

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI FISICA E ASTRONOMIA “GALILEO GALILEI”

MASTER’S DEGREE IN PHYSICS OF DATA

FINAL DISSERTATION

Accelerating Online Selection Algorithms for Level-1
Trigger Scouting in the CMS Phase-2 Upgrade

Thesis supervisor

Dr. Giovanni Petrucciani

Thesis co-supervisor

Prof. Andrea Triossi

Candidate

Giovanni Zago

Academic Year 2024/2025

Abstract

The Level-1 (L1) Trigger Scouting is a novel real-time data collection system being introduced as part of the L1 trigger of the CMS experiment at the CERN Large Hadron Collider (LHC). While an initial demonstrator implementation already exists, the full-scale scouting system will become a key component of the L1 trigger only with the major Phase-2 upgrade planned for the High-Luminosity LHC (HL-LHC) era. The main motivation for implementing such a system is to search for physics signatures that would evade the traditional trigger and analysis chain. The extreme pile-up conditions foreseen at the HL-LHC are accompanied by a significant upgrade of the entire CMS detector, its readout electronics, and the L1 trigger itself. In particular, the latter will be capable of running sophisticated algorithms that were previously impossible at L1, thus providing reconstructed L1 objects with a resolution comparable to offline processing. By leveraging the enhanced quality of L1 objects and the vast statistics provided by the 40 MHz bunch crossing rate, the L1 Trigger Scouting system enables real-time physics analyses that would be technically and economically unfeasible if relying on full detector data. Beyond a data acquisition infrastructure, the scouting system requires a computing farm to execute its dedicated algorithms. This allows for the exploration of a wide range of compute units and platforms, including CPUs, GPUs, and recently commercialized heterogeneous devices like AMD Versal SoCs, which integrate a Processing System (PS), Programmable Logic (PL), and Adaptable Intelligent (AI) engines on the same chip. This work focuses on exploring how Versal devices can be integrated into the scouting system to offload algorithms originally designed for CPUs and/or FPGAs. In particular, a cut-based search for the rare W boson decay into three charged pions is used as a test case and ported to AI engines. Moreover, a hardware implementation of the final design is tested on an AMD VCK5000 Versal Development Card powered by a VC1902 chip.

Acknowledgments

This thesis concludes an important chapter in my academic journey, and I am deeply grateful to the many people who supported, guided, and accompanied me along the way. First and foremost, I wish to express my sincere gratitude to Prof. Andrea Triossi for offering me a unique and challenging thesis proposal, and for granting me the invaluable opportunity to join CERN, where I gained valuable experience in one of the world's most prestigious research environments. I am especially thankful to Dr. Giovanni Petrucciani, who closely supervised my work during my time at CERN. His insightful guidance, practical suggestions, and enriching discussions were fundamental to the development of this project. I would also like to extend my thanks to the entire CMG-OS group, whose openness made it possible for me to learn and grow in a truly inspiring setting. I am also deeply grateful to Prof. Marco Zanetti for his support and mentorship throughout this journey. His advice and encouragement, both on a professional and personal level, were instrumental in helping me feel welcomed and integrated into the BoostLab group, to which I also express my appreciation.

To my CERN colleagues Matteo, Gabriele, Sabrina, Rocco, and Nicolò L., thank you for generously sharing your knowledge and for offering vital support during the development of this work. The light-hearted atmosphere you created in the office made the day-to-day struggles not only manageable, but often enjoyable. I will never forget the masterpieces (the so-called “towers”) you built on my desk in my absence. Such works of art had me convinced for a while you were pursuing dual PhDs in physics and architecture. A special thanks also goes to Emanuele, Leonardo, Andrea, Nicolò S., Enrico, Jacopo, and Antonino, who made every lunch break a moment of *true discussions*, and a source of both small and great adventures. I am also grateful to Davide, Valentina, Piero, Kyungmin and Haya, a.k.a. the Saint-Genis cooking lovers, for the *delicious* moments shared together. More broadly, I wish to thank all those I met during my time at CERN, as every interaction, every chat, and every moment contributed to shaping me both as a student and as a person.

To my family, thank you for always standing by me, through both smooth and difficult times. Your unwavering support has meant the world to me. To my parents, who have always been my biggest fans: this achievement is as much yours as it is mine. I truly hope it is just another stepping stone toward new beginnings. Finally, my deepest thanks to my dear Silvia (Sissi), who so patiently endured the unpredictability of a curious and often wandering *special* person.

Contents

Abstract	2
Acknowledgments	3
Contents	5
1 Introduction	6
2 The CMS Experiment and its Upgrade for the High-Luminosity LHC Era	8
2.1 The Large Hadron Collider at CERN	8
2.1.1 The CERN Accelerator Complex and Experiments	8
2.1.2 Collisions at LHC	9
2.2 The Compact Muon Solenoid (CMS) Experiment	11
2.2.1 Sub-detectors	12
2.2.2 Trigger and Data Acquisition Systems	14
2.2.3 Event Reconstruction	15
2.3 CMS in the HL-LHC Era	16
2.3.1 Sub-detectors Upgrades	17
2.3.2 A New Level-1 Trigger System	17
2.4 The Phase-2 Level-1 Trigger Scouting System	18
2.4.1 The Physics' Case for L1T Scouting	19
2.4.2 Baseline Architecture	20
2.4.3 Phase-2 L1T Scouting Demonstrator	22
3 AMD Versal Adaptive SoCs	24
3.1 Adaptable Intelligent Engines Architecture	25
3.1.1 AI Engine Tile Overview	25
3.1.2 AI Engine Processor	26
3.1.3 AI Engine Interfaces	28
3.1.4 AI Engine Memory Module	29
3.1.5 AI Engine Parallelism	30

3.2	AI Engine Coding and Programming Model	30
3.2.1	Scalar and Vector Data Types	31
3.2.2	Vector Operations	32
3.2.3	Programming Model	33
3.3	Vitis Tools	36
3.3.1	AMD Data Center Acceleration	37
3.4	Deploying Accelerated Applications on Hardware	38
3.4.1	AMD VCK5000 Versal Development Card	39
3.4.2	Installing the Card and the Deployment Software	39
3.4.3	Programming the Host and Running the Application	42
4	The Rare $W \rightarrow 3\pi$ Decay Online Selection	
	on AI Engines	44
4.1	The Cut-Based $W \rightarrow 3\pi$ Selection Algorithm	44
4.2	Signal and Background Dataset	45
4.2.1	PUPPI Data Format	46
4.2.2	Gen-matching and Target Triplets	47
4.2.3	Dataset Format and Reprocessing	48
4.3	Hardware Implementation	49
4.3.1	Design Overview	49
4.3.2	PL Kernels	51
4.3.3	AI Engine Kernel	54
4.3.4	Host Program	60
4.3.5	Compiling, Linking and Packaging the Design	62
4.3.6	Physics Performance	66
4.3.7	Computational Performance	66
5	Conclusions	69
	Bibliography	76
	Acronyms	77

Chapter 1

Introduction

In the grand endeavor to understand the universe at its most fundamental level, the Standard Model (SM) of particle physics has stood as a remarkably successful theoretical framework. It encapsulates our most precise understanding of the behavior of elementary particles and the fundamental forces, with predictions verified by a wealth of experimental results. A cornerstone achievement was the discovery of the Higgs boson at the Large Hadron Collider (LHC) in 2012, a peaking moment that not only affirmed the last missing piece of the SM but also opened a new frontier for the search for physics beyond the Standard Model (BSM). Yet, despite its accuracy, the SM is incomplete. It leaves unanswered questions about neutrino masses, the origin of the electroweak scale, the matter-antimatter asymmetry, and the nature of dark matter and dark energy. These limitations motivate the ongoing quest for BSM physics, a journey which continues to shape experimental strategies at the forefront of high-energy physics.

