_array, BUFF_SIZE_XY);
36    std::cout << "INFO: Xq array allocated. "<< std::endl;
37
38    // ...
39    // allocation and initialization of Yq array and Pixel array
40    // ...
41
42    auto dout_buffer = xrt::bo (my_device, BUFF_SIZE_OUT,
43    xrt::bo::flags::normal, 0);
44    uint32_t* doutArray= dout_buffer.map<uint32_t*>();
45    std::cout << "INFO: Output buffer allocated at virtual memory space:
46    " << &doutArray << std::endl;
47
48    // Load AIE graph
49
50    std::cout << "Allocating aie graph..." << std::endl;
51    auto my_graph = xrt::graph(my_device, xclbin_uuid, "blint");
52    std::cout << "Aie graph allocation completed" << std::endl;
53
```

```
54    my_graph.reset();
```

Listing 6.4. Host application code - first part

The second part is shown in listing 6.5 and starts from allocating external buffers. The class object `xrt::aie::buffer` represents GMIO and external buffers [1] that facilitates data movement from global memory to the AI Engine and vice versa. Indeed, GMIO and External buffers work together to manage data flow efficiently by ensuring that large datasets can be processed effectively without overwhelming local memory resources [11].

The function `xrt::bo::write()` starts a DMA transfer between the memory buffer and the GMIO port.A for loop manages the execution of the AI Engine graph. `NRUN` is equal to 512 because processing one line of the output image at a time is handled. For each output image row, GMIOs are used to transfer to the AI Engine two pixels row (i.e. 1024x2 elements) and one row of x_q, y_q. `xrt::aie::buffer::async( )` start an asynchronous operation between the GMIO buffer and the global memory buffer. After the graph run, the output data are transferred from the GMIO output buffer to global memory. In addition, the host application calculates the new buffer offsets before starting a new iteration.

```
1     // creation of external buffer
2     auto xq_buffer_ext = xrt::aie::bo (my_device, BUFF_SIZE_XY,
3     xrt::bo::flags::normal, 0);
4     auto yq_buffer_ext = xrt::aie::bo (my_device, BUFF_SIZE_XY,
5     xrt::bo::flags::normal, 0);
6     auto img_buffer_ext = xrt::aie::bo (my_device, BUFF_SIZE_IMG,
7     xrt::bo::flags::normal, 0);
8     auto out_buffer = xrt::aie::bo (my_device, BUFF_SIZE_OUT,
9     xrt::bo::flags::normal, 0);
10
11    xq_buffer_ext.write(dinXqArray);
12    yq_buffer_ext.write(dinYqArray);
13    img_buffer_ext.write(dinImgArray);
14
15    int img_offset = 0;
16    int xq_offset = 0;
17    int yq_offset = 0;
18
19    for(int i= 0; i < NRUN; i ++){
20
```

[1] external buffer is a memory space allocated in DDR. Instead, shared buffers are always instantiated inside an AIE-ML tile or in a memory tile

```
        // GMIO run
        std::cout << "INFO: host app iteration: " << i << std::endl;
        img_buffer_ext.async("blint.gmioIn1", XCL_BO_SYNC_BO_GMIO_TO_AIE,
        2048 * sizeof(uint32_t), img_offset * sizeof(uint32_t));
        xq_buffer_ext.async("blint.gmioIn2", XCL_BO_SYNC_BO_GMIO_TO_AIE,
        512 * sizeof(uint32_t), xq_offset * sizeof(uint32_t));
        yq_buffer_ext.async("blint.gmioIn3", XCL_BO_SYNC_BO_GMIO_TO_AIE,
        512 * sizeof(uint32_t), yq_offset * sizeof(uint32_t));

        // Run AIE graph
        my_graph.run(1);
        my_graph.wait(0);

        auto dout_buffer_run = out_buffer.async("blint.gmioOut",
        XCL_BO_SYNC_BO_AIE_TO_GMIO, 512 * sizeof(uint32_t),
        xq_offset * sizeof(uint32_t));

        dout_buffer_run.wait();
        std::cout << "INFO: out buffer run completed!" << std::endl;

        //reading output data from AIE buffer to external buffer object
        unsigned array_lb = i*512;
        dout_buffer.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
        out_buffer.read(&doutArray[array_lb], 512 * sizeof(uint32_t),
        xq_offset * sizeof(uint32_t));

        //computation of memory offset for next iteration
        xq_offset = xq_offset + 512;
        yq_offset = yq_offset + 512;
        img_offset = img_offset + YRES*SCALE_FACTOR;

    }
    my_graph.end();
    return 0;
}
```

Listing 6.5. Host application code - second part

At the end of the loop (i.e., when the entire output image is produced), the graph is deallocated. The complete code has an additional part that reads the data written in the DDR's output buffer and compares it with reference data.

Chapter 7

Running the design on hardware

Usually, the first step after packaging is a hardware emulation on the host machine, before compiling and running the design on target hardware. **Hardware emulation** simulates a complete Versal adaptive SoC system composed of the AI Engines-ML, PS, and PL. The required co-simulation setup involves RTL, SystemC, and QEMU models:

- Embedded software code running on the PS is emulated using **QEMU**;
- Code running on the AI Engines is emulated using the **SystemC AI Engine simulator**;
- User PL kernels are simulated as **RTL code**;
- IP blocks in the hardware platform are simulated as **RTL or SystemC** based on the available models.

The Vitis hardware emulation is very close to the actual execution on hardware but not fully cycle-accurate. However, verifying the design behavior before running on the board is a fundamental step. In addition, debugging complex problems in this environment is more straightforward than in real hardware, exploiting different tools provided by Vitis™ IDE. For example, it is possible to check the system exploiting a co-simulation with Vivado, showing the waveforms of the signals in programmable logic.

7.1 Design 2: system execution

7.1.1 Hardware emulation

In figure 7.1 it is possible to see the hardware emulation waveforms for design 2.

Figure 7.1. Waveforms - design 2

As expected, the IP kernel and the AI Engine graph execute concurrently. It is possible to see read operations of the IP kernel on `m_axi` channel, getting the input pixel and the coordinates from the global memory, and the output transfer on AXI-stream interfaces to the AI Engine array. By zooming in on the s2mm interface signals (figure 7.2), it is also possible to see write operations to the global memory, transferring interpolated pixels from the bilinear interpolation kernel.

It is also possible to estimate the total time required to elaborate an input image from hardware emulation. In this example, the input image resolution is 1024 x 1024 pixels, and the design produces an output image rescaled by half (512 x 512). The time required for this operation is measured in figure 7.3 and is $\simeq 8.22$ ms.

The console output is shown in figure 7.4.

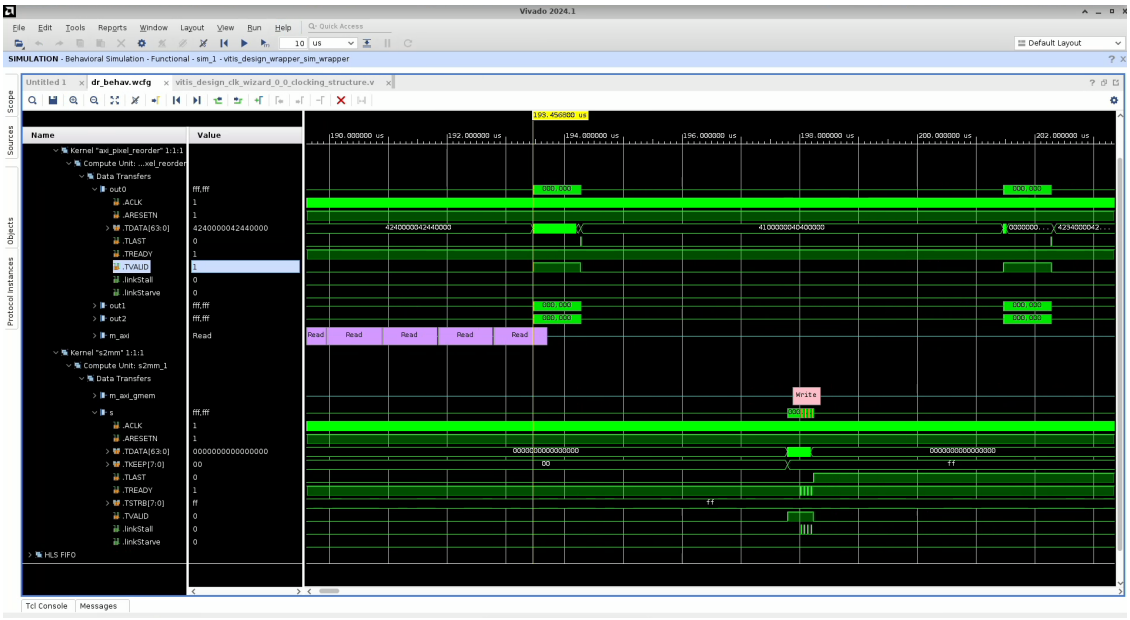

Figure 7.2. Waveforms - design 2

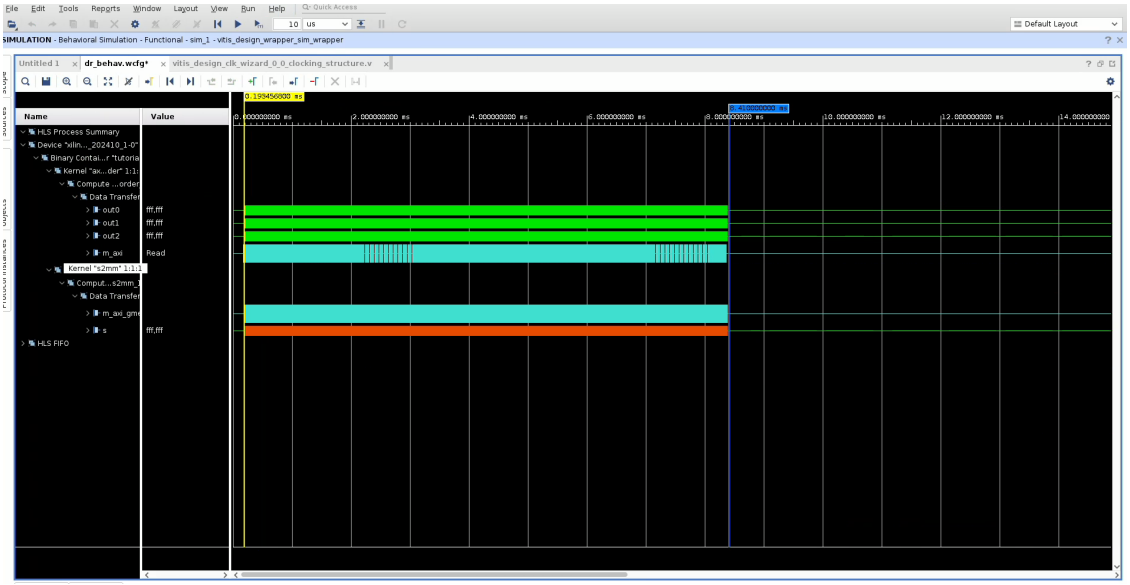

Figure 7.3. Waveforms - design 2

```
INFO: RTL IP kernel has been allocated!
INFO: Input memory virtual address 1 : 0xffff3d805000
INFO: Input memory virtual address 2: 0xffff3d605000
INFO: Output memory virtual address buffer: 0xffff3d505000
INFO: Memory allocated for input and output buffer
INFO: Setting IP data:
BASE_ADDR_1 : 1307574272
BASE_ADDR_2 : 1311768576
INFO: setting register: 20
INFO: setting register: 28
INFO:  28
Reg 0x14: 0
Reg 0x18: 1307574272
Reg 0x1c: 0
Reg 0x20: 1311768576
INFO: base address setup complete!
INFO: RTL Prescaler setup completed
[  191.194307] zocl-drm zyxclmm_drm:  ffff000001edf810 kds_add_context: Client pid(640) add context Domain(0) CU(0x1) shared(true)
s2mm run started
INFO: HLS Kernel run start
INFO: RTL Kernel run start
Allocating aie graph...
Aie graph allocation completed
Running graph for 1024 iterations
Waiting for completion...
[169524.955]XPlm_KeepAliveTask ERROR: PSM is not alive
[172161.082]PLM Error Status: 0x02080000
[  959.467245] cadence-qspi f1010000.spi: QSPI is still busy after 500ms timeout.
[  959.574526] cadence-qspi f1010000.spi: error -110 reading JEDEC ID
[  959.574936] cadence-qspi f1010000.spi: Setting dll delay error (-110)
Completed!

Waiting for kernels to end...

INFO: Kernel execution completed!