The LHC, the world's most powerful particle collider, produces proton-proton collisions at 40 MHz frequency, generating massive volumes of data. Due to practical constraints in data storage and processing, only a tiny fraction of events can be retained for offline analysis. This stringent filtering is performed by a hierarchical trigger system, which makes real-time decisions based on predefined selection criteria. However, this necessary reduction comes at a cost: potentially interesting events, especially those arising from rare or unexpected BSM processes, might be lost forever. To address this challenge, the CMS collaboration has developed the Level-1 Trigger (L1T) scouting system, a novel paradigm that enables the collection and processing of trigger-level event information at the full 40 MHz rate. This system—expected to start operating at the beginning of the HL-LHC era—increases the rate at which potentially interesting events can be investigated, without overwhelming the data acquisition pipeline. The scouting infrastructure will operate by parasitically intercepting trigger-level data directly from the L1T processing boards. This data, consisting of reconstructed physics objects, is transmitted via high-speed links to custom Data Acquisition (DAQ) boards, equipped with FPGA-based logic for zero-suppression and pre-processing. The processed data is then routed through a network switch to a processing farm, where further filtering, classification, and physics reconstruction tasks are executed asynchronously from the trigger decision logic. In

addition to simulated physics performance studies, ongoing research and development aims to enhance the computational capabilities of the processing farm through the use of hardware accelerators such as FPGAs, GPUs, and heterogeneous platforms like the AMD Versal Adaptive SoCs. The latter integrate conventional processors, reconfigurable FPGA logic, and recently developed AI Engines, which are composed of a 2D array of hard-silicon programmable cores. Each core features a scalar unit, a vector unit, dedicated on-chip memory, and a rich set of interconnects. This combination enables the efficient development of accelerated applications that leverage all available compute resources and exploit multiple levels of parallelism. This research is conducted within the framework of the Next Generation Triggers (NGT) project, specifically under Work Package 3.5, which aims to provide additional resources and support for the commissioning of the Phase-2 L1T scouting system. Through the milestones defined by the NGT project, CERN seeks to advance the computing technologies used in the CMS and ATLAS experiments, with the ultimate goal of optimizing and enhancing the discovery potential for ultra-rare physics signatures.

In this work, we present a complete implementation of the rare $W \rightarrow 3\pi$ decay online selection algorithm on Versal AI engines, providing insights into the performance and benefits of leveraging heterogeneous accelerator architectures. The implementation is deployed on the AMD VCK5000 Development Card, equipped with a VC1902 Versal chip, which is well-suited for the current scouting baseline architecture as it interfaces with a host server via PCIe.

The structure of this thesis is as follows: Chapter 2 introduces the CERN LHC and the CMS experiment, with a particular focus on the L1T redesign planned for the HL-LHC era and the Phase-2 L1T scouting system. Chapter 3 details the key features of the Versal Adaptive SoC architecture and its associated programming model. It also outlines the development workflow for deploying applications on AI engines using the AMD VCK5000 platform. In Chapter 4, we provide a thorough description of the online selection algorithm implemented on the Versal chip, discussing both the physics and computational performance obtained from Monte Carlo simulated data reflecting information available at L1. Finally, Chapter 5 concludes the thesis with final remarks and future perspectives.

Chapter 2

The CMS Experiment and its Upgrade for the High-Luminosity LHC Era

The Large Hadron Collider (LHC) and the Compact Muon Solenoid (CMS) experiment are central to this thesis. This chapter provides an overview of their key aspects, including design, operational principles, and planned upgrades. The concepts introduced here serve as essential background information, providing context for the work discussed in later chapters.

The first sections focus on the current status of the LHC and CMS, followed by an introduction to the High-Luminosity LHC (HL-LHC) upgrades. Particular emphasis is placed on the Level-1 Trigger system of the CMS experiment, its Phase-2 configuration, and the Phase-2 Level-1 Trigger scouting system, which plays a crucial role in this research.

2.1 The Large Hadron Collider at CERN

The Large Hadron Collider (LHC)[1], located at the European Organization for Nuclear Research (CERN) near Geneva, Switzerland, stands as the most powerful particle accelerator ever built. This 27-kilometer-long circular collider, originally constructed for the Large Electron-Positron (LEP) collider, is positioned between 45 and 170 meters underground. The LHC is engineered to accelerate both protons and heavy ions, currently reaching a center-of-mass energy of 13.6 TeV in proton-proton collisions, solidifying its role as the most advanced machine of its kind. The development of the LHC spanned more than two decades, from its proposal in 1984 to the beginning of operations in 2008. The machine is now in its third operational run (Run 3), started in 2022, with a HL-LHC upgrade planned to extend its scientific capabilities into the early 2040s.

2.1.1 The CERN Accelerator Complex and Experiments

The LHC serves as the final and most powerful stage in a multi-step accelerator chain designed to bring protons to extreme energies. The process begins with proton extraction from hydrogen gas, followed by pre-acceleration through a series of facilities, LINAC 4 (160 MeV),

The CERN accelerator complex Complexe des accélérateurs du CERN

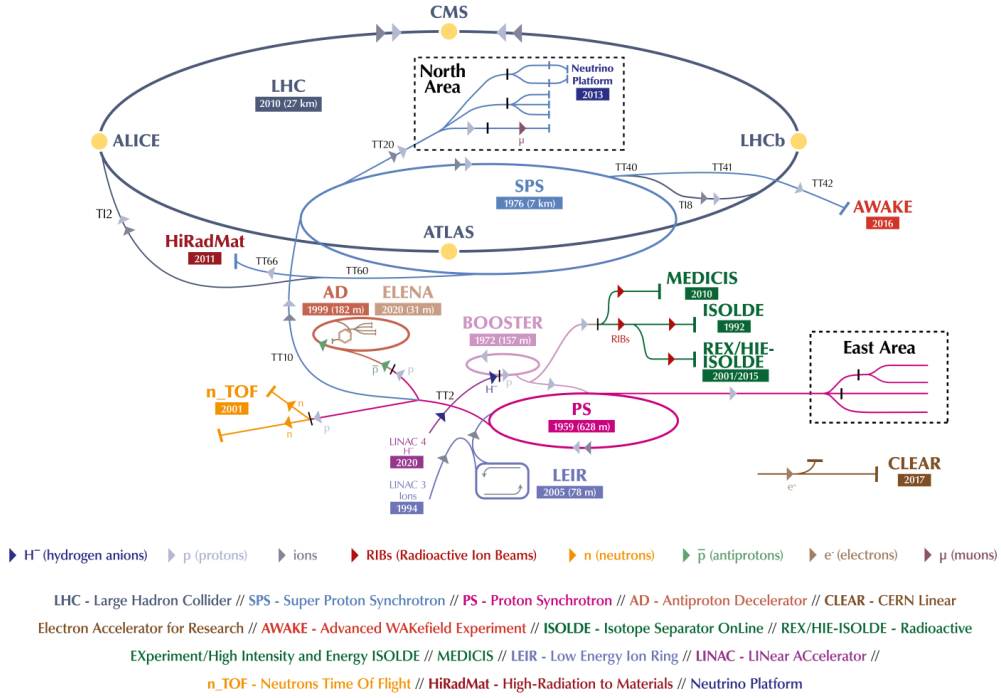


Figure 2.1: The CERN accelerator complex[2].

Proton Synchrotron Booster (2 GeV), Proton Synchrotron (26 GeV), Super Proton Synchrotron (450 GeV). Once reaching 450 GeV, the protons are injected into the LHC main ring, where cavities further accelerate them to up to 7 TeV per beam. Within the LHC, thousands of superconducting magnets—primarily dipole and quadrupole magnets—steer and focus the beams. Dipole magnets bend the particle trajectories, while quadrupole magnets act as focusing elements, ensuring beam stability and optimal collision conditions.