Verification of output data with respect to golden reference:

MATCH!
[11466.912256] zocl-drm zyxclmm_drm:  ffff000001edf810 kds_del_context: Client pid(640) del context Domain(0) CU(0x1)
[11466.922377] zocl-drm zyxclmm_drm:  ffff000001edf810 kds_del_context: Client pid(640) del context Domain(0) CU(0x0)
[11466.922991] [drm] bitstream 71615f4f-6f20-8bdd-6313-59e41b88398f unlocked, ref=0
[11466.976799] zocl-drm zyxclmm_drm: zocl_destroy_client: client exits pid(640)
versal-rootfs-common-20241:/mnt#
```

Figure 7.4. Waveforms - design 2

7.1.2 Hardware execution

After checking with hardware emulation, the design is packaged for execution on hardware. The PDI generated during the process is used to format an SD card plugged into the device and employed during system boot. If the trace option is enabled, the compiler inserts additional dedicated trace memory to capture events of interest at runtime. The operative systems translate trace data into trace files that can be opened in the Vitis analyzer tool to check the behavior of the design. The timeline trace is of significant use and is shown in figure 7.5.

The captured timeline confirms the behavior expected from hardware emulation and allows for maximum accuracy in capturing the execution time. The terminal output of the hardware run is shown in figure 7.6.

Figure 7.7 shows the trace generated during execution on hardware, taking into account also the execution times of software and XRT API calls, which instead is not considered by hardware emulation. The time required by the hardware execution coincides with that predicted by the simulation, which is

Figure 7.5. Hardware trace - design 2

approximately 8 ms. However, the total time required is larger if the initial latency is also considered. Of course, a certain amount of time is required

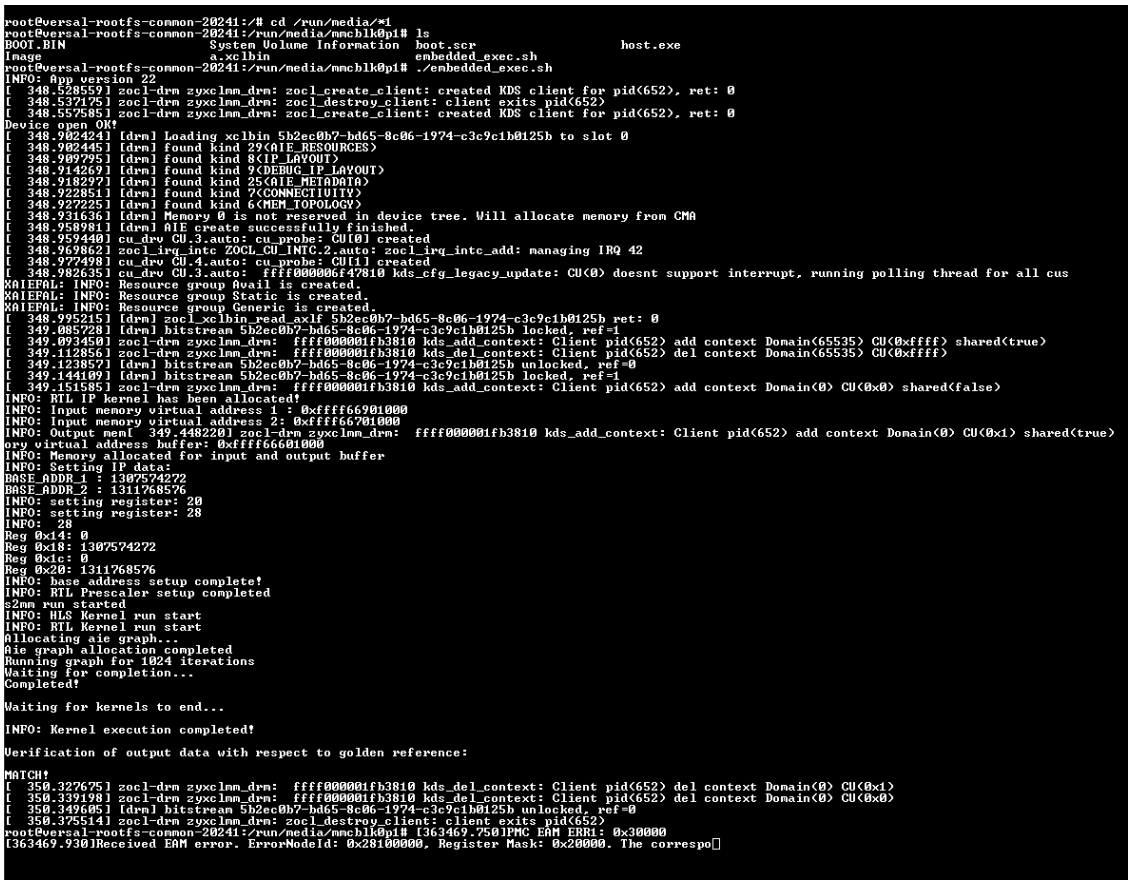

Figure 7.6. Hardware trace - design 2

to load the xclbin file and allocate resources. This latency is not considered in performance evaluation since these operations are performed just once at the beginning of the application execution.

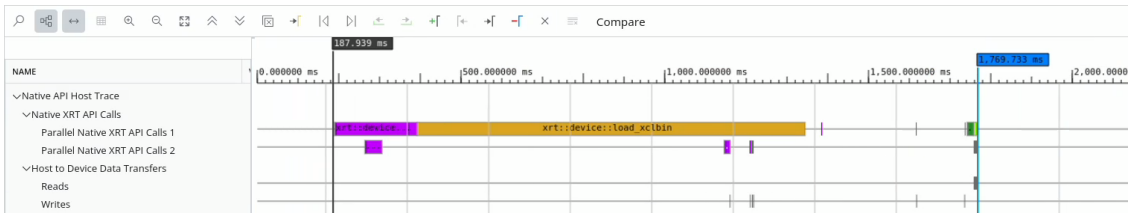

Figure 7.7. XRT trace - design 2

7.2 Design 1 system execution

7.2.1 Hardware emulation

Hardware emulation and hardware run have also been performed for design 1. Waveforms from hardware emulation show the expected behavior also for this system (figure 7.8). The RTL kernel is executed first, coherently with the design choice. The AI Engine graph starts only when the pre-processing is completed, as it is possible to see from the data traffic from/to mm2s interfaces. The total execution time in this case is $\simeq$ 14.31 ms.

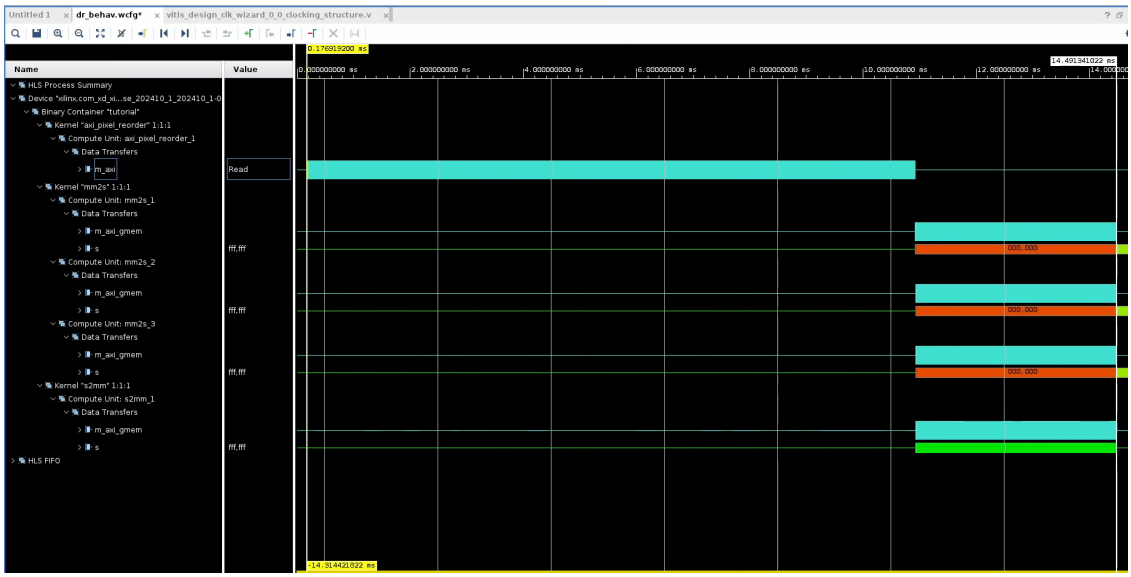

Figure 7.8. Hardware emulation waveforms - design 1

Figure 7.9 shows reading operations of image pixels and query coordinates from the DDR and consequently, the write operations to the global memory, exploiting the AXI4 Memory Mapped interface.

At the end of the RTL IP processing, the AI Engine graph starts the execution. Figure 7.10 shows HLS kernel running: mm2s interfaces read input data from global memory, and s2mm interface writes interpolated pixels in DDR.

7.2.2 Hardware run

Hardware run has been performed also for the first design, as shown in figure 7.11.

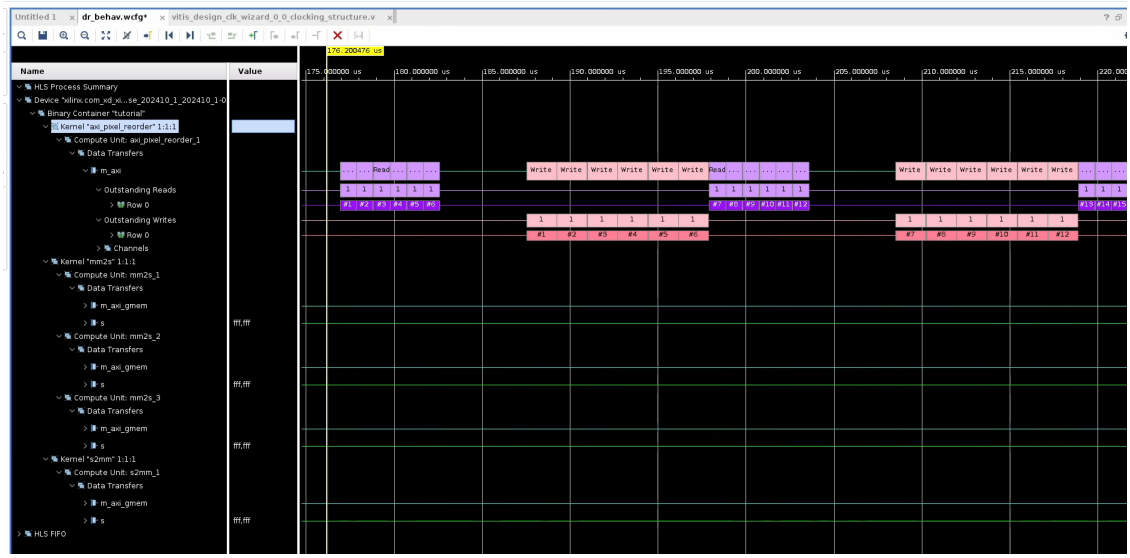

Figure 7.9. Hardware emulation waveforms of RTL IP processing - design 1

Figure 7.10. Hardware emulation waveforms of AI Engine execution - design 1

Also in this case, the measured execution time for the image scaling is similar to the time measured from the hardware emulation, approximately 18 ms. This time is around 4 milliseconds longer than that measured during hardware emulation. However, the emulation does not consider the additional

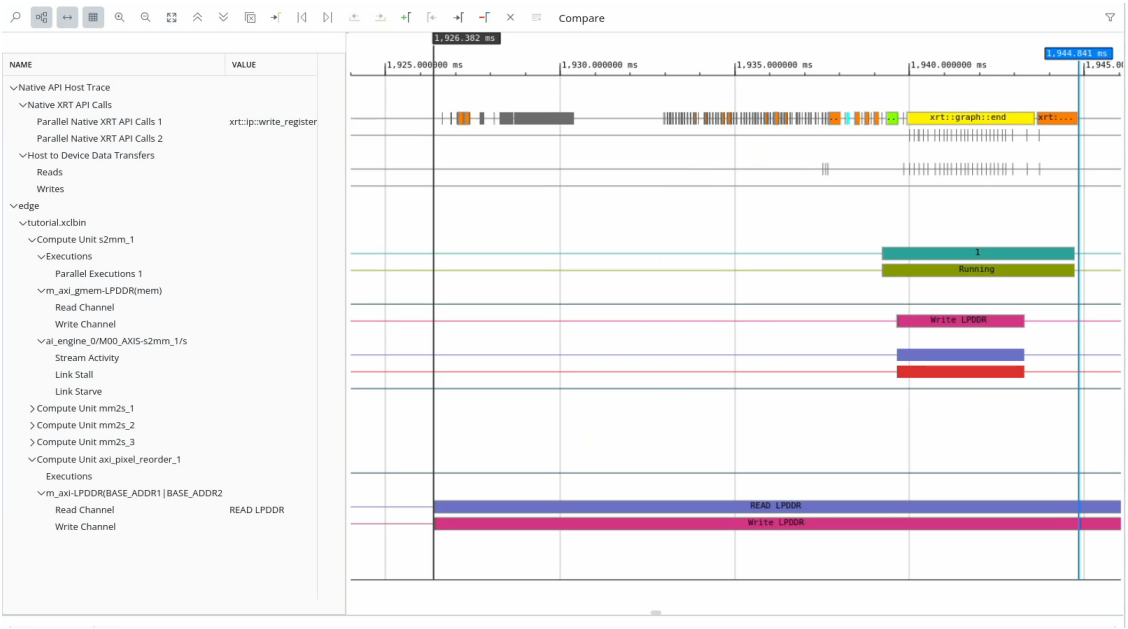

Figure 7.11. Hardware run - design 1

time required by the XRT API to perform some operations, such as starting the execution of the mm2s/s2mm interfaces and starting the graph, as shown from a deeper analysis of the timeline trace.

7.3 Design 3: system execution

As presented above, also for the third design, a hardware emulation was performed first, followed by hardware execution. Since this design has no kernel in programmable logic, it is interesting to present the results of the execution on hardware directly. Figure 7.12 shows a timeline trace full of events for the XRT API. Indeed, the role of software is predominant in this design. The host application must not only carry out the routine operations seen before but also calculate memory addresses, manage the copying of data from the GMIO buffers to memory, and initiate the execution of the graph AI Engine for each desired iteration.

The software here handles many previously demanded roles to the programmable logic, which is better suited to manage data transfer and memory address processing more efficiently. This fact explains the high execution time required to process an image, which is approximately 5

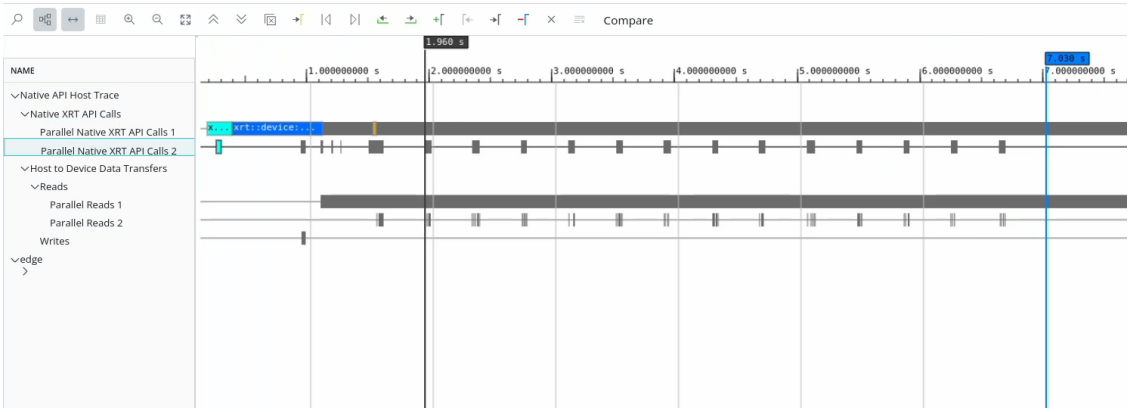

Figure 7.12. Hardware run - design 3

seconds, evaluated from the first execution of the AI Engine graph to the instant when the graph execution ends (figure 7.13).

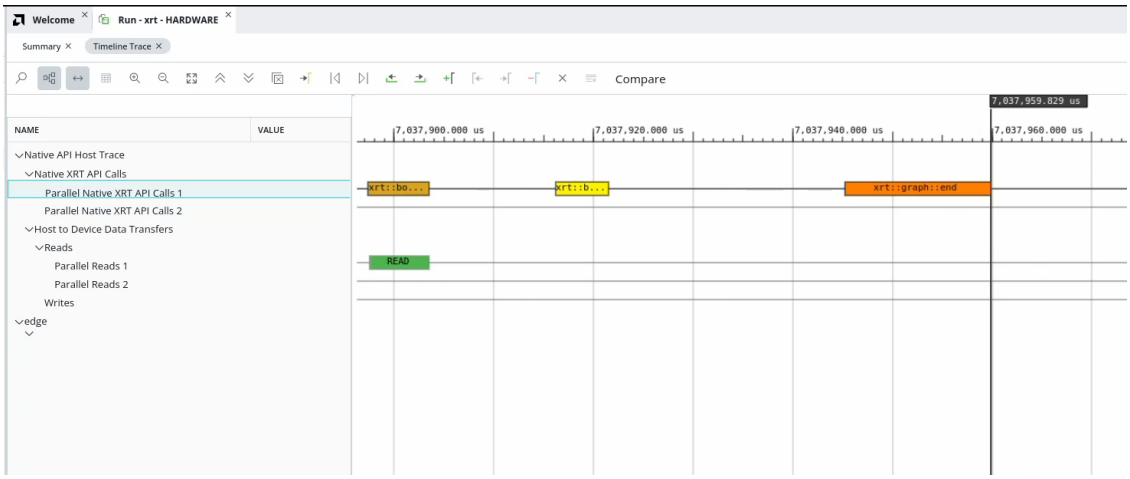

Figure 7.13. AI Engine graph end - design 3

From the hardware trace, it is possible to see the primary operations executed during the hardware run, such as data transfer between the global memory buffer to GMIO buffers after xclbin loading and the AI Engine graph run, as shown in figure 7.14 and 7.15 respectively.

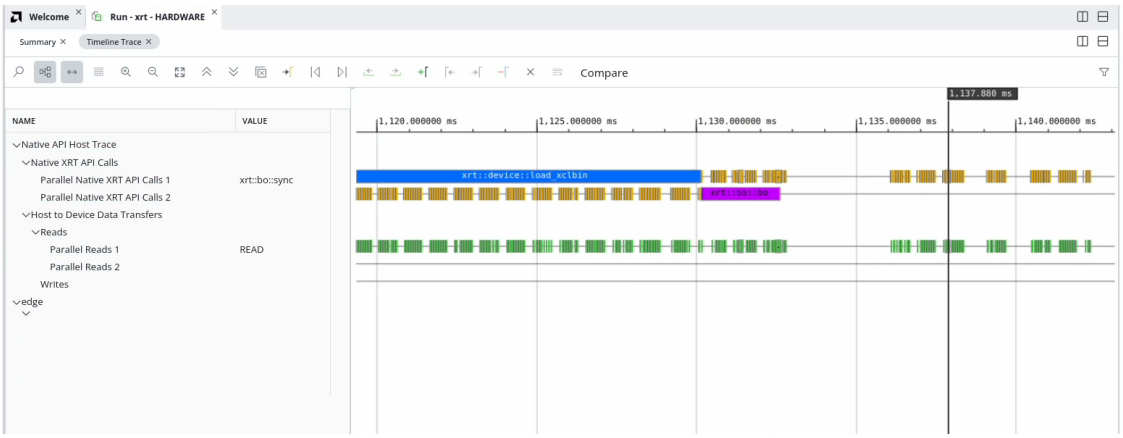

Figure 7.14. Data transfer after xclbin loading - design 3

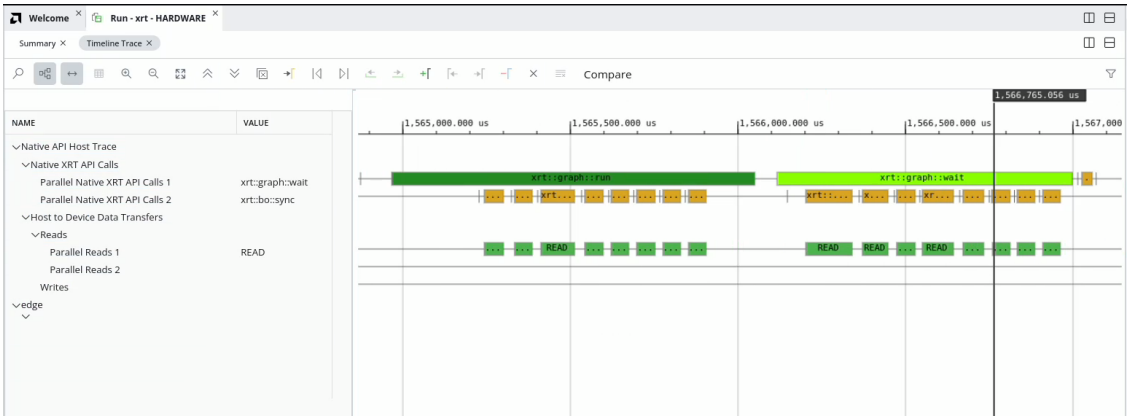

Figure 7.15. AI Engine graph run - design 3

7.4 Resources utilisation

The utilization report of the design can be checked from the Vitis project created after hardware linking. The results are summarized in tables 7.1, 7.2, 7.3, 7.4.

Table 7.1. Registers and LUT utilisation

Devices	**Registers**	**CLB LUTs**	**LUT as Logic**	**LUT as Memory**	**LOOK AHEAD8**	**SLICE**
VE2802	1,041,048	520,704	520,704	260,352	65,088	65,088
VE2302	300,544	150,272	150,272	75,136	18,784	18,784
% of VE2802	28.87%	28.86%	28.86%	28.86%	28.86%	28.86%
Design 1 (DDR)	12,861	9435	8173	1262	262	2726
% of VE2302	4,28%	6,28%	5,44%	1,68%	1,39%	14,51%
% of VE2802	1,24%	1,81%	1,57%	0,48%	0,40%	4,19%
Design 2 (AXI-Stream)	6,670	5,522	5,111	411	184	1,494
% of VE2302	2.22%	3.67%	3.40%	0.55%	0.98%	7.95%
% of VE2802	0.64%	1.06%	0.98%	0.16%	0.28%	2.30%
Design 3 (AI Engines only)	1,445	1,110	1,107	3	0	365
% of VE2302	0.48%	0.74%	0.74%	0.00%	0.00%	1.94%
% of VE2802	0.14%	0.21%	0.21%	0.00%	0.00%	0.56%

Table 7.2. CLB, BRAM, URAM and DSP utilisation

Devices	**CLB Registers**	**BRAM Tile**	**URAM**	**DSP Slices**
VE2802	1,041,048	600	264	11,312
VE2302	300,544	155	155	464
% of VE2802	28.87%	25.83%	58.71%	4.10%
Design 1 (DDR)	12,589	14	10	0
% of VE2302	4,19%	9,03%	6,45%	0,00%
% of VE2802	1,21%	2,33%	3,79%	0,00%
Design 2 (AXI-Stream)	6,670	17	4	0
% of VE2302	2.22%	10.97%	2.58%	0.00%
% of VE2802	0.64%	2.83%	1.52%	0.00%
Design 3 (AI Engines only)	1,445	0	0	0
% of VE2302	0.48%	0.00%	0.00%	0.00%
% of VE2802	0.14%	0.00%	0.00%	0.00%

Table 7.3. Clocking and NOC/AI Engine resources – Part 1

Devices	BUFG	XPLL	MMCM	PS NOC Master	AI Engine NOC Slave
VE2802	752	18	9	10	12
VE2302	256	8	4	10	6
% of VE2802	34.04%	44.44%	44.44%	100.00%	50.00%
Design 1 (DDR)	4	6	1	8	1
% of VE2302	1,56%	75,00%	25,00%	80,00%	16,67%
% of VE2802	0.53%	33.33%	11.11%	80.00%	8.33%
Design 2 (AXI-Stream)	4	6	1	8	1
% of VE2302	1.56%	75.00%	25.00%	80.00%	16.67%
% of VE2802	0.53%	33.33%	11.11%	80.00%	8.33%
Design 3 (AI Engines only)	4.00	6	1	8	1
% of VE2302	1.56%	75.00%	25.00%	80.00%	16.67%
% of VE2802	0.53%	33.33%	11.11%	80.00%	8.33%

Table 7.4. Clocking and NOC/AI Engine resources – Part 1

Type	AI ML NOC Slave	AI ML Engines	GTYP	PS9	BLI Registers
VE2802	12	340	8	1	54,752
VE2302	6	34	2	1	20,296
% of VE2802	50.00%	10.00%	25.00%	100.00%	37.07%
Design 1 (DDR)	1	2	0	1	816
% of VE2302	16.67%	5.88%	0.00%	100.00%	1.34%
% of VE2802	8.33%	0.59%	0.00%	100.00%	0.50%
Design 2 (AXI-Stream)	1	2	0	1	0
% of VE2302	16.67%	5.88%	0.00%	100.00%	0.00%
% of VE2802	8.33%	0.59%	0.00%	100.00%	0.00%
Design 3 (AI Engines only)	1	4	0	1	0
% of VE2302	16.67%	11.76%	0.00%	100.00%	0.00%
% of VE2802	8.33%	1.18%	0.00%	100.00%	0.00%

7.5 Final considerations and design comparison

7.5.1 Performances

After the execution on hardware, some final considerations can be made about the performances and utilization of the presented systems. The

performances of these three designs are summarized in table 7.5.

System	Execution time (ms)	throughput (FPS)
Design 1 (DDR buffer based)	18	55.56
Design 2 (AXI Stream interface)	8	125
Design 3 (AI Engines only)	5000	0.2

Table 7.5. Performance comparison, evaluated on a scaling of a 1024x1024 image by half

From the performance analysis, it is clear that:

- **The second design has the best performance**, capable of elaborating frame at a speed of 125 FPS. Indeed, as could easily be deduced from the outset, a design in which the different components can work simultaneously would be faster than one in which the graph only begins once the pre-processing operations have been completed. With a concurrent execution, PL elaboration time is absorbed in graph processing time, giving a higher throughput. In addition, the second design makes less global memory access since the intermediate data are not saved on DDR external buffers but are sent directly to the AI engine. Indeed, access to DDR is time- and energy-consuming. The elaboration rate also makes this design suitable for integration into a system where **real-time processing** is required, considering that common camera frame rates are between 15-60 FPS.

- The first design has less capability than the system's AXI4-Stream version. Lower performances are expected since the exchanges between the PL and the AI Engines pass through global memory, and the two phases of the interpolation operation are not simultaneous. Despite the lower performance, this design could also be used for real-time processing, provided the camera quality is medium/low, i.e., for 15 to 30 FPS cameras, or can be a good solution for **post-processing** image or video elaboration. However, an advantage is that this design is easily modifiable if different components are added. Since the PL and AI Engine parts are independent, making changes or adding other elements within this processing chain is easier.

- The third design, which uses only AI Engines for the bilinear interpolation implementation, is the worst in terms of performance

and unsuitable for real-time image or video processing. In any case, this design is an interesting experiment highlighting how an approach that exploits the **heterogeneity** of the platform **is the winning way forward**. Indeed, programmable logic and hardware accelerators are essential to developing high-performance designs. At the same time, this approach could be acceptable if there is the need to free up resources in PL to make room for other logic.

7.5.2 Resources utilization

The second aspect to consider regards **resources utilization**. This point is essential since, as introduced at the beginning, the three designs have been tested on the VEK280 Board, but the final target is the VE2302 Development Kit. For this reason, the first point is to understand whether the resources required by each design are not greater than those available on the xcve2302 SoC, which are fewer in number than those on the xcve2802. The tables presented in section 7.4 show that all three designs are suitable for implementation on the target board since they do not exceed the VE2302 available resources. Here, the information provided before is presented in charts to better compare the three different cases. Figures 7.16 and 7.17 compare the resource utilization in the three cases on VE2302-DK.

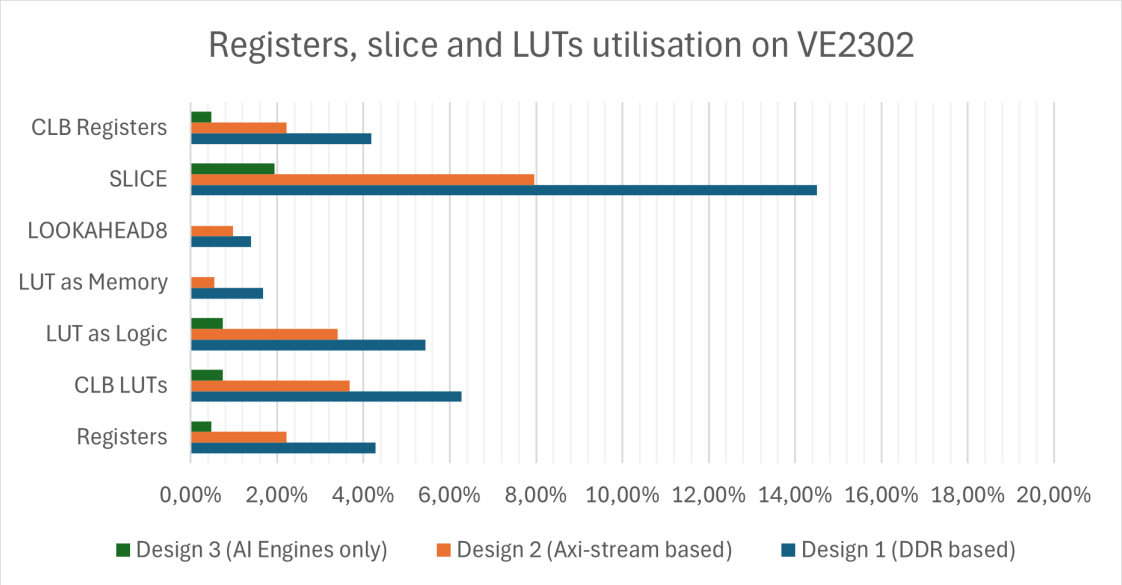

Figure 7.16. Registers, LUTs and SLICE utilization on VE2302

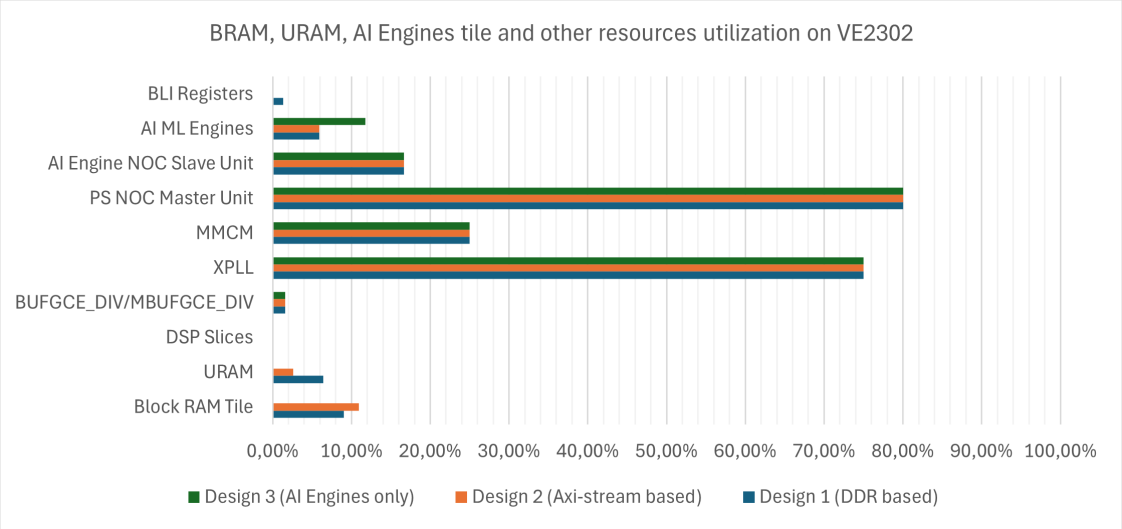

Figure 7.17. BRAM, URAM, AI Engines tiles utilization on VE2302

Figures 7.18 and 7.19 provide a comparison for VE2802 case.

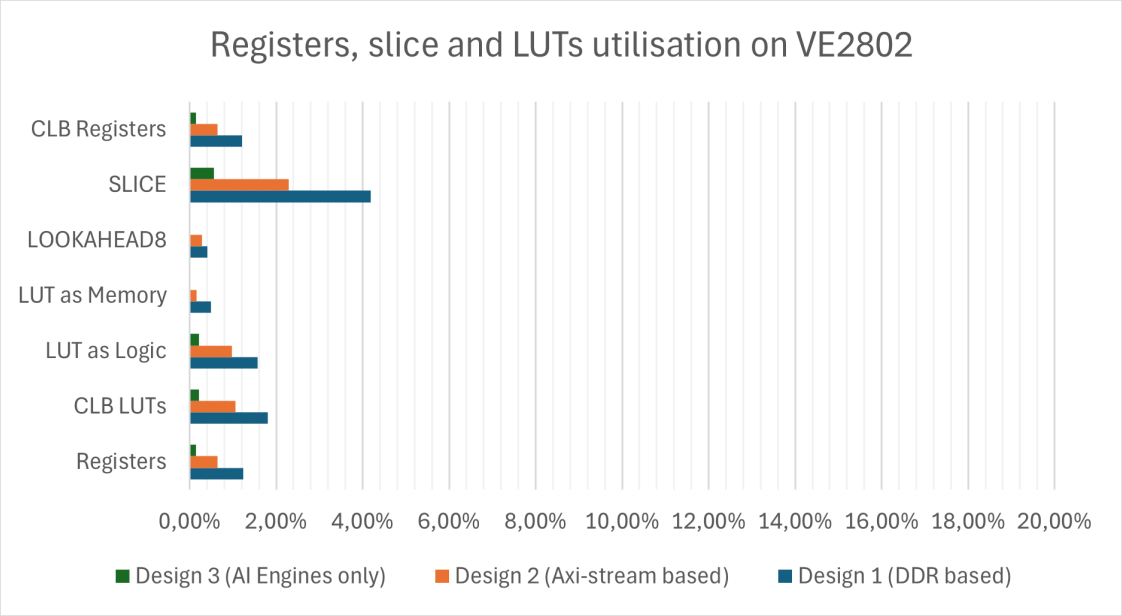

Figure 7.18. Registers, LUTs and SLICE utilization on VE2802

The third design is the most optimized in terms of PL resources: of course, this result is not a surprise since the approach relies totally on the AI Engines

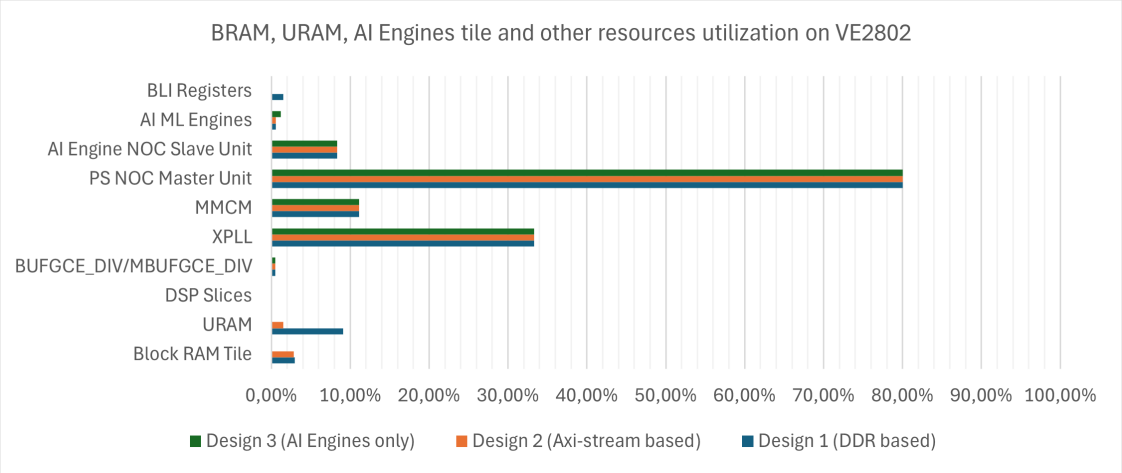

Figure 7.19. BRAM, URAM, AI Engines tiles utilization on VE2802

tile. At the same time, this design requires a higher number of AI Engine tiles since AIE kernels substitute the PL kernels' role. However, it is worth focusing on the other two designs, as the integration of resources and the performance achieved make them more suitable for implementing a reference design to be presented to final board users.

It is possible to make some considerations on the two heterogeneous designs:

- The second design is also the most optimized regarding resource utilization. Charts show a lower impact on registers, LUTs, and all the memory elements available in the programmable logic domain. Also, this design doesn't use mm2s interfaces, which are not part of the synthesis, and this helps to save space;

- The first design requires a slightly larger number of resources, especially registers, LUTs, URAM... This is explained considering that, in addition to the pixel reorder kernel and the s2mm interface, the PL domain also includes three mm2s interfaces that consume additional resources.

The advantages of lower resource utilization are lower power consumption and the possibility of saving programmable logic resources to add additional hardware to complete the reference design. For example, it is possible to configure a **Video Processing Subsystem** (VPS) to capture images from a camera connected to the board.

All these considerations are summarized in table 7.6. The third design is not considered since it is not interesting for the implementation of the VE2302-DK reference design, as explained above.

Design 1 (intermediate DDR buffers)	**Design 2 (AXI-4 Stream interface)**
Total execution time: 8 ms (approximately 125 FPS)	Total execution time: 18 ms (about 55 FPS)
Higher resources utilisation in PL	Lower resources utilisation in PL
Higher number of global memory accesses	Lower number of global memory accesses
All the system components are independent	System components are not independent of each other

Table 7.6. Design 1 and design 2 comparison

From this analysis, the choice for VE2302-DK reference design is represented by **Design 2**, because it is better for performance and utilization.

Chapter 8

Conclusions and future developments

The aim of this thesis, namely, the **Development of demo designs for the VE2302 development kit**, has been satisfied. A complete system based on programmable logic, processing systems, and AI Engines-ML has been realized by analyzing different designs and their features.

This design is a starting point for all users interested in developing and discovering the potential of the VE2302-DK board, making it easier to understand the integrated development process on the Versal SoC. As analyzed at the beginning, the SoCs of the Versal family are particularly powerful but simultaneously complex. However, a reference design can help users become familiar with the hardware when developing custom applications. Furthermore, the work carried out lends itself well to presenting the potential of the hardware under analysis.

In addition, it can be an interesting starting point for those who want to develop an image processing system. This design provides an excellent basis for placing various elements to process incoming video streams. The bilinear interpolation algorithm offers a flexible ground for implementing various transformations, not only scaling. In fact, changing the output coordinate grid is sufficient to obtain different transformations, even more complex ones (rotations, translations, cutouts, etc.). Moreover, this algorithm is well suited to artificial intelligence processing. A couple of examples of potential applications of this case are:

- **Image pre-processing** to resize for AI models: it is common for various artificial intelligence models processing images to have precise

specifications on the size of the input frame, which must then first be resized and modified.

- **Region of interest selection** for dual inference pipelines. ROI is used to choose a specific portion to analyze within an image. In machine learning, during image pre-processing, it is common to crop one or more ROIs to reduce the data to be processed. Or, for object detection techniques, multiple candidate ROIs are generated to be classified. With this design, ROI selection can be easily implemented by giving the coordinate grid corresponding to the portion of the region of interest and using this project as an accelerator for the cropping stage.

These are just two examples, but they demonstrate the flexibility and multitude of applications that this design can have.

In this perspective, the following are some ideas for possible future developments:

- Adding a DPU or NPU for **AI model inference**. One possibility is to add AMD Vitis™ AI, an Integrated Development Environment that can be leveraged to accelerate AI inference on AMD adaptable platforms. It is designed with high efficiency and ease of use, unleashing the full potential of AI acceleration on AMD adaptable SoCs [12]. This would allow the accelerator to be combined with AI image processing models, as described above.

- Extend the existing design by adding all the elements for a **video pipeline**. The idea is to receive frames from an external source, use the bilinear interpolation algorithm for processing, and add a display element showing the elaborated video frames.

Chapter 9

Appendix

9.1 Appendix A

9.1.1 bilinear_graph.h

```
//
// Copyright (C) 2024, Advanced Micro Devices, Inc. All rights reserved.
// SPDX-License-Identifier: MIT
//
// Original code by: Richard Buz

// Modified version for VEK280 Evaluation board by: Francesca Franzese

#pragma once

#include <adf.h>
#include "bilinear_kernel.h"
#include "buffers.h"

using namespace adf;

class bilinear_graph : public adf::graph {
private:
  kernel bli_krnl[NCORE];

public:
  std::array<input_plio, NCORE> iplio_A, iplio_B, iplio_C;
  std::array<output_plio, NCORE> oplio;

  bilinear_graph()
  {
```

```
27    for (int i = 0; i < NCORE; i++) {
28      std::string iplio_A_name = "DIN_" + std::to_string(i) + "_A";
29      std::string iplio_B_name = "DIN_" + std::to_string(i) + "_B";
30      std::string iplio_C_name = "DIN_" + std::to_string(i) + "_C";
31      std::string oplio_name = "DOUT_" + std::to_string(i);
32      std::string iplio_A_file = "data/input_" + std::to_string(i+1)
33      + "_A.txt";
34      std::string iplio_B_file = "data/input_" + std::to_string(i+1)
35      + "_B.txt";
36      std::string iplio_C_file = "data/input_" + std::to_string(i+1)
37      + "_C.txt";
38      std::string oplio_file = "data/output_" + std::to_string(i+1)
39      + "_aie.txt";
40
41      iplio_A[i] = input_plio::create(iplio_A_name, plio_64_bits,
42        iplio_A_file, 156.25);
43      iplio_B[i] = input_plio::create(iplio_B_name, plio_64_bits,
44        iplio_B_file, 156.25);
45      iplio_C[i] = input_plio::create(iplio_C_name, plio_64_bits,
46        iplio_C_file, 156.25);
47      oplio[i] = output_plio::create(oplio_name, plio_64_bits,
48        oplio_file, 156.25);
49
50      bli_krnl[i] = kernel::create_object<bilinear_kernel>();
51
52      connect(iplio_A[i].out[0], bli_krnl[i].in[0]);
53      connect(iplio_B[i].out[0], bli_krnl[i].in[1]);
54      connect(iplio_C[i].out[0], bli_krnl[i].in[2]);
55      connect(bli_krnl[i].out[0], oplio[i].in[0]);
56
57      source(bli_krnl[i]) = "src/bilinear_kernel.cpp";
58
59      runtime<ratio>(bli_krnl[i]) = 0.9;
60    }
61  }
62 };
```

Listing 9.1. bilinear kernel graph code

9.1.2 bilinear_kernel.h

```
//
// Copyright (C) 2024, Advanced Micro Devices, Inc. All rights reserved.
// SPDX-License-Identifier: MIT
//
// Original code by: Richard Buz

// Modified version for VEK280 Evaluation board by: Francesca Franzese

#pragma once

#include <aie_api/aie.hpp>
#include "buffers.h"

using namespace adf;

class bilinear_kernel
{
private:

public:
bilinear_kernel() {}
void interp(input_buffer<int32, extents<BUFFER_SIZE_IN>>& __restrict in_A,
           input_buffer<int32, extents<BUFFER_SIZE_IN>>& __restrict in_B,
           input_buffer<int32, extents<BUFFER_SIZE_IN>>& __restrict in_C,
           output_buffer<int32, extents<BUFFER_SIZE_OUT>>& __restrict out);
static void registerKernelClass()
{
    REGISTER_FUNCTION(bilinear_kernel::interp);
}
};
```

Listing 9.2. bilinear kernel header

9.1.3 bilinear_kernel.cpp

```
//
// Copyright (C) 2024, Advanced Micro Devices, Inc. All rights reserved.
// SPDX-License-Identifier: MIT
//
// Original code by: Richard Buz

// Modified version for VEK280 Evaluation board by: Francesca Franzese

#include "bilinear_kernel.h"
#include "aie_api/accum.hpp"
#include "aie_api/vector.hpp"
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>

void bilinear_kernel::interp(input_buffer<int32, extents<BUFFER_SIZE_IN>>&
                             __restrict in_A,
                             input_buffer<int32, extents<BUFFER_SIZE_IN>>&
                             __restrict in_B,
                             input_buffer<int32, extents<BUFFER_SIZE_IN>>&
                             __restrict in_C,
                             output_buffer<int32, extents<BUFFER_SIZE_OUT>>&
                             __restrict out)
{
    // iterators for input & output buffers
    auto pInA = aie::begin_vector<8>(in_A);
    auto pInB = aie::begin_vector<8>(in_B);
    auto pInC = aie::begin_vector<8>(in_C);
    auto pOut = aie::begin_vector<8>(out);

    for (unsigned i = 0; i < PXLPERGRP/8; i++)
        chess_prepare_for_pipelining
        chess_loop_count(PXLPERGRP/8)
    {
        // get data for first x interpolation
        aie::vector<float, 8> xfrac = (*pInA++).cast_to<float>();
        aie::vector<float, 8> p11 = (*pInB++).cast_to<float>();
        aie::vector<float, 8> p21 = (*pInC++).cast_to<float>();

        aie::accum<accfloat, 8> p11_acc;
        p11_acc.from_vector(p11);