The counter-rotating beams travel at velocities near the speed of light before colliding at four main interaction points, where major experiments such as ATLAS, CMS, ALICE, and LHCb are located. Just before reaching the collision points, the beams undergo squeezing, a process that increases their density and enhances the likelihood of high-energy interactions. To maximize data collection and collision rates, the LHC follows an operational cycle in which beams circulate for approximately 10 hours before being dumped and replaced. This process is continuously repeated to maintain optimal collision statistics throughout each run.

2.1.2 Collisions at LHC

In the Large Hadron Collider, protons are organized into discrete packets known as bunches, each containing approximately 10^{11} protons under nominal conditions. These bunches follow a two-dimensional Gaussian transverse distribution and are injected into the accelerator with a temporal spacing of 25 ns, allowing them to complete a full revolution around the 27 km LHC

tunnel in $89.1 \mu\text{s}$. This revolution is referred to as an LHC orbit, which can accommodate up to 3564 bunches. The bunch crossing (BX) frequency between the two counter-rotating beams is 40 MHz. However, due to operational constraints, the bunch structure is not continuous and, during Run 3, a maximum of approximately 2400 bunches is typically filled, leading to an effective bunch crossing rate of ~ 30 MHz. These bunches are arranged in *trains*, sequences of closely spaced bunches, separated by gaps. Additionally, some bunches may be isolated, meaning they are positioned individually between two trains. At the end of each orbit, there exists an abort gap, typically $3 \mu\text{s}$ long, where no bunches are present. This gap serves multiple purposes, including allowing for safe beam extraction during an emergency shutdown or at the end of a fill, as well as providing time for synchronization and calibration of the front-end electronics in the experiments. LHC filling schemes are periodically modified to accommodate different experimental needs and optimize collision rates.

Another important quantity related to LHC operations is the *instantaneous luminosity* \mathcal{L} , which is closely related to the number of effective collisions at a specific time t . The average instantaneous luminosity for Run 3 is currently $\mathcal{L} = 2 \cdot 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$. From the instantaneous luminosity, it is possible to obtain the *integrated luminosity* $L = \int_{t_0}^{t_1} \mathcal{L}(t) dt$ which corresponds to the total number of collisions delivered to an experiment in the time interval $[t_0, t_1]$. In order to collect as much statistics as possible, the LHC is designed to maximize luminosity, but this comes at the price of introducing *pileup* (PU), namely the occurrence of multiple proton-proton collisions in the same bunch crossing. Current average pileup values are approximately $\langle PU \rangle = 60$.

When it comes to detecting collision products, it is important to establish a reference frame and an adequate coordinate system that best suites for representing the position of such particles. The CMS experiment defines its detector geometry and collision events using a right-handed Cartesian coordinate system, centered at the interaction point where bunch crossings occur. In this system the x and y axes define the transverse plane, which is perpendicular to the proton beamline: the x -axis points toward the geometric center of the LHC ring, while the y -axis is oriented vertically. The z -axis represents the longitudinal direction, aligned with the path of the counterclockwise proton beam. Due to the cylindrical symmetry of the detector, a polar coordinate system is also used, as shown in Figure 2.2 (a). In this framework, the radial coordinate r represents the distance from the interaction point, the polar angle θ is defined between r and the z -axis, and the azimuthal angle ϕ is the one between r and the x -axis in the transverse plane. Instead of θ , the *pseudorapidity*

$$\eta = -\ln \tan (\theta/2) \quad (2.1)$$

is used, as its distribution at relativistic regimes results to be approximately uniform for the particles being produced during a collision. The value of η ranges from $-\infty$ to $+\infty$ with such extrema representing angular positions extremely close to the beam axis, as shown in Figure 2.2 (b).

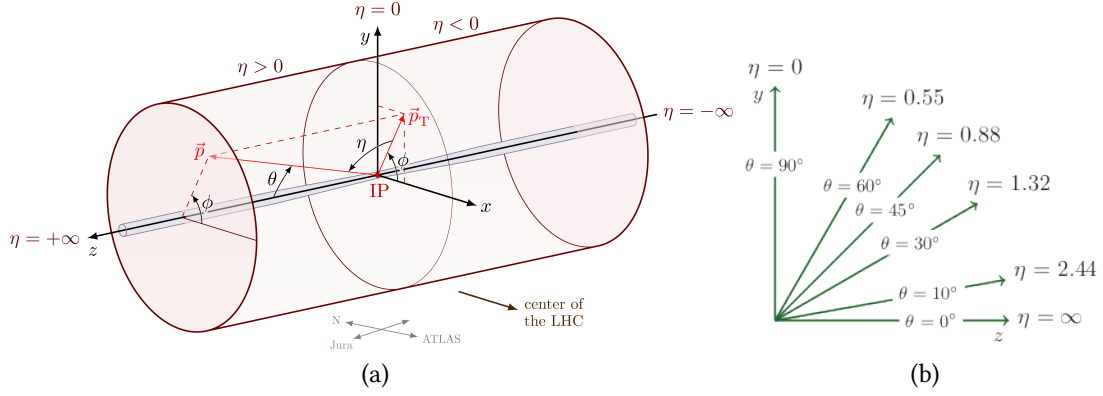


Figure 2.2: (a) Representation of CMS coordinate system. (b) Range of η values.

Other important quantities used in this context are the *transverse momentum*

$$p_T = \sqrt{p_x^2 + p_y^2}, \quad (2.2)$$

which corresponds to the momentum component in the $x - y$ plane, the *angular distance*

$$\Delta R = \sqrt{\Delta\eta^2 + \Delta\phi^2}, \quad (2.3)$$

which gives an index of how much two particles are away from one another, and the *invariant mass*

$$m = \sqrt{\left(\sum_{i=1}^N E_i\right)^2 - \left\|\sum_{i=1}^N \vec{p}_i\right\|^2} \quad (2.4)$$

which is a quantity that allows the identification of particles and decays by combining the energies and momenta of their decay products in a Lorentz-invariant way.

2.2 The Compact Muon Solenoid (CMS) Experiment

The LHC contains four main experiments, each located at a specific interaction point along the 27-kilometer tunnel. ATLAS[3] and CMS [4] are general-purpose detectors designed to study a wide range of physics processes. ALICE[5] focuses on heavy-ion collisions and the study of quark-gluon plasma, while LHCb[6] is dedicated to b-quark physics and CP violation.

The CMS detector is designed as a dense apparatus optimized for precision measurements in a limited volume. It incorporates a powerful superconducting solenoid capable of generating a 3.8 Tesla magnetic field, which enables accurate momentum determination by bending the trajectories of charged particles. Surrounding the solenoid, a steel yoke confines the magnetic field and supports the outer detection systems. A key feature of the detector is its dedicated muon system, which provides reliable identification and tracking of muons as they pass through the outermost layers of the experiment. Despite its smaller size compared to other LHC detectors, CMS is notably heavy, with a total mass of around 14000 tons.