        // compute first x interpolation
        aie::accum<accfloat, 8> tempy1 = aie::mac(p11_acc,xfrac,p21);
        p11= p11_acc.to_vector();
```

```
45          aie::accum<accfloat, 8> pxy1 = aie::msc(tempy1, xfrac, p11);
46
47          // get data for second x interpolation
48          aie::vector<float, 8> p12 = (*pInB++).cast_to<float>();
49          aie::vector<float, 8> p22 = (*pInC++).cast_to<float>();
50
51          aie::accum<accfloat, 8> p12_acc;
52          p12_acc.from_vector(p12);
53
54          // compute second x interpolation
55          aie::accum<accfloat, 8> tempy2 = aie::mac(p12_acc,xfrac,p22);
56          p12 = p12_acc.to_vector();
57          aie::accum<accfloat, 8> pxy2 = msc(tempy2,xfrac,p12);
58
59          // get data for y interpolation
60          aie::vector<float, 8> yfrac = (*pInA++).cast_to<float>();
61
62          // compute y interpolation
63          aie::accum<accfloat, 8> tempxy = aie::mac(pxy1,yfrac,
64              pxy2.to_vector());
65          aie::accum<accfloat, 8> pxy = aie::msc(tempxy, yfrac,
66              pxy1.to_vector());
67          // write interpolated pixels to output
68          *pOut++ = aie::vector_cast<int32>(pxy.to_vector());
69
70      }
71  }
```

Listing 9.3. bilinear kernel complete code

9.1.4 config.h

```
//
// Copyright (C) 2024, Advanced Micro Devices, Inc. All rights reserved.
// SPDX-License-Identifier: MIT
//
// Original code by: Richard Buz

// Modified version for VEK280 Evaluation board by: Francesca Franzese

#pragma once

#define NCORE 1
#define NRUN 1024
#define PXLPERGRP 256
```

Listing 9.4. configuration file

9.1.5 buffers.h

```
//
// Copyright (C) 2024, Advanced Micro Devices, Inc. All rights reserved.
// SPDX-License-Identifier: MIT
//
// Original code by: Richard Buz

// Modified version for VEK280 Evaluation board by: Francesca Franzese

#pragma once

#include "config.h"

#define BUFFER_SIZE_IN (2 * PXLPERGRP)
#define BUFFER_SIZE_OUT (PXLPERGRP)
```

Listing 9.5. buffer dimensions file

9.1.6 reorder_kernel.cpp

```
1
2  // Author: Francesca Franzese
3  // Company: Tria Technologies
4
5  #include <adf.h>
6  #include "buffers.h"
7  #include "reorder_kernel.h"
8
9  #include <aie_api/aie.hpp>
10 #include <aie_api/aie_adf.hpp>
11 #include "aie_api/vector.hpp"
12
13
14 void reorder_kernel::reorder(input_buffer<int32 , adf::extents<BUFF_DIM_1>> & __restrict i
15                              input_buffer<int32, adf::extents<BUFF_DIM_2>> & __restrict in2, /
16                              input_buffer<int32, adf::extents<BUFF_DIM_2>> & __restrict in3, /
17                              output_buffer<int32, adf::extents<BUFF_DIM_INT>> & __restrict out
18                              output_buffer<int32, adf::extents<BUFF_DIM_INT>> & __restrict out
19                              output_buffer<int32, adf::extents<BUFF_DIM_INT>> & __restrict out
20 {
21     // iterators to access input and output buffer
22     auto inPtrP=aie::begin_random_circular(in1); //random access for pixel lut
23     auto inPtrXq=aie::begin(in2);
24     auto inPtrYq=aie::begin(in3);
25
26     auto outPtr1=aie::begin_vector<8>(out1);
27     auto outPtr2=aie::begin_vector<8>(out2);
28     auto outPtr3=aie::begin_vector<8>(out3);
29
30     // inspired by matlab code:
31     // declaration of test vector
32     aie::vector<float, 8> xfrac_v = aie::zeros<float,8>();
33     aie::vector<float, 8> yfrac_v = aie::zeros<float,8>();
34     aie::vector<float, 8> p11_v = aie::zeros<float,8>();
35     aie::vector<float, 8> p12_v = aie::zeros<float,8>();
36     aie::vector<float, 8> p21_v = aie::zeros<float,8>();
37     aie::vector<float, 8> p22_v = aie::zeros<float,8>();
38
39     int samp_cnt = 0;
40
41     /*#if defined(__AIESIM__) || defined(__X86SIM__)
42     std::cout<< std::endl;
43     std::cout<<"REORDER KERNEL EXECUTION " << std::endl;
44     #endif */
```

```
45
46  //iterate over query coordinates buffer (for each query, we need to determine xfrac, y
47   for (unsigned i = 0; i < YRES_OUT; i++)
48      chess_prepare_for_pipelining
49      chess_loop_count(YRES_OUT)
50  {
51
52      //get query and get integer part
53      auto xq = (*inPtrXq++); //prendo query pixel e mando avanti iteratore (+ cast a flo
54      auto yq = (*inPtrYq++);
55
56      float xq_fp = *reinterpret_cast<float*>(&xq);
57      float yq_fp = *reinterpret_cast<float*>(&yq);
58
59      auto xint = std::floor(xq_fp);
60      auto yint = std::floor(yq_fp);
61
62      auto xfrac = xq_fp - xint;
63      auto yfrac = yq_fp - yint;
64
65      /*#if defined(__AIESIM__) || defined(__X86SIM__)
66      std::cout << "xq: " << xq << std::endl;
67      std::cout << "yq: " << yq << std::endl;
68      std::cout << "xq_fp: " << xq_fp << std::endl;
69      std::cout << "yq_fp: " << yq_fp << std::endl;
70      std::cout << "xfrac: " << xfrac << std::endl;
71      std::cout << "yfrac: " << yfrac << std::endl;
72      std::cout << "xint " << xint << std::endl;
73      std::cout << "yint: " << yint << std::endl;
74      #endif*/
75
76
77      //get index of p11, p12, p21, p22 in input buffer
78      uint32_t p11_idx = yint; //xint*YRES+yint;
79      uint32_t p12_idx = yint+1; //xint*YRES+yint+1;
80      uint32_t p21_idx = YRES+yint;//(xint+1)*YRES+yint;
81      uint32_t p22_idx = YRES+yint+1;//(xint+1)*YRES+yint+1;
82
83      inPtrP += p11_idx;
84      auto p11 = *inPtrP;
85      float p11_fp = *reinterpret_cast<float*>(&p11);
86
87      inPtrP += 1; //p12 index is p11_index + 1
88      auto p12 = *inPtrP;
89      float p12_fp = *reinterpret_cast<float*>(&p12);
90      inPtrP=aie::begin_random_circular(in1); //position the pointer again at the startin
```

```
91
92          inPtrP += p21_idx;
93          auto p21= *inPtrP;
94          float p21_fp = *reinterpret_cast<float*>(&p21);
95
96          inPtrP += 1;
97          auto p22= *inPtrP;
98          float p22_fp = *reinterpret_cast<float*>(&p22);
99          inPtrP=aie::begin_random_circular(in1); //position the pointer again at the startin
100
101
102         //insert element in 8-vector
103         xfrac_v.set(xfrac, samp_cnt);
104         yfrac_v.set(yfrac, samp_cnt);
105         p11_v.set(p11_fp, samp_cnt);
106         p12_v.set(p12_fp, samp_cnt);
107         p21_v.set(p21_fp, samp_cnt);
108         p22_v.set(p22_fp, samp_cnt);
109
110
111         if(samp_cnt == 7){
112
113             *outPtr1++ = aie::vector_cast<int32>(xfrac_v); //xfrac, yfrac vanno nello stess
114             *outPtr1++ = aie::vector_cast<int32>(yfrac_v);
115
116             *outPtr2++ = aie::vector_cast<int32>(p11_v);
117             *outPtr2++ = aie::vector_cast<int32>(p12_v);
118
119             *outPtr3++ = aie::vector_cast<int32>(p21_v);
120             *outPtr3++ = aie::vector_cast<int32>(p22_v);
121
122             samp_cnt = 0;
123
124         /*#if defined(__AIESIM__) || defined(__X86SIM__)
125         std::cout << "REORDER KERNEL, VECTORS: " << std::endl;
126         std::cout << "xfrac: " << std::endl;
127         for (int i = 0; i < 8; i++) {
128
129             printf("%d ", static_cast<int32_t>(xfrac_v[i]));
130
131         }
132         std::cout << std::endl;
133
134         std::cout << "yfrac: " << std::endl;
135         for (int i = 0; i < 8; i++) {
136
```

```
137             printf("%d ", static_cast<int32_t>(yfrac_v[i]));
138 
139         }
140         std::cout << std::endl;
141 
142         std::cout << "p11: " << std::endl;
143         for (int i = 0; i < 8; i++) {
144 
145             printf("%d ", static_cast<int32_t>(p11_v[i]));
146 
147         }
148         std::cout << std::endl;
149 
150         std::cout << "p12: " << std::endl;
151         for (int i = 0; i < 8; i++) {
152 
153             printf("%d ", static_cast<int32_t>(p12_v[i]));
154 
155         }
156         std::cout << std::endl;
157 
158         std::cout << "p21: " << std::endl;
159         for (int i = 0; i < 8; i++) {
160 
161             printf("%d ", static_cast<int32_t>(p21_v[i]));
162 
163         }
164         std::cout << std::endl;
165 
166         std::cout << "p22: " << std::endl;
167         for (int i = 0; i < 8; i++) {
168 
169             printf("%d ", static_cast<int32_t>(p22_v[i]));
170 
171         }
172         std::cout << std::endl;
173 
174         #endif*/
175 
176         }
177         else{
178 
179             samp_cnt++;
180         }
181 
182 
```

```
183     }
184
185 }
```

Listing 9.6. reorder kernel (design 3)

9.2 Appendix B

9.2.1 s2mm.cpp

```
/*
Copyright (C) 2023, Advanced Micro Devices, Inc. All rights reserved.
SPDX-License-Identifier: X11
*/

#include <ap_int.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>


extern "C" {

void s2mm(ap_int<32>* mem, hls::stream<ap_axis<32, 0, 0, 0> >& s,
          int size) {

    #pragma HLS INTERFACE m_axi port=mem offset=slave bundle=gmem

    #pragma HLS interface axis port=s

    #pragma HLS INTERFACE s_axilite port=mem bundle=control
    #pragma HLS INTERFACE s_axilite port=size bundle=control
    #pragma HLS interface s_axilite port=return bundle=control

    for(int i = 0; i < size; i++) {
        #pragma HLS PIPELINE II=1
        ap_axis<32, 0, 0, 0> x = s.read();
        mem[i] = x.data;
    }
}
}
```

Listing 9.7. s2mm HLS code [10]

9.3 Appendix C

9.3.1 Input and golden file matlab script

```
% Author: Francesca Franzese
% Bilinear interpolation for image resizing
%with interpolation coordinate grid

%% get pixels grid of original image
fname = '../images/epyc.jpg';
I = imread(fname);

%zero padding
I_padd = zeros(1024,1024);
I_padd(1:size(I,1), 1:size(I,2)) = I;
I = I_padd;

[resy, resx] = size(I);
xshft = (resx-1)/2;
yshft = (resy-1)/2;

% get image LUT indexes min and max values
[ysz, xsz] = size(I);
lut_xmin = 0;
lut_xmax = xsz - 1;
lut_ymin = 0;
lut_ymax = ysz - 1;

% Convert LUT to single precision float format and
%initialize output image to all zeros.

ImLUT = single(I(:));
resized = single(zeros(size(Yq,1) * size(Xq,2), 1));
simLUT = zeros(512*512, 1);
numpxl = 256;
max_pxl_per_proc = numpxl * ceil(size(Yq,1) * size(Xq,2)
                / numpxl);

input_tv_A = repmat(typecast(realmax('single'), 'int32'),
                    2*max_pxl_per_proc, 1);
input_tv_B = repmat(typecast(realmax('single'), 'int32'),
                    2*max_pxl_per_proc, 1);
```

```
input_tv_C = repmat(typecast(realmax('single'), 'int32'),
                    2*max_pxl_per_proc, 1);
output_tv = repmat(typecast(realmax('single'), 'int32'),
                    max_pxl_per_proc, 1);

%% get reference data for PL simulation
itv_idx = 1;
otv_idx = 1;
samp_cnt = 0;
tvx = single(zeros(8,1));
tvy = single(zeros(8,1));
tvp11 = single(zeros(8,1));
tvp12 = single(zeros(8,1));
tvp21 = single(zeros(8,1));
tvp22 = single(zeros(8,1));
tvo = single(zeros(8,1));
in_bnd_cnt = 0;

output_idx=1;
for ix = 1:size(Xq_vec)
        % get query coordinates
        xval = Xq_vec(ix);
        yval = Yq_vec(ix);

         % Points requiring extrapolation are set to zero.
        is_interp = xval >= lut_xmin && xval < lut_xmax
        && yval >= lut_ymin && yval < lut_ymax;

        if is_interp

         % get reference point data from camera image
            x1 = int32(floor(xval));
            y1 = int32(floor(yval));

            x1_v(output_idx) = x1;
            y1_v(output_idx) = y1;

            P11 = ImLUT(x1*(lut_ymax+1)+y1+1);
            P12 = ImLUT(x1*(lut_ymax+1)+y1+2);
            P21 = ImLUT((x1+1)*(lut_ymax+1)+y1+1);
            P22 = ImLUT((x1+1)*(lut_ymax+1)+y1+2);

```

```
81              % fractional part for interpolation
82              xfrac = single(xval - floor(xval));
83              yfrac = single(yval - floor(yval));
84
85               % perform bilinear interpolation
86              blint_in = [P11 P12 P21 P22 xfrac yfrac];
87              pxl_intrp = blint(blint_in,4);
88
89          else
90              P11 = single(0);
91              P12 = single(0);
92              P21 = single(0);
93              P22 = single(0);
94              xfrac = single(0);
95              yfrac = single(0);
96
97              blint_in = [ P11 P12  P21 P22 xfrac yfrac];
98              pxl_intrp = blint(blint_in,4);
99
100         end
101
102         tvx(mod(samp_cnt,8) + 1) = xfrac;
103         tvy(mod(samp_cnt,8) + 1) = yfrac;
104         tvp11(mod(samp_cnt,8) + 1) = P11;
105         tvp12(mod(samp_cnt,8) + 1) = P12;
106         tvp21(mod(samp_cnt,8) + 1) = P21;
107         tvp22(mod(samp_cnt,8) + 1) = P22;
108
109         if mod(samp_cnt,8) == 7
110             input_tv_A(itv_idx:itv_idx+15) =
111                 typecast([tvx; tvy], 'int32');
112             input_tv_B(itv_idx:itv_idx+15) =
113                 typecast([tvp11; tvp12], 'int32');
114             input_tv_C(itv_idx:itv_idx+15) =
115                 typecast([tvp21; tvp22], 'int32');
116             itv_idx = itv_idx + 16;
117             output_tv((otv_idx:otv_idx+7)) =
118                 typecast(tvo, 'int32');
119             otv_idx = otv_idx + 8;
120         end
121
122         % update counters
```

```
123         samp_cnt = samp_cnt + 1;
124
125         if samp_cnt == max_pxl_per_proc
126             samp_cnt = 0;
127             itv_idx = 1;
128             otv_idx = 1;
129         end
130
131         resized(output_idx) = pxl_intrp;
132         output_idx = output_idx +1;
133 end
134
135 %% creation of files for simulation and check
136 coordinates=[Xq_vec Yq_vec];
137
138 %creation of Xq array
139 fname = sprintf('./xq_array.h');
140 fid = fopen(fname,'w');
141 fprintf(fid,'uint32_t xq_array[512*512] ={');
142 for i = 1:size(Xq_vec,1)
143     if fid > 0
144         value=num2hex(single(Xq_vec(i)));
145             if i == size(Xq_vec,1)
146                 fprintf(fid,'0x%s};',value);
147             else
148                 fprintf(fid,'0x%s,\n',value);
149             end
150     end
151 end
152 fclose(fid);
153
154 %% creation of Yq array
155 fname = sprintf('./yq_array.h');
156 fid = fopen(fname,'w');
157 fprintf(fid,'uint32_t yq_array[512*512] ={ ');
158 for i = 1:size(Yq_vec,1)
159     if fid > 0
160         value=num2hex(single(Yq_vec(i)));
161             if i == size(Yq_vec,1)
162                 fprintf(fid,'0x%s};',value);
163             else
164                 fprintf(fid,'0x%s,\n',value);
```

```
            end
    end
end
fclose(fid);

% pixel_array for aie only
fname = sprintf('./pixel_array.h');
fid = fopen(fname,'w');
fprintf(fid,'uint32_t pxl_array[1024*1024] ={');
for i = 1:size(ImLUT,1)
    if fid > 0
        value = num2hex(ImLUT(i,1));
            if i == size(ImLUT,1)
                fprintf(fid,'0x%s};',value);
            else
                fprintf(fid,'0x%s,\n',value);
            end
    end
end
fclose(fid);

ImgOut = reshape(resized, size(Yq,1), size(Xq,2));

% golden reference
fname = sprintf('./golden.h');
fid = fopen(fname,'w');
if fid > 0
    fprintf(fid,'%d, %d,\n',int32(ImgOut));
end
fclose(fid);

% padding output image to see difference with
% original image
resized_padd = zeros(1024,1024);
resized_padd(1:size(ImgOut,1), 1:size(ImgOut,2)) = ImgOut;
ImgOut = resized_padd;

%conversione in numeri fixed point 32 bit 16.16
fp_coords = fi(coordinates, true, 32, 16);
fp_pxl = fi(ImLUT, true, 32, 16);

assignin('base', "fp_pxl_LUT", fp_pxl);
```

```
207 fp_pxl = reshape(fp_pxl, 2, [])';
208
209 % pixel input LUT
210 fname = sprintf('./input_pixel.h');
211 fid = fopen(fname,'w');
212
213 for i = 1:size(fp_pxl,1)
214     if fid > 0
215         value1 = bin(fp_pxl(i,1));
216         value2 = bin(fp_pxl(i,2));
217         fprintf(fid,'0b%s%s,\n',value1, value2);
218     end
219 end
220 fclose(fid);
221
222 %creation of input files (fixed point Xq Yq) header
223 fname = sprintf('./input_xq_yq.h');
224 fid = fopen(fname,'w');
225 for i = 1:size(fp_coords,1)
226     if fid > 0
227         value1 = bin(fp_coords(i,1));
228         value2 = bin(fp_coords(i,2));
229         fprintf(fid,'0b%s%s,\n',value1, value2);
230     end
231 end
232 fclose(fid);
233
234 % golden reference
235 golden = reshape(resized, 2, [])';
236 fname = sprintf('./golden.h');
237 fid = fopen(fname,'w');
238 for i = 1:size(golden,1)
239     if fid > 0
240         value1 = typecast(golden(i,1), 'int32');
241         value2 = typecast(golden(i,2), 'int32');
242         fprintf(fid,'%d, %d,\n',value1, value2);
243     end
244 end
245 fclose(fid);
246
247 %% display result
248 subplot(1,2,1);
```

```
249 imshow(double(I)/255)
250 title('Camera␣Image')
251 subplot(1,2,2);
252 imshow(ImgOut/255)
253 title('Bilinear␣Interpolation␣(Single␣Precision)');
```

Listing 9.8. buffer dimensions file

9.3.2 Output coordinate grid generation matlab script

```
% Author: Francesca Franzese
% Bilinear interpolation for image resizing
% with interpolation coordinate grid

%% get output grid
sclx = 2;  % scale
scly = 2;
resx = 1024;
resy = 1024;
xshft = (resx-1)/2;
yshft = (resy-1)/2;

output_dim = [floor(resx/sclx), floor(resy/scly)];

[Xo, Yo] = meshgrid((0:output_dim(1)-1)' -
            (output_dim(1)-1)/2, (0:output_dim(2)-1)'
            - (output_dim(2)-1)/2);
Xq = zeros(output_dim(2),output_dim(1));
Yq = zeros(output_dim(2),output_dim(1));
coords = [Xo(:) Yo(:)]';
Xq(:,:,1) = Xo + xshft;
Yq(:,:,1) = Yo + yshft;

vec_xfrm = coords ;

% scale
Mscl = [sclx 0; 0 scly];
vec_xfrm = Mscl * vec_xfrm;

% add to output array
Xtf = reshape(vec_xfrm(1,:)',output_dim(2),output_dim(1));
Ytf = reshape(vec_xfrm(2,:)',output_dim(2),output_dim(1));
Xq(:,:) = Xtf + xshft;
Yq(:,:) = Ytf + yshft;

Xq_vec = Xq(:);
Yq_vec = Yq(:);
```

Listing 9.9. buffer dimensions file

9.4 Appendix D

9.4.1 Host application for design 1

```
//
// Author: Francesca Franzese
//

#include "../aie/src/config.h"

#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <adf.h>
#include <input_pixel.h>
#include <input_xq_yq.h>
#include <golden.h>

#include <unistd.h>

#include <experimental/xrt_xclbin.h>
#include <experimental/xrt_kernel.h>
#include <experimental/xrt_device.h>
#include <experimental/xrt_bo.h>
#include <experimental/xrt_ip.h>
#include <experimental/xrt_graph.h>
#include <experimental/xrt_aie.h>
#include <experimental/xrt_graph.h>
#include <experimental/xrt_ip.h>

// RTL IP registers
#define CTRL_ADDR_REG 0x10
#define BASE_ADDR1 0x14
#define BASE_ADDR2 0x1C
#define BASE_ADDR3 0x24
#define BASE_ADDR4 0x2C
#define BASE_ADDR5 0x34

#define IP_START 0x01
#define IP_NONE 0x00 /
#define IP_END 0x04 // at the end: start=0, done = 0, idle = 1

#define APP_VERSION 24

// ------------------------------------------------------------
// DDR Parameters
```

```
43 // ----------------------------------------------------------
44
45 static constexpr unsigned IMG_INPUT_RESOLUTION = 1024 * 1024;
46 static constexpr unsigned IMG_OUTPUT_RESOLUTION = 512 * 512;
47
48 static constexpr unsigned DDR1_BUFFSIZE_I_BYTES = IMG_INPUT_RESOLUTION
49                                                  * sizeof(int32);
50 static constexpr unsigned DDR2_BUFFSIZE_I_BYTES = IMG_OUTPUT_RESOLUTION
51                                                  * sizeof(int64);
52 static constexpr unsigned PLIO_1_BUFFSIZE_I_BYTES = IMG_OUTPUT_RESOLUTION
53                                                  * sizeof(int64);
54 static constexpr unsigned PLIO_2_BUFFSIZE_I_BYTES = IMG_OUTPUT_RESOLUTION
55                                                  * sizeof(int64);
56 static constexpr unsigned PLIO_3_BUFFSIZE_I_BYTES = IMG_OUTPUT_RESOLUTION
57                                                  * sizeof(int64);
58 static constexpr unsigned DDR_BUFFSIZE_O_BYTES = IMG_OUTPUT_RESOLUTION
59                                                  * sizeof(int32);
60
61 using namespace adf;
62 using namespace std;
63
64 int main(int argc, char* argv[]) {
65
66     std::cout << "INFO: App version " << APP_VERSION << std::endl;
67
68     if (argc != 2) {
69         std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
70         return 1;
71     }
72
73     ////////////////////////////////////////
74     // Open device and load xclbin
75     ////////////////////////////////////////
76
77     char* xclbinFilename = argv[1];
78     unsigned dev_index = 0;
79     auto my_device = xrt::device(dev_index);
80     if(!my_device){
81         std::cout << "Device open error!" << std::endl;
82     }else{
83         std::cout << "Device open OK!" << std::endl;
84     }
85
86     auto xclbin_uuid = my_device.load_xclbin(xclbinFilename);
87
88     ////////////////////////////////////////
```

```
89      // allocating RTL IP kernel
90      ////////////////////////////////////////
91
92      auto ip1 = xrt::ip(my_device, xclbin_uuid,
93      "axi_pixel_reorder:{axi_pixel_reorder_1}");
94      std::cout << "INFO: RTL IP kernel has been allocated! " << std::endl;
95
96      ////////////////////////////////////////
97      // allocating input and output memory
98      ////////////////////////////////////////
99
100     // Pixel buffer
101     auto in_bo1 = xrt::bo(my_device, DDR1_BUFFSIZE_I_BYTES, 0);
102     auto in_bo1_mapped = in_bo1.map<uint64_t*>();
103     memcpy(in_bo1_mapped, img_pxl, DDR1_BUFFSIZE_I_BYTES);
104     in_bo1.sync(XCL_BO_SYNC_BO_TO_DEVICE);
105     std::cout << "INFO: Input memory virtual address 1 : "
106                           << in_bo1_mapped << std::endl;
107
108     // Xq-Yq buffer
109     auto in_bo2 = xrt::bo(my_device, DDR1_BUFFSIZE_I_BYTES, 0);
110     auto in_bo2_mapped = in_bo2.map<uint64_t*>();
111     memcpy(in_bo2_mapped, coords, DDR1_BUFFSIZE_I_BYTES);
112     in_bo2.sync(XCL_BO_SYNC_BO_TO_DEVICE);
113     std::cout << "INFO: Input memory virtual address 2: "
114                           << in_bo2_mapped << std::endl;
115
116     // PLIO 0 buffer
117     auto in_bo3 = xrt::bo(my_device, PLIO_1_BUFFSIZE_I_BYTES, 0);
118     auto in_bo3_mapped = in_bo3.map<uint32_t*>();
119     memset(in_bo3_mapped, 0x0000, PLIO_1_BUFFSIZE_I_BYTES);
120     in_bo3.sync(XCL_BO_SYNC_BO_TO_DEVICE);
121     std::cout << "INFO: Input memory virtual address 3 : "
122                         << in_bo3_mapped << std::endl;
123
124     // PLIO 1 buffer
125     auto in_bo4 = xrt::bo(my_device, PLIO_2_BUFFSIZE_I_BYTES, 0);
126     auto in_bo4_mapped = in_bo4.map<uint32_t*>();
127     memset(in_bo4_mapped, 0x0000, PLIO_2_BUFFSIZE_I_BYTES);
128     in_bo4.sync(XCL_BO_SYNC_BO_TO_DEVICE);
129     std::cout << "INFO: Input memory virtual address 4: "
130                           << in_bo4_mapped << std::endl;
131
132     // PLIO 2 buffer
133     auto in_bo5 = xrt::bo(my_device, PLIO_3_BUFFSIZE_I_BYTES, 0);
134     auto in_bo5_mapped = in_bo5.map<uint32_t*>();
```

```
135     memset(in_bo5_mapped, 0x0000, PLIO_3_BUFFSIZE_I_BYTES);
136     in_bo5.sync(XCL_BO_SYNC_BO_TO_DEVICE);
137     std::cout << "INFO: Input memory virtual address 2: "
138                           << in_bo5_mapped << std::endl;
139 
140     // Output buffer
141     auto out_bo = xrt::bo(my_device, DDR_BUFFSIZE_O_BYTES, 0);
142     auto out_bo_mapped = out_bo.map<uint32_t*>();
143     memset(out_bo_mapped, 0x0000, DDR_BUFFSIZE_O_BYTES);
144     std::cout << "INFO: Output memory virtual address buffer: "
145                           << out_bo_mapped << std::endl;
146 
147     std::cout << "INFO: Memory allocated for input and output buffer" << std::endl;
148 
149     ////////////////////////////////////////
150     // setting up RTL IP registers
151     ////////////////////////////////////////
152 
153     uint64_t addr1 = in_bo1.address();
154     uint64_t addr2 = in_bo2.address();
155     uint64_t addr3 = in_bo3.address(); //PLIO 0 --> xfrac yfrac
156     uint64_t addr4 = in_bo4.address(); //PLIO 1 --> P11 P12
157     uint64_t addr5 = in_bo5.address(); //PLIO 2 --> P21 P22
158 
159     std::cout << "INFO: Setting IP data: " << std::endl;
160     std::cout << "BASE_ADDR_1 : " << addr1 << std::endl;
161     std::cout << "BASE_ADDR_2 : " << addr2 << std::endl;
162     std::cout << "BASE_ADDR_3 : " << addr3 << std::endl;
163     std::cout << "BASE_ADDR_4 : " << addr4 << std::endl;
164     std::cout << "BASE_ADDR_5 : " << addr5 << std::endl;
165 
166     std::cout << "INFO: setting register: " << BASE_ADDR1 << std::endl;
167     ip1.write_register(BASE_ADDR1, addr1 >> 32);
168     ip1.write_register(BASE_ADDR1 + 4, addr1);
169     std::cout << "INFO: setting register: " << BASE_ADDR2 << std::endl;
170     ip1.write_register(BASE_ADDR2, addr2 >> 32);
171     ip1.write_register(BASE_ADDR2 + 4, addr2 );
172 
173     std::cout << "INFO: setting register: " << BASE_ADDR3 << std::endl;
174     ip1.write_register(BASE_ADDR3, addr3 >> 32);
175     ip1.write_register(BASE_ADDR3 + 4, addr3);
176     std::cout << "INFO: setting register: " << BASE_ADDR4 << std::endl;
177     ip1.write_register(BASE_ADDR4, addr4 >> 32);
178     ip1.write_register(BASE_ADDR4 + 4, addr4 );
179     std::cout << "INFO: setting register: " << BASE_ADDR5 << std::endl;
180     ip1.write_register(BASE_ADDR5, addr5 >> 32);
```