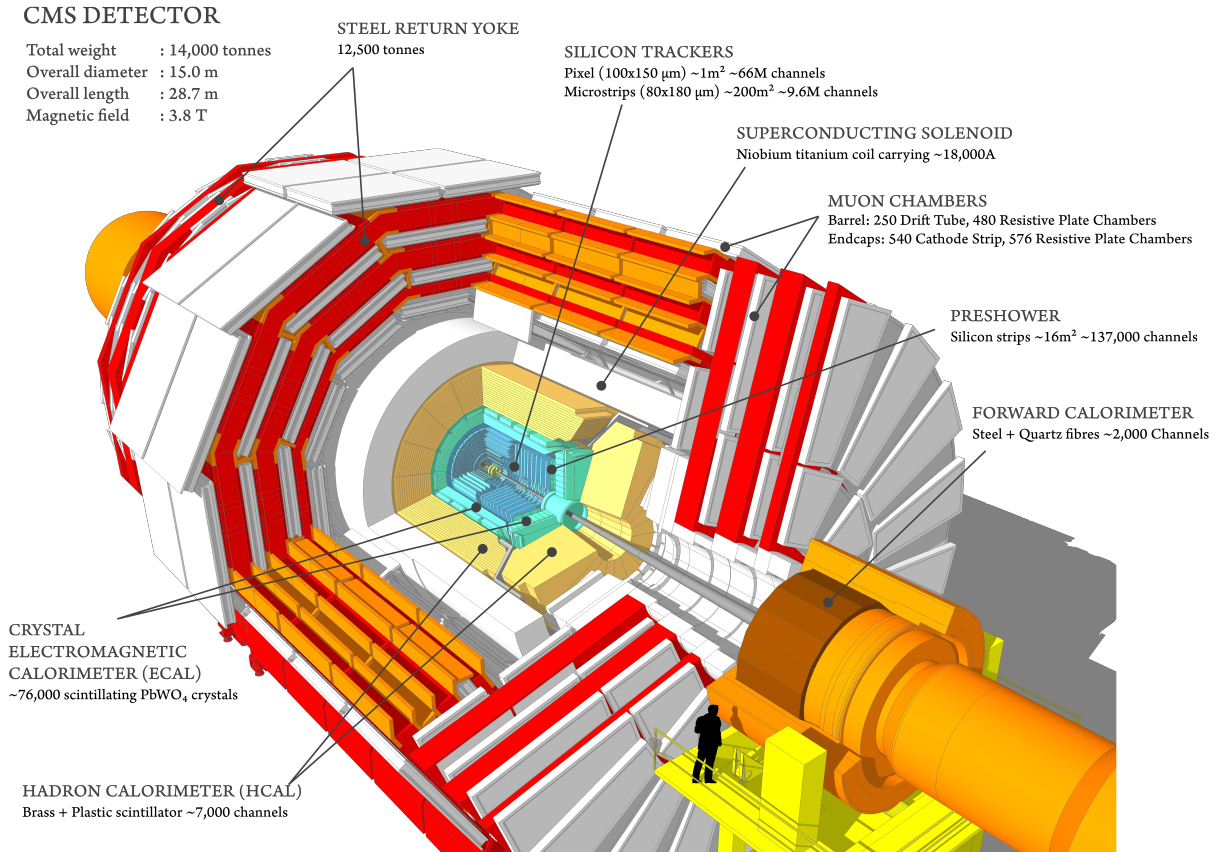


Figure 2.3: Cutaway view of the CMS detector[7].

In this section we first provide an overview of the current CMS sub-detectors, we then discuss the current Level-1 Trigger (L1T) and data acquisition system and we conclude by showcasing the CMS upgrades foreseen for the HL-LHC era, with a particular focus on the new L1 trigger.

2.2.1 Sub-detectors

The goal of a complex detector like CMS is to identify the particles produced during a collision and measuring their properties. To do so, multiple experimental techniques are employed. As charged particles move outward from the collision point, their trajectories bend due to the solenoidal magnet. This bending is crucial for determining both the charge and momentum of particles. In the presence of the same magnetic field, positively and negatively charged particles curve in opposite directions, while high-momentum particles exhibit a larger bending radius compared to low-momentum ones. The detector is also responsible for identifying tracks, which correspond to the trajectories of charged particles. Accurate tracking is essential for precise event reconstruction, and this task is carried out using a combination of the silicon pixel detector, silicon strip tracker, and muon detectors. Measuring the energy of produced particles is achieved using calorimeter detectors. The Electromagnetic Calorimeter (ECAL), located in the inner section, stops and measures the energy of electrons and photons. In contrast, hadrons and jets, which are particle showers resulting from parton hadronization, pass

through the ECAL and are stopped in the Hadron Calorimeter (HCAL), which surrounds it. Muons, unlike other particles, pass through both calorimeter layers without being absorbed. Muon momenta can be measured both by tracking devices and muon chambers depending on the displacement of their origin.

All these techniques are possible thanks to the interplay of multiple sub-detectors, i.e. detectors specialized for the observation of specific collision products. The arrangement of the subdetectors within CMS is shown in Figure 2.3. In the following, we cite the main features of CMS subdetectors.

- The Silicon Tracker[8] is the innermost layer of the CMS detector, designed to track charged particles with high precision near the interaction point. It reconstructs muons, electrons, and hadrons with excellent momentum resolution and efficiency. The tracker is fully silicon-based and detects charged particles through ionization in semiconductor material. It consists of two main substructures: the pixel detector and the silicon strip detector. The pixel detector, forming the inner part, has small pixelated sensors ($100 \times 150 \mu\text{m}^2$) arranged in four barrel layers and three endcap disks, providing spatial resolutions of $10 \mu\text{m}$ (transverse) and $20 \mu\text{m}$ (longitudinal). Surrounding this is the silicon strip detector, composed of multiple layers and endcaps, where strip dimensions vary with distance from the interaction point, ensuring single-point resolutions of $20\text{--}50 \mu\text{m}$ (transverse) and $200\text{--}500 \mu\text{m}$ (longitudinal).
- The Electromagnetic Calorimeter (ECAL)[9] is designed to detect photons and electrons and measure their energy with high precision. The ECAL consists of two regions: the barrel, covering $|\eta| < 1.479$, and the endcaps, extending from $1.479 < |\eta| < 3.000$. The barrel is composed of approximately 61200 lead tungstate (PbWO_4) crystals, each $2 \times 2 \times 23 \text{ cm}^3$, while the endcaps contain 14648 crystals of $3 \times 3 \times 22 \text{ cm}^3$. As photons and electrons traverse these dense materials, they produce visible light, which is collected and converted into an electric signal via an avalanche photodiode. A calibration process then translates the signal into a precise energy measurement.
- The Hadron Calorimeter (HCAL)[10] measures the energy of hadrons and provides indirect detection of non-interacting particles like neutrinos. It must be hermetic to ensure that no particles escape detection, allowing for the reconstruction of missing transverse energy. The HCAL consists of five sections: the barrel covering $|\eta| < 1.3$, the endcaps covering $1.3 < |\eta| < 3.0$, the outer barrel complementing for $|\eta| < 1.4$, and the Hadron Forward calorimeter (HF) covering $3.0 < |\eta| < 5.0$. It is built as a sampling calorimeter, alternating between absorber and scintillator layers. When a hadron interacts with the scintillator, it generates a light pulse, which is collected by optical fibers and converted into an electronic signal. This signal is then amplified and calibrated to provide an energy measurement.
- The muon-tracking system[11] is dedicated to detecting muons with energies ranging

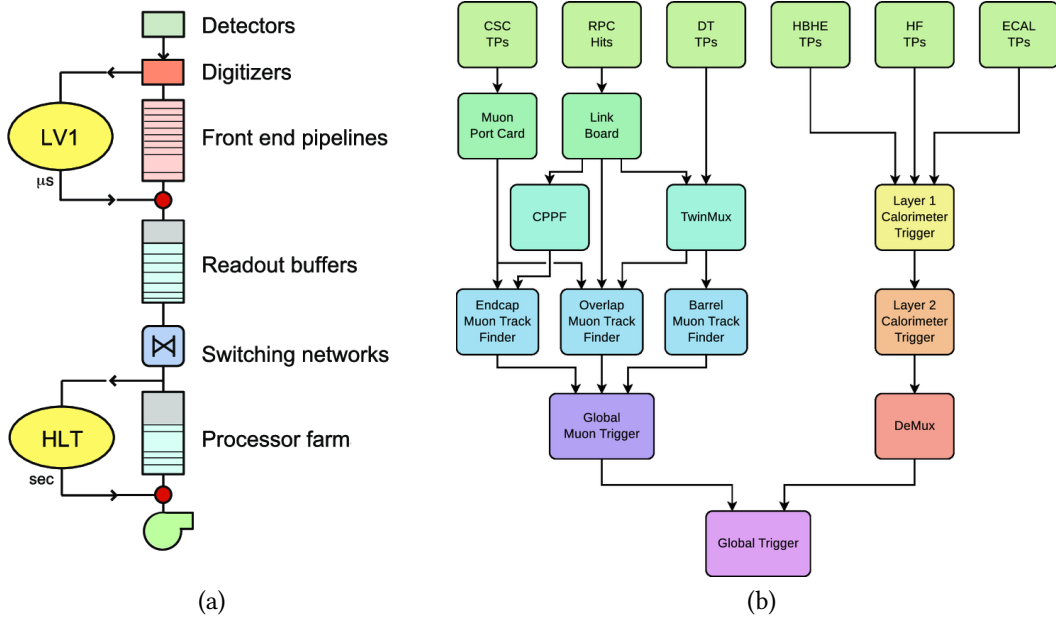


Figure 2.4: (a) Schematic representation of CMS trigger chain. (b) Current configuration of the L1T, after the Phase-1 upgrade.

from a few GeV to hundreds of GeV. As minimum ionizing particles, muons lose little energy while traversing the detector and escape most of its subsystems. The muon chambers are positioned outside the solenoid magnet, using the flux return yokes for mechanical support and additional shielding against hadrons. The presence of a ~ 2 T return magnetic field enables the measurement of muon charge and momentum, complementing information from the inner tracker and enhancing the detector's triggering capabilities. The muon system consists of multiple types of gas detectors, optimized for their respective positions within CMS. In the barrel region ($|\eta| < 1.2$), where muon rates are lower, drift tubes (DTs) are arranged in four concentric layers. These rectangular gas-filled chambers detect muon trajectories by measuring drift times. In the endcap regions ($0.9 < |\eta| < 2.4$), where muon rates and background noise are higher, Cathode Strip Chambers (CSCs) provide fast response times, fine granularity, and high radiation resistance. Each endcap contains four stations of CSCs. To improve muon triggering, Resistive Plate Chambers (RPCs) are distributed across both the barrel and endcap regions ($|\eta| < 1.8$). While their spatial resolution is lower than DTs and CSCs, RPCs offer excellent time resolution, which is critical for precise bunch crossing identification and reliable trigger performance.

2.2.2 Trigger and Data Acquisition Systems

The CMS detector generates approximately 1 MB of data per event, and with a collision rate of 30 MHz, this results in a potential data throughput of ~ 30 TB/s. Storing such vast amounts is unfeasible, especially since most of this data originates from low-energy interactions that

are not relevant to CMS physics goals. To address this challenge, CMS employs a trigger system that reduces the event rate from 40 MHz to ~ 1 kHz, ensuring that only the most significant collisions are recorded. The CMS trigger system consists of two levels: the Level-1 Trigger (L1T)[12] and the High-Level Trigger (HLT)[13]. The L1T, implemented on custom electronics, processes coarse-granularity data from the detector and decides within a few (~ 4) microseconds whether an event should be retained, providing a passed-selection rate up to 100 kHz. If selected, the full event data is sent to the HLT, which performs a refined analysis using full detector resolution and commercial processing units, further filtering events to an output rate of ~ 1 kHz.

The current L1T architecture[14] is divided into two main subsystems: the Muon Trigger and the Calorimeter Trigger. The Muon Trigger[15] processes data from muon detectors across three regions: the Barrel Muon Track Finder (BMFT) for $|\eta| < 0.8$, the Overlap Muon Track Finder (OMFT) for $0.8 < |\eta| < 1.24$, and the Endcap Muon Track Finder (EMFT) for $1.24 < |\eta| < 2.4$. The system reconstructs muon tracks and sends candidates to the Global Muon Trigger (GMT), which removes duplicates and selects the best candidates.

The Calorimeter Trigger[16] reconstructs electron/photon, tau, and jet candidates, as well as global energy sums like missing transverse momentum (p_T^{miss}). Input data from ECAL, HCAL, and HF calorimeters are processed in a two-layer time-multiplexed system, where Layer-1 gathers information and Layer-2 applies physics algorithms before passing results to the Global Trigger (GT).

The Global Trigger (GT)[17] collects and evaluates the best candidates from both trigger systems. Decisions are based on predefined Trigger Menu algorithms, which apply conditions such as p_T thresholds and multi-object correlations. The GT can process up to 512 algorithms simultaneously, issuing a Level-1 Accept (L1A) signal for events that meet selection criteria. Common triggers include single-muon triggers ($p_T > 22$ GeV, ~ 10 kHz rate), electron/photon triggers ($p_T > 30$ GeV, ~ 20 kHz rate), and multi-jet triggers.

The High-Level Trigger (HLT)[18] is the second stage of the CMS trigger system, consisting of a processor farm that runs an optimized version of the full event reconstruction software for fast processing. It reduces the event rate to approximately 1 kHz before data storage. The process begins when an event is accepted by the L1T. The raw detector data are then transferred to an event builder, where the event is reconstructed. The processed event undergoes a filtering stage, where HLT algorithms are applied to determine its relevance. The selected events are categorized into different online data streams, depending on the specific trigger algorithm, before being sent to the storage system.

2.2.3 Event Reconstruction

Once an event has gone through all the trigger and data acquisition chain, it is stored waiting for further analysis: the first step is the event *reconstruction*, i.e. the complete high-level physics objects identification. The Particle Flow (PF) algorithm[19, 20] is responsible for re-

constructing particle interactions by integrating information from all sub-detectors. The process begins with decoding raw detector data, followed by pattern recognition, track finding, and fitting to reconstruct high-level physics objects.

Charged particle tracks are reconstructed using the Combinatorial Track Finder algorithm[21]-based on a Kalman Filter-in three steps: seeding, propagation through detector layers, and final fitting. Tracks are then clustered into primary and secondary vertices, ranked based on the sum of transverse momenta.

Muon reconstruction[22] is achieved by combining information from both the tracker and muon chambers. Muons are classified as standalone, tracker, or global muons, with efficiencies exceeding 96%. The momentum resolution is 1% in the barrel and 3% in the endcaps for $p_T < 100$ GeV, and remains better than 7% for p_T up to 1 TeV[23].

Electrons and photons[24] are reconstructed by clustering energy deposits in ECAL. A Gaussian Sum Filter is used to refine electron tracks, and superclusters are formed to capture bremsstrahlung radiation. Energy resolution for electrons with $p_T \simeq 45$ GeV ranges from 1.6% to 5%, with photons achieving 1% resolution in the barrel.

Jets are clustered using the anti-kT algorithm[25] ($R = 0.4$). Jet momentum is corrected for pileup contamination and calibrated using simulation and in situ balance methods. The jet energy resolution improves with increasing p_T , reaching 5% at 1 TeV.

Hadronically-decaying taus (τ_h) are identified using the hadrons-plus-strips algorithm, distinguishing genuine τ_h decays from quark/gluon jets using the DeepTau algorithm.