```
181     ip1.write_register(BASE_ADDR5 + 4, addr5);
182 
183     std::cout << "INFO: base address setup complete! " << std::endl;
184 
185     ////////////////////////////////////////
186     // allocating input-output HLS kernels
187     ////////////////////////////////////////
188 
189     auto mm2s_k1 = xrt::kernel(my_device, xclbin_uuid, "mm2s:{mm2s_1}");
190     auto mm2s_k2 = xrt::kernel(my_device, xclbin_uuid, "mm2s:{mm2s_2}");
191     auto mm2s_k3 = xrt::kernel(my_device, xclbin_uuid, "mm2s:{mm2s_3}");
192     auto s2mm_k = xrt::kernel(my_device, xclbin_uuid, "s2mm:{s2mm_1}");
193     std::cout << "INFO: HLS Kernel allocation completed" << std::endl;
194 
195     ////////////////////////////////////////
196     // Load AIE graph
197     ////////////////////////////////////////
198 
199     std::cout << "INFO: Allocating aie graph..." << std::endl;
200     auto my_graph = xrt::graph(my_device, xclbin_uuid, "blint");
201     std::cout << "INFO: Aie graph allocation completed" << std::endl;
202 
203     ////////////////////////////////////////
204     // start RTL kernel
205     ////////////////////////////////////////
206 
207     uint32_t axi_ctrl = IP_START;
208     ip1.write_register(CTRL_ADDR_REG, axi_ctrl);
209     std::cout << "INFO: RTL Kernel run start" << std::endl;
210 
211     while( axi_ctrl != IP_END ){
212         // gives start command to the IP using s_axi_control
213         axi_ctrl = ip1.read_register(CTRL_ADDR_REG);
214         if(axi_ctrl == IP_START){
215             ip1.write_register(CTRL_ADDR_REG, IP_NONE);
216         }
217     }
218 
219     std::cout << "INFO: RTL processing ended!" << std::endl;
220 
221     ////////////////////////////////////////
222     // forcing cache/memory coherence
223     ////////////////////////////////////////
224 
225     in_bo3.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
226     in_bo4.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
```

```
227     in_bo5.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
228 
229     //////////////////////////////////////////
230     // start input-output kernel
231     //////////////////////////////////////////
232 
233     auto mm2s_r1 =
234      mm2s_k1(in_bo5, nullptr, PLIO_1_BUFFSIZE_I_BYTES/sizeof(int64));
235     std::cout << "INFO: mm2s1 run started" << std::endl;
236 
237     auto mm2s_r2 =
238     mm2s_k2(in_bo3, nullptr, PLIO_2_BUFFSIZE_I_BYTES/sizeof(int64));
239     std::cout << "INFO: mm2s2 run started" << std::endl;
240 
241     auto mm2s_r3 =
242     mm2s_k3(in_bo4, nullptr, PLIO_3_BUFFSIZE_I_BYTES/sizeof(int64));
243     std::cout << "INFO: mm2s3 run started" << std::endl;
244 
245     auto s2mm_r =
246     s2mm_k(out_bo, nullptr, DDR_BUFFSIZE_O_BYTES/sizeof(int64));
247     std::cout << "INFO: s2mm run started" << std::endl;
248 
249 
250     //////////////////////////////////////////
251     // Start AIE graph
252     //////////////////////////////////////////
253 
254     std::cout << "INFO: AIE graph run started" << std::endl;
255     std::cout << "INFO: Running graph for " << NRUN << " iterations\n";
256     my_graph.run(NRUN);
257     std::cout << "INFO: Waiting for completion..." << std::endl;
258     my_graph.end();
259     std::cout << "INFO: AIE graph run completed!" << std::endl;
260 
261 
262     //////////////////////////////////////////
263     // Wait for kernels completion
264     //////////////////////////////////////////
265 
266     std::cout << std::endl << "INFO: Waiting for kernels to end...\n";
267     s2mm_r.wait();
268     out_bo.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
269     std::cout << std::endl << "INFO: Kernel execution completed!\n";
270 
271     //////////////////////////////////////////
272     // Data validation
```

```
273     ////////////////////////////////////////
274
275     std::cout << std::endl
276     << "INFO: Verification of output data with respect to golden reference:"
277     << std::endl;
278
279     float max_diff;
280     float diff;
281     float mismatch_count;
282     float slack = 1;
283
284     for (unsigned ss=0; ss < DDR_BUFFSIZE_0_BYTES/sizeof(int32); ss++) {
285
286         float data_golden;
287         float data_aie;
288         data_aie = (float)out_bo_mapped[ss];
289         data_golden = (float)output_ref[ss];
290         diff = abs(data_golden-data_aie);
291
292         if(diff > slack){
293
294             mismatch_count++;
295         }
296         if(diff > max_diff){
297
298             max_diff = diff;
299         }
300         ss++;
301     }
302
303
304     if(max_diff != 0){
305
306         std::cout << std::endl << "MISMATCH!" << std::endl;
307         std::cout << std::endl << "Maximum error is " << max_diff;
308         std::cout << std::endl << mismatch_count << "/"
309         << IMG_OUTPUT_RESOLUTION << " different samples" << std::endl;
310
311     }else{
312
313         std::cout << std::endl << "MATCH!" << std::endl;
314
315     }
316
317     // output file is printed
318
```

```
319     std::ofstream outputFile("output.txt");
320     for (unsigned ss=0; ss < DDR_BUFFSIZE_0_BYTES/sizeof(int32); ss++) {
321 
322         uint32_t data;
323         uint32_t data1;
324 
325         data = out_bo_mapped[ss];
326         data1 = out_bo_mapped[ss+1];
327         outputFile << data << " " << data1 << endl;
328         ss++;
329     }
330     outputFile.close();
331 
332     return 0;
333 }
```

Listing 9.10. Design 1 host code

9.4.2 Host application for design 2

```
//
// Author: Francesca Franzese
//

#include "../aie/src/config.h"
#include "../aie/src/buffers.h"

#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <adf.h>
#include <input_pixel.h>
#include <input_xq_yq.h>
#include <golden.h>

#include <experimental/xrt_xclbin.h>
#include <experimental/xrt_kernel.h>
#include <experimental/xrt_device.h>
#include <experimental/xrt_bo.h>
#include <experimental/xrt_ip.h>
#include <experimental/xrt_graph.h>
#include <experimental/xrt_aie.h>
#include <experimental/xrt_graph.h>
#include <experimental/xrt_ip.h>

// RTL IP registers
#define CTRL_ADDR_REG 0x10
#define BASE_ADDR1 0x14
#define BASE_ADDR2 0x1C
#define IP_START 0x01
#define IP_DONE 0x02
#define IP_IDLE 0x04
#define IP_CLEAR 0x00
#define PRESCALER_REG 0x2C
#define IP_IDLE 0x04
#define PRESCALER_VALUE 0x9C4 //1 packet of 256 data each 8 us

#define APP_VERSION 22

// ------------------------------------------------------------
// DDR Parameters
// ------------------------------------------------------------

static constexpr unsigned IMG_INPUT_RESOLUTION = 1024 * 1024;
```

```
45 static constexpr unsigned IMG_OUTPUT_RESOLUTION = 512 * 512;
46
47 static constexpr unsigned DDR1_BUFFSIZE_I_BYTES =
48 IMG_INPUT_RESOLUTION * sizeof(int32); // pixel of original image
49 static constexpr unsigned DDR2_BUFFSIZE_I_BYTES =
50 IMG_OUTPUT_RESOLUTION * sizeof(int64); //query coordinates
51 static constexpr unsigned DDR_BUFFSIZE_O_BYTES =
52 IMG_OUTPUT_RESOLUTION * sizeof(int32); //output image buffer
53
54
55 // input buffer size is BUFFER_SIZE_IN declared in buffers.h
56 // output buffer size is BUFFER_SIZE_OUT declared in buffers.h
57
58 #define INPUT_SIZE BUFFER_SIZE_IN * sizeof(int32)
59 #define OUTPUT_SIZE BUFFER_SIZE_OUT * sizeof(int32)
60
61 using namespace adf;
62 using namespace std;
63
64 int main(int argc, char* argv[]) {
65
66
67     std::cout << "INFO: App version " << APP_VERSION << std::endl;
68     //TARGET_DEVICE macro needs to be passed from gcc command line
69     if (argc != 2) {
70         std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
71         return 1;
72     }
73
74     ////////////////////////////////////////
75     // Open device and load xclbin
76     ////////////////////////////////////////
77
78     char* xclbinFilename = argv[1];
79     unsigned dev_index = 0;
80     auto my_device = xrt::device(dev_index);
81     if(!my_device){
82         std::cout << "Device open error!" << std::endl;
83     }else{
84         std::cout << "Device open OK!" << std::endl;
85     }
86
87     auto xclbin_uuid = my_device.load_xclbin(xclbinFilename);
88
89     ////////////////////////////////////////
90     // allocating RTL IP kernel
```

```
91    ////////////////////////////////////////
92
93    auto ip1 = xrt::ip(my_device, xclbin_uuid,
94                  "axi_pixel_reorder:{axi_pixel_reorder_1}");
95    std::cout << "INFO: RTL IP kernel has been allocated! " << std::endl;
96
97    ////////////////////////////////////////
98    // allocating input and output memory
99    ////////////////////////////////////////
100
101   //input buffer for RTL IP
102   auto in_bo1 = xrt::bo(my_device, DDR1_BUFFSIZE_I_BYTES,
103                         xrt::bo::flags::host_only, 0);
104   auto in_bo1_mapped = in_bo1.map<uint64_t*>();
105   memcpy(in_bo1_mapped, img_pxl, DDR1_BUFFSIZE_I_BYTES);
106   in_bo1.sync(XCL_BO_SYNC_BO_TO_DEVICE);
107   std::cout << "INFO: Input memory virtual address 1 : "
108                         << in_bo1_mapped << std::endl;
109
110   auto in_bo2 = xrt::bo(my_device, DDR2_BUFFSIZE_I_BYTES,
111                         xrt::bo::flags::host_only, 0);
112   auto in_bo2_mapped = in_bo2.map<uint64_t*>();
113   memcpy(in_bo2_mapped, coords, DDR2_BUFFSIZE_I_BYTES);
114   in_bo2.sync(XCL_BO_SYNC_BO_TO_DEVICE);
115   std::cout << "INFO: Input memory virtual address 2: "
116                         << in_bo2_mapped << std::endl;
117
118   //ouput buffer for AIE results
119   auto out_bo = xrt::bo(my_device, DDR_BUFFSIZE_O_BYTES, 0);
120   auto out_bo_mapped = out_bo.map<uint32_t*>();
121   memset(out_bo_mapped, 0x0000, DDR_BUFFSIZE_O_BYTES);
122   std::cout << "INFO: Output memory virtual address buffer: "
123                            << out_bo_mapped << std::endl;
124
125   ////////////////////////////////////////
126   // setting up RTL IP registers
127   ////////////////////////////////////////
128
129   uint64_t addr1;
130   uint64_t addr2;
131   addr1 = in_bo1.address();
132   addr2 = in_bo2.address();
133
134   std::cout << "INFO: Setting IP data: " << std::endl;
135   std::cout << "BASE_ADDR_1 : " << addr1 << std::endl;
136   std::cout << "BASE_ADDR_2 : " << addr2 << std::endl;
```

```
137
138     std::cout << "INFO: setting register: " << BASE_ADDR1 << std::endl;
139     ip1.write_register(BASE_ADDR1, addr1 >> 32);
140     ip1.write_register(BASE_ADDR1 + 4, addr1);
141     std::cout << "INFO: setting register: " << BASE_ADDR2 << std::endl;
142     ip1.write_register(BASE_ADDR2, addr2 >> 32);
143     ip1.write_register(BASE_ADDR2 + 4, addr2 );
144
145     std::cout << "INFO: base address setup complete! " << std::endl;
146
147     ip1.write_register(PRESCALER_REG, PRESCALER_VALUE);
148     std::cout << "INFO: RTL Prescaler setup completed" << std::endl;
149
150     ////////////////////////////////////////
151     // start output kernel
152     ////////////////////////////////////////
153
154     auto s2mm_k = xrt::kernel(my_device, xclbin_uuid, "s2mm:{s2mm_1}");
155     auto s2mm_r =
156     s2mm_k(out_bo, nullptr, DDR_BUFFSIZE_0_BYTES/sizeof(int64));
157     std::cout << "s2mm run started" << std::endl;
158
159     std::cout << "INFO: HLS Kernel run start" << std::endl;
160
161     ////////////////////////////////////////
162     // start RTL kernel
163     ////////////////////////////////////////
164
165     // gives start command to the IP
166     uint32_t axi_ctrl = IP_START;
167     ip1.write_register(CTRL_ADDR_REG, axi_ctrl);
168     std::cout << "INFO: RTL Kernel run start" << std::endl;
169     ip1.write_register(CTRL_ADDR_REG, IP_CLEAR);
170
171     ////////////////////////////////////////
172     // Load AIE graph
173     ////////////////////////////////////////
174
175     std::cout << "Allocating aie graph..." << std::endl;
176     auto my_graph = xrt::graph(my_device, xclbin_uuid, "blint");
177     std::cout << "Aie graph allocation completed" << std::endl;
178
179     ////////////////////////////////////////
180     // Run AIE graph
181     ////////////////////////////////////////
182
```