2.3 CMS in the HL-LHC Era

Currently, LHC is heading toward the end of the *Phase-1* period—which started more than a decade ago—and, consequently, experimental collaborations and accelerator experts are focusing their efforts into the upgrades foreseen for the *Phase-2* era, which will see the start of the High-Luminosity LHC (HL-LHC)[26] operations. The HL-LHC aims for an instantaneous luminosity of $\mathcal{L} = 5 \cdot 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$, reaching up to $\mathcal{L} = 7 \cdot 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$ in its first years, leading to pileup levels of 140–200 interactions per bunch crossing. The center-of-mass energy will increase to 14 TeV, exceeding previous operational values. Major accelerator upgrades, including new injector systems, cryogenics, and superconducting magnets, will be installed during Long Shutdown 3 (2026–2028), with full commissioning planned for 2029. These enhancements will improve beam manipulation, vacuum systems, and collimation. Over its 12-year operation, the HL-LHC is expected to deliver 3000–4000 fb^{-1} of integrated luminosity, enabling detailed studies of rare phenomena and potential new physics discoveries.

In this section we provide an overview of the upgrades foreseen for the sub-detectors and the L1T.

2.3.1 Sub-detectors Upgrades

The HL-LHC will deliver significantly higher instantaneous luminosity, presenting major challenges for detectors, including increased radiation levels and pileup effects. To address these issues, CMS will undergo extensive Phase-2 upgrades[27], involving subdetector replacements, enhanced trigger systems, and improved data acquisition.

The tracker system[28] will be completely replaced with a new inner tracker, featuring finer pixel sensors, and an outer tracker, which will include strip and macro-pixel sensors extending coverage up to $|\eta| = 4.0$. The outer tracker will also introduce p_T modules, which allow track reconstruction at the L1T for the first time, improving event selection.

The MIP Timing Detector (MTD)[29] will be introduced to provide precision timing measurements with a resolution of 30–60 ps, mitigating pileup effects by correctly associating tracks to their interaction vertices. The Barrel Timing Layer and Endcap Timing Layer will cover up to $|\eta| < 3.0$.

The calorimeter system will also see major upgrades[30]. The ECAL and HCAL electronics will be replaced to improve granularity, noise suppression, and trigger capabilities. The High-Granularity Calorimeter (HGCAL)[31] will replace the endcap calorimeters, extending coverage to $|\eta| < 3.0$. HGCAL will provide precise energy and timing information, improving pileup rejection and Particle Flow reconstruction.

The muon system[32] will undergo multiple enhancements. DT electronics will be upgraded to handle higher trigger rates, while RPCs will see increased readout frequency and the addition of new chambers to extend coverage to $|\eta| < 2.4$. The CSCs will receive new electronics to manage higher data rates. GEM detectors (GE1/1, GE2/1, and ME0) will be deployed in the forward region up to $|\eta| = 2.8$, improving muon track reconstruction and managing the increased trigger rates.

2.3.2 A New Level-1 Trigger System

The CMS Phase-2 Level-1 Trigger[33] is designed to maintain high efficiency in the HL-LHC environment, handling up to 200 pileup interactions per bunch crossing. The system increases the maximum output rate from 100 kHz to 750 kHz, and also the latency budget grows to approximately 12 μ s. A key innovation is the Correlator Trigger (CT), which integrates PF reconstruction at the hardware level, significantly improving object identification. Such big enhancements lead to a complete redesign of the trigger architecture, shown in Figure 2.5.

The Calorimeter Trigger[34] processes ECAL, HCAL, and HGCAL TPs, reconstructing electrons, photons, jets, hadronic taus, and energy sums before forwarding them to the Correlator Trigger for PF-based reconstruction.

The Track Finder (TF) reconstructs charged particle trajectories using p_T modules in the Outer Tracker, applying a Kalman filter algorithm at the full 40 MHz rate. After forwarding the reconstructed tracks to the Global Track Trigger (GTT), the FastHisto[35] algorithm determines the primary vertex, thus filtering tracks to improve downstream PF reconstruction.

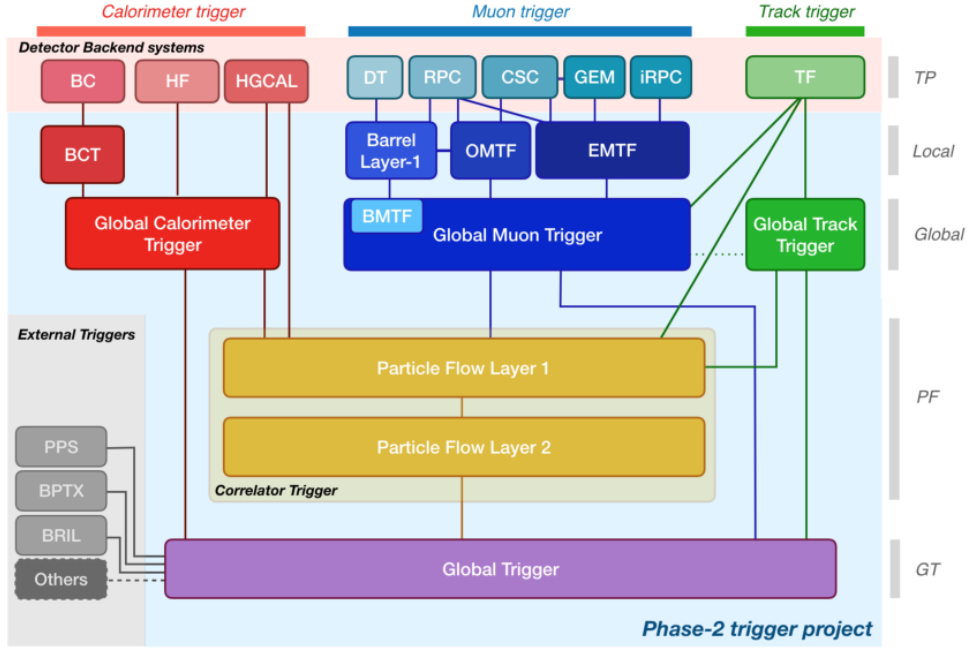


Figure 2.5: Diagram of the Phase-2 Level-1 Trigger[33].

The Global Muon Trigger (GMT) processes hits from the DTs, RPCs, CSCs, and GEMs, improving p_T resolution by matching muon tracks with tracks provided by the GTT.

A key novelty is the introduction of Correlator Triggers (CTs), where for the first time PF reconstruction is performed in hardware[36, 37]. The first CT layer (CL1) receives inputs from standalone muons, L1 tracks, HGCAL 3D clusters, ECAL, and HCAL TPs, executing a regionalized PF reconstruction in $\eta - \phi$ space. The Pile-Up Per Particle Identification (PUPPI)[38] algorithm is applied, suppressing pileup contributions by assigning weights to neutral candidates and rejecting tracks not associated with the primary vertex. The PUPPI objects, significantly reducing redundant data, are sent to the second CT layer (CL2), where global jet clustering[39], missing transverse energy calculation, and hadronic tau reconstruction are performed.

The Global Trigger (GT)[40] evaluates up to a thousand algorithms, incorporating cut-based and machine learning-based selections to optimize signal-to-background discrimination. The GT can analyze objects from multiple bunch crossings, enhancing sensitivity to long-lived particles and exotic signatures.

2.4 The Phase-2 Level-1 Trigger Scouting System

The need for an alternative approach to data collection has been evident within the CMS collaboration for over a decade, as the traditional two-tiered trigger system—regardless of its specific implementation—introduces an inherent bias in the recorded data. A trigger menu restricts the collected events to a predefined phase space region, permanently discarding those that do not pass the selection criteria. With the start of Phase-2, CMS addresses this challenge

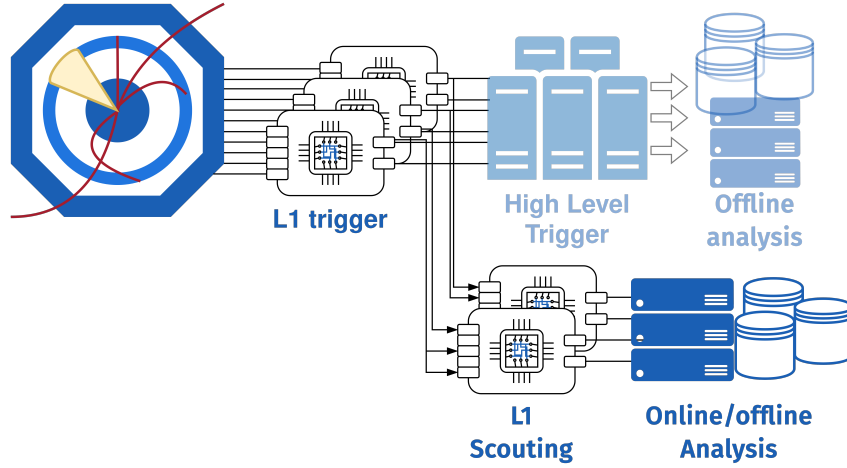


Figure 2.6: Conceptual diagram of Level-1 Trigger scouting[41].

by introducing a brand new Level-1 Trigger scouting system, capable of performing online selection analyses at the full 40 MHz bunch crossing rate. This system enables the exploration of rare and exotic physics signatures that would otherwise evade the traditional trigger chain.

In this section, we first present the physics motivation for a L1T scouting system, then discuss the main technical aspects of its architecture, and finally summarize the current studies regarding its implementation.

2.4.1 The Physics' Case for L1T Scouting

Previous approaches to overcoming the limitations of the traditional trigger system include data *parking*[42], where a portion of raw data is stored and processed at a later time when additional computing resources become available. This method has proven effective in compensating for low trigger acceptance rates in searches for rare decay channels.

Another approach, known as data *scouting*[42], involves performing online selection on the full event stream using coarser reconstructed physics objects. In particular, High-Level Trigger (HLT) scouting was systematically used during Run 2, demonstrating great effectiveness in studying narrow resonances that could not be accessed through conventional trigger selection due to an excessively high event rate. The main advantage of data scouting is that events are processed once online, with only the most relevant reconstructed objects stored, significantly reducing storage requirements compared to saving full event data.

For CMS Phase 2, the L1T scouting system will be introduced[43, 44]. Such system is designed to capture and process L1 objects at the full LHC bunch crossing rate of 40 MHz. It is evident that this approach is conceptually similar to the aforementioned HLT scouting, with the significant difference that the online selection is based on reconstructed objects now accessible at L1 thanks to the massive L1T overhauling planned for Phase 2. Moreover, unlike HLT scouting, this new readout flow operates without rate or latency constraints, since it is completely independent of the usual trigger and data acquisition chain. This system enhances physics reach by accessing events that traditional triggers might miss due to latency

constraints or rate limitations. It also enables studies of exotic topologies, such as correlations across multiple bunch crossings, and offers extensive detector and trigger monitoring with potentially unlimited statistics. However, it is important to stress that L1 objects are optimized for triggering, not physics performance, requiring careful calibration and validation. Studies have shown its potential in rare W decays[45] (e.g. $W \rightarrow 3\pi$, $W \rightarrow D_s\gamma$, $\pi\gamma$), low-mass dijets[41] ($X \rightarrow q\bar{q}$) and exclusive rare Higgs decay channels[46] ($H \rightarrow \rho\gamma$, $\phi\gamma$, $J/\psi\gamma$ or $H \rightarrow 2\rho$, 2ϕ , $\phi J/\psi$).

2.4.2 Baseline Architecture

The CMS Phase-2 L1T scouting system is designed for staged implementation, with a baseline architecture (Stage 1)—shown in Figure 2.7 (a)—focused on collecting data from the decision system (sDS) and global system (sGS). These sources include global trigger results, TCDS metadata, and high-level inputs such as calorimeter, muon, and tracker objects, including outputs from the correlator layer-2 PF and PUPPI algorithms. This configuration provides useful physics data and trigger monitoring with a limited data volume, as only processed quantities are transmitted.

The system can be extended (Stage 2) to include lower-level outputs such as regional muon and calorimeter triggers (sLS), tracking system data (sTS), and even raw detector data (sPS). Reading data closer to the detector front-end increases flexibility in reconstruction but also significantly increases throughput and system complexity.

Trigger data will be collected via spare optical links on L1T boards using 25 Gbps serial protocols. The custom DAQ800 board[47], designed for CMS Phase-2 readout, serves as the interface between synchronous L1 processing and the asynchronous scouting chain. It supports up to 48 input links, with a total input bandwidth of 1.2 Tbps, and delivers up to 800 Gbps output via ten 100 GbE links. Data pre-processing, including zero-suppression, will be handled by Xilinx Ultrascale+ FPGAs with High-Bandwidth Memory (HBM) for efficient buffering. The DAQ800 transmits processed data using a custom TCP/IP protocol to Ingestion Unitss (IUs)—commercial servers optimized for I/O—which buffer and prepare data for analysis. These are followed by Processing Units (PUs) that execute parallel tasks such as re-calibration, classification, and feature extraction. PUs are a fundamental component of the scouting architecture, since they run the actual online selection algorithms developed to identify the aforementioned rare physics signatures. It is then important to optimize the computational load on such units, especially investigating if hardware acceleration devices, like GPUs, FPGAs and heterogeneous SoCs can be beneficial for meeting the processing performance requirements. Indeed, a key factor to consider is scalability, i.e. how many PUs are needed to run a predefined “scouting menu” at 40 MHz: the ideal target, of course, would be to run as many algorithms as possible consuming the lowest possible amount of computing resources. The results provided by the PUs are then aggregated in Storage Units (SUs), with the baseline design supporting up to 5 GB/s storage throughput, about 10% of the central DAQ