```
183     my_graph.reset();
184     std::cout << "Running graph for " << NRUN << " iterations\n";
185     my_graph.run(NRUN);
186     std::cout << "Waiting for completion..." << std::endl;
187 
188     my_graph.end();
189     std::cout << "Completed!" << std::endl;
190 
191 
192     ////////////////////////////////////////
193     // Wait for kernels completion
194     ////////////////////////////////////////
195 
196     std::cout << std::endl << "Waiting for kernels to end..." << std::endl;
197 
198     s2mm_r.wait();
199     out_bo.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
200     std::cout << std::endl << "INFO: Kernel execution completed!\n";
201 
202     ////////////////////////////////////////
203     // Data validation
204     ////////////////////////////////////////
205 
206     std::cout << "Verification of output data wrt golden reference:\n";
207 
208     float max_diff;
209     float diff;
210     float mismatch_count;
211     float slack = 1;
212 
213     for (unsigned ss=0; ss < OUTPUT_SIZE/sizeof(int32); ss++) {
214 
215         float data_golden;
216         float data_aie;
217 
218         data_aie = (float)out_bo_mapped[ss];
219         data_golden = (float)output_ref[ss];
220 
221         diff = abs(data_golden-data_aie);
222         if(diff > slack){
223 
224             mismatch_count++;
225         }
226 
227         if(diff > max_diff){
228 
```

```
            max_diff = diff;
        }

        ss++;
    }


    if(max_diff != 0){

        std::cout << std::endl << "MISMATCH!" << std::endl;
        std::cout << std::endl << "Maximum error is " << max_diff << std::endl;
        std::cout << std::endl << mismatch_count << "/"
        << IMG_OUTPUT_RESOLUTION << " different samples" << std::endl;

    }else{

        std::cout << std::endl << "MATCH!" << std::endl;

    }


   // output file is printed
    std::ofstream outputFile("output.txt");
    for (unsigned ss=0; ss < DDR_BUFFSIZE_0_BYTES/sizeof(int32); ss++) {

        uint32_t data;
        uint32_t data1;

        data = out_bo_mapped[ss];
        data1 = out_bo_mapped[ss+1];
        outputFile << data << " " << data1 << endl;
        ss++;
    }
    outputFile.close();

    return 0;

}
```

Listing 9.11. Design 2 host code

9.4.3 Host application for design 3

```
//
// Author: Francesca Franzese
//

#include "../aie/src/config.h"
#include "../aie/src/buffers.h"

#include <cstdint>
#include <bitset>
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <adf.h>
#include "deprecated/xrt.h"
#include "pixel_array.h"
#include "xq_array.h"
#include "yq_array.h"
#include "golden.h"

#include <experimental/xrt_xclbin.h>
#include <experimental/xrt_kernel.h>
#include <experimental/xrt_device.h>
#include <experimental/xrt_bo.h>
#include <experimental/xrt_ip.h>
#include <experimental/xrt_graph.h>
#include <experimental/xrt_aie.h>

#define APP_VERSION 10
#define NRUN 512
#define SCALE_FACTOR 2
#define YRES 1024

// ------------------------------------------------------------
// DDR Parameters
// ------------------------------------------------------------

static constexpr unsigned IMG_INPUT_RESOLUTION = 1024 * 1024;
static constexpr unsigned IMG_OUTPUT_RESOLUTION = 512 * 512;

static constexpr unsigned BUFF_SIZE_IMG = IMG_INPUT_RESOLUTION
    * sizeof(int32); // pixel of original image
static constexpr unsigned BUFF_SIZE_XY = IMG_OUTPUT_RESOLUTION
    * sizeof(int32); //query coordinates (XqYq data on 64 bit)
static constexpr unsigned BUFF_SIZE_OUT = IMG_OUTPUT_RESOLUTION
```

```
45      * sizeof(int32); //output image buffer
46
47  #define INPUT_SIZE BUFFER_SIZE_IN * sizeof(int32)
48  #define OUTPUT_SIZE BUFFER_SIZE_OUT * sizeof(int32)
49
50  using namespace adf;
51  using namespace std;
52
53  int main(int argc, char* argv[]) {
54
55      std::cout << "INFO: App version " << APP_VERSION << std::endl;
56      //TARGET_DEVICE macro needs to be passed from gcc command line
57      if (argc != 2) {
58          std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
59          return 1;
60      }
61
62      ////////////////////////////////////////
63      // Open device and load xclbin
64      ////////////////////////////////////////
65
66      char* xclbinFilename = argv[1];
67      unsigned dev_index = 0;
68      auto my_device = xrt::device(dev_index);
69      if(!my_device){
70          std::cout << "Device open error!" << std::endl;
71      }else{
72          std::cout << "Device open OK!" << std::endl;
73      }
74
75      auto xclbin_uuid = my_device.load_xclbin(xclbinFilename);
76
77      ////////////////////////////////////////
78      // allocating input and output memory
79      ////////////////////////////////////////
80
81      auto din_xq_buffer =
82          xrt::bo (my_device, BUFF_SIZE_XY,xrt::bo::flags::normal, 0);
83      uint32_t* dinXqArray= din_xq_buffer.map<uint32_t*>();
84      memcpy(dinXqArray, xq_array, BUFF_SIZE_XY);
85      std::cout << "INFO: Xq array allocated. "<< std::endl;
86
87      auto din_yq_buffer =
88          xrt::bo (my_device, BUFF_SIZE_XY,xrt::bo::flags::normal, 0);
89      uint32_t* dinYqArray= din_yq_buffer.map<uint32_t*>();
90      memcpy(dinYqArray, yq_array, BUFF_SIZE_XY);
```

```
91      std::cout << "INFO: Yq array allocated. "<< std::endl;
92
93      auto din_img_buffer =
94          xrt::bo (my_device, BUFF_SIZE_IMG,xrt::bo::flags::normal, 0);
95      uint32_t* dinImgArray= din_img_buffer.map<uint32_t*>();
96      memcpy(dinImgArray, pxl_array, BUFF_SIZE_IMG);
97      std::cout << "INFO: Image array allocated." << std::endl;
98
99      auto dout_buffer =
100         xrt::bo (my_device, BUFF_SIZE_OUT,xrt::bo::flags::normal, 0);
101     uint32_t* doutArray= dout_buffer.map<uint32_t*>();
102     std::cout << "INFO: Output buffer allocated." << std::endl;
103
104     ////////////////////////////////////////
105     // Load AIE graph
106     ////////////////////////////////////////
107
108     std::cout << "Allocating aie graph..." << std::endl;
109     auto my_graph = xrt::graph(my_device, xclbin_uuid, "blint");
110     std::cout << "Aie graph allocation completed" << std::endl;
111
112     my_graph.reset();
113
114     ////////////////////////////////////
115     // creation of external buffer
116     ////////////////////////////////////
117
118     auto xq_buffer_ext =
119         xrt::aie::bo (my_device, BUFF_SIZE_XY,xrt::bo::flags::normal, 0);
120     auto yq_buffer_ext =
121         xrt::aie::bo (my_device, BUFF_SIZE_XY,xrt::bo::flags::normal, 0);
122     auto img_buffer_ext =
123         xrt::aie::bo (my_device, BUFF_SIZE_IMG,xrt::bo::flags::normal, 0);
124
125     auto out_buffer =
126         xrt::aie::bo (my_device, BUFF_SIZE_OUT,xrt::bo::flags::normal, 0);
127
128     xq_buffer_ext.write(dinXqArray);
129     yq_buffer_ext.write(dinYqArray);
130     img_buffer_ext.write(dinImgArray);
131
132     int img_offset = 0;
133     int xq_offset = 0;
134     int yq_offset = 0;
135
136     for(int i= 0; i < NRUN; i ++){
```

```
137
138         ////////////////////////////////////
139         // GMIO run
140         ////////////////////////////////////
141
142         img_buffer_ext.async("blint.gmioIn1", XCL_BO_SYNC_BO_GMIO_TO_AIE,
143             2048 * sizeof(uint32_t), img_offset * sizeof(uint32_t));
144         xq_buffer_ext.async("blint.gmioIn2", XCL_BO_SYNC_BO_GMIO_TO_AIE,
145             512 * sizeof(uint32_t), xq_offset * sizeof(uint32_t));
146         yq_buffer_ext.async("blint.gmioIn3", XCL_BO_SYNC_BO_GMIO_TO_AIE,
147             512 * sizeof(uint32_t), yq_offset * sizeof(uint32_t));
148
149         ////////////////////////////////////////
150         // Run AIE graph
151         ////////////////////////////////////////
152
153         std::cout << "Running graph for " << NRUN << " iterations\n";
154         my_graph.run(1);
155         std::cout << "Waiting for completion..." << std::endl;
156         my_graph.wait(0);
157
158         auto dout_buffer_run = out_buffer.async("blint.gmioOut",
159             XCL_BO_SYNC_BO_AIE_TO_GMIO, 512 * sizeof(uint32_t),
160             xq_offset * sizeof(uint32_t));
161
162         dout_buffer_run.wait();
163         std::cout << "INFO: out buffer run completed!" << std::endl;
164
165         unsigned array_lb = i*512;
166         dout_buffer.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
167         out_buffer.read(&doutArray[array_lb], 512 * sizeof(uint32_t),
168             xq_offset * sizeof(uint32_t));
169
170         xq_offset = xq_offset + 512;
171         yq_offset = yq_offset + 512;
172         img_offset = img_offset + YRES*SCALE_FACTOR;
173     }
174
175     my_graph.end();
176
177     ////////////////////////////////////////
178     // Data validation
179     ////////////////////////////////////////
180
181     std::cout << "Verification of output data wrt to golden reference:\n";
182
```

```
183     float max_diff;
184     float diff;
185     float mismatch_count = 0;
186     float slack = 1;
187 
188     for (unsigned ss=0; ss < BUFF_SIZE_OUT/sizeof(int32); ss++) {
189 
190         float data_golden;
191         float data_aie;
192 
193         data_aie = (float)doutArray[ss];
194         data_golden = (float)output_ref[ss];
195 
196         diff = abs(data_golden-data_aie);
197         if(diff > slack){
198 
199             mismatch_count++;
200         }
201 
202         if(diff > max_diff){
203 
204             max_diff = diff;
205         }
206 
207         ss++;
208     }
209 
210     if(mismatch_count != 0){
211 
212         std::cout << std::endl << "MISMATCH!" << std::endl;
213         std::cout << std::endl << "Max error is " << max_diff << std::endl;
214         std::cout << std::endl << mismatch_count << "/"
215         << IMG_OUTPUT_RESOLUTION << " different samples" << std::endl;
216 
217     }else{
218 
219         std::cout << std::endl << "MATCH!" << std::endl;
220 
221     }
222 
223 
224    // output file is printed
225     std::cout << std::endl << "INFO: Printing output file..." << std::endl;
226 
227     std::ofstream outputFile("output.txt");
228     for (unsigned ss=0; ss < BUFF_SIZE_OUT/sizeof(int32); ss++) {
```

```
229
230         uint32_t data;
231         data = doutArray[ss];
232
233         outputFile << data << endl;
234     }
235     outputFile.close();
236
237     std::cout << std::endl << "INFO: Output file printed!" << std::endl;
238     return 0;
239
240 }
```

Listing 9.12. Design 3 host code

Acknowledgements

My sincere thanks go to my colleagues of the Ivrea Tria team for assisting me in the development of this thesis and for guiding me through this first work experience. This past year has been incredibly stimulating, and you have made me feel part of the team from day one.

I would especially like to thank the co-supervisor of this thesis, M. Cignetti, for giving me this opportunity. I also thank Prof. M.R. Casu for supporting me as supervisor.

All my gratitude goes to my parents, Fernando and Maria, and my sister, Antonella, to whom I dedicate this thesis and my deepest affection. Without you, I would not be the person I am today.

Bibliography

[1] *VE2302 SOM*, based on the Versal AI Edge VE2302 © Copyright 2024 Avnet, Inc. All rights reserved. https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2

[2] *AMD Versal™ AI Edge Series ©*, 2024 Advanced Micro Devices, Inc.https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal/ai-edge-series.html

[3] *AMD Versal™ AI Edge Series ©*, 2024 Advanced Micro Devices, Inc. All rights reserved.

[4] *AMD Versal™ AI Edge Series ©*, 2024 Advanced Micro Devices, Inc.https://docs.amd.com/r/en-US/ug1611-vhk158-eval-bd/GTYP-Transceivers

[5] *AMD Versal™ AI Edge Series VEK280 Evaluation Kit ©*, 2024 Advanced Micro Devices, Inc.https://www.xilinx.com/products/boards-and-kits/vek280.html#information

[6] *Bilinear Interpolation, Swienegel - Own work*, CC BY-SA 4.0,https://commons.wikimedia.org/w/index.php?curid=124957943

[7] *AMD - AIE Engine Development: Bilinear Interpolation*, 2024 Advanced Micro Devices, Inc.,https://github.com/Xilinx/Vitis-Tutorials/tree/2024.2/AI_Engine_Development/AIE/Design_Tutorials/11-Bilinear_Interpolation

[8] *P. He et al., "Super-Resolution of Digital Elevation Model with Local Implicit Function Representation"*, 2022 International Conference on Machine Learning and Intelligent Systems Engineering (MLISE), Guangzhou, China, 2022, pp. 111-116, doi: 10.1109/MLISE57402.2022.00030.

[9] *AI Engine-ML Kernel and Graph Programming Guide (UG1603) ©*, 2024 Advanced Micro Devices, Inc.https://docs.amd.com/r/en-US/ug1603-ai-engine-ml-kernel-graph

[10] *AI Engine Versal Integration*, copyright © 2020–2024

Advanced Micro Devices, Inc.https://github.com/Xilinx/Vitis-Tutorials/tree/2024.2/AI_Engine_Development/AIE/Feature_Tutorials/05-AI-engine-versal-integration#ai-engine-versal-integration

[11] *XRT Native APIs*, © Copyright 2017-2023, Advanced Micro Devices, Inc. https://xilinx.github.io/XRT/master/html/xrt_native_apis.html#

[12] *Vitis AI*, © Copyright 2017-2023, Advanced Micro Devices, Inc. https://github.com/Xilinx/Vitis-AI

[13] *AMD Vitis™ In-Depth Tutorials*, © Copyright 2017-2023, Advanced Micro Devices, Inc. https://github.com/Xilinx/Vitis-Tutorials

[14] *AI Engine API User Guide*, UG1529 © 2024 Advanced Micro Devices, Inc. https://www.xilinx.com/htmldocs/xilinx2024_2/aiengine_api/aie_api/doc/index.html