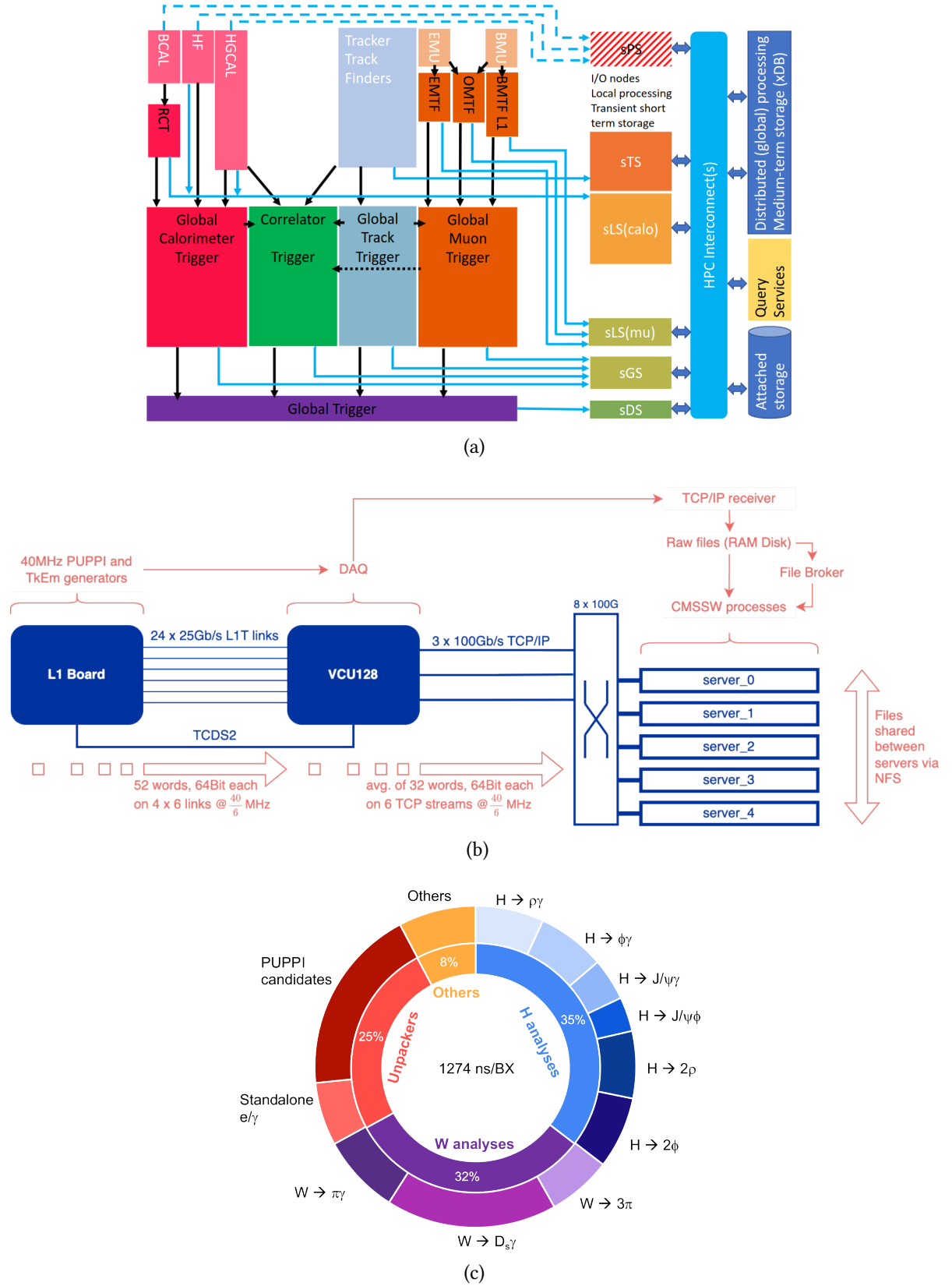


Figure 2.7: (a) Schematic representation of Phase-2 L1T scouting baseline architecture[33]. (b) Technical diagram of the current demonstrator[45]. (c) Timing performance of the demonstrator processing units on a per-BX basis[45].

bandwidth.

Online monitoring and real-time analyses can be performed using DQM-like tools, including full storage of selected bunch crossings and their neighbors to study inter-BX correlations. Any residual bandwidth can be used to store a prescaled sample of full orbits, enabling the development of unbiased analysis techniques.

2.4.3 Phase-2 L1T Scouting Demonstrator

Recently, a first version of the Phase-2 L1T scouting demonstrator has been implemented in order to evaluate on simulation the physics performance of the baseline architecture, utilizing a real hardware setup involving a prototype DAQ and processing system. It is important to note that the expertise gained in assembling and commissioning the Run-3 L1T scouting demonstrator[41, 46]—which captures data from the L1T boards currently installed in the experiment—has significantly contributed to the development of the Phase-2 demonstrator.

The schematic of the demonstrator is shown in Figure 2.7 (b). It features a Xilinx VCU128 development kit, used as a stand-in for the larger and more capable DAQ800 board. The VCU128 receives 24 input links at 25 Gbps and provides three 100 Gbps output ports via QSFPs using the TCP/IP protocol. An additional 10 Gbps input port is dedicated to the LHC clock and synchronization, managed through the TCDS2 protocol. A DTH P1 prototype ATCA board functions as a CT layer 2 traffic generator, producing six output streams. Each stream uses 4×25 Gbps links, simulating L1 PUPPI inputs from six distinct time slices. Indeed, the CT layer 2 is expected to deliver up to 208×64 -bit PUPPI candidate words, transmitted via 4×64 -bit links operating at 360 MHz. Consequently, each board requires 52 frames to transmit all 208 candidates from an event. To sustain the 40 MHz LHC bunch crossing rate, six boards are needed, yielding a total of 24 links that transmit a full-event PUPPI candidates set at a 40/6 MHz rate. The DTH board also generates the TCDS2 clock and synchronization signal, which is delivered to the VCU128 via 10G SFP optics. The Processing Units (PUs) consist of five servers, each equipped with a 96-core AMD EPYC 9654 processor, 768 GB of RAM, and a dual-port 100 Gbps network card. The connection between the PUs and the VCU128 is managed by a 100 Gbps network switch.

A series of tests demonstrated sending 24 links at 25 Gbps from the DTH to the VCU128 successfully. Moreover, the VCU128 correctly performed data aggregation, zero-suppression and transmission of 6 PUPPI candidates parallel streams—32 PUPPI words per event on average—to the PUs via 100 GbE.

Regarding the computing performance, the PUs were able to receive and process 4 of the 6 PUPPI candidates streams (corresponding to a 27 MHz rate) plus standalone e/γ s by running 9 different online selection algorithms, as shown in Figure 2.7 (c). The achieved processing latency, on a BX basis, is $\sim 1.3 \mu\text{s}$. Such processing is currently based on software running within the CMS software framework. The online processing is a key aspect of this research, as its goal is to provide new insights and explore new solutions to enhance the computing power

of the Phase-2 L1T scouting demonstrator. Chapter 4 is entirely dedicated to the optimization and hardware implementation of the $W \rightarrow 3\pi$ analysis using innovative AMD Versal SoCs devices.

Chapter 3

AMD Versal Adaptive SoCs

The Versal Adaptive System on Chip (SoC)[48]—known also as Versal Adaptive Compute Acceleration Platform (ACAP)—is a heterogeneous platform that combines multiple compute architectures into a single chip. The Versal technology has been developed and commercialized by AMD in order to provide a solution to the constantly increasing demand of domain-specific architectures in the realm of high performance and efficient computing. These architectures include:

- **Scalar units** (e.g. CPUs), that are very efficient at executing complex algorithm involving multiple decision branches. They support a wide variety of software libraries, but they are limited in performance scaling.
- **Vector units** (e.g. GPUs and DSPs), that achieve excellent performance on a narrower set of parallelizable tasks, but they are limited by a rigid memory hierarchy.
- **Reconfigurable units** (e.g. FPGAs), which can be precisely tailored to the specific task at hand, making it possible to achieve extremely low latencies at the cost of a long and less flexible development process.

Versal Adaptive SoCs include all of the above units, together with a high-bandwidth Network on Chip (NoC), which allows for interconnections and memory-mapped data movements. Besides the significant evolution in platform customization, AMD also provides a fully unified tool chain that developers can exploit in order to target the units they are specialized on. For example, software developers can target either the scalar processor or the vector processor with C/C++ programs, while hardware developers can target the PL using Hardware Description Language (HDL) kernels written in VHDL or Verilog.

A high-level overview of the ACAP architecture is provided in Figure 3.1. The scalar engines include a Arm[®] Cortex-A72 and a Arm[®] Cortex-R5F, the latter being a real-time processing core. The adaptable engines are hosted inside the PL, which can be reprogrammed to embed a custom memory hierarchy aimed at maximizing the memory access efficiency. Eventually, the Adaptable Intelligent (AI) engines[49] consist of an array of Very Long Instruction Word (VLIW), Single Instruction Multiple Data (SIMD) processing cores, each one equipped

with dedicated memory and interconnections. AI engines constitute the actual novelty of Versal ACAPs, and their purpose is to increase the parallel workload computation, digital signal processing and machine learning inference power w.r.t. the previous generation of AMD SoCs. AI engines architecture and features will be discussed extensively in Section 3.1. The specific availability of each compute unit depends on the Series of the Versal device: in this work we will always refer to the AI Core Series.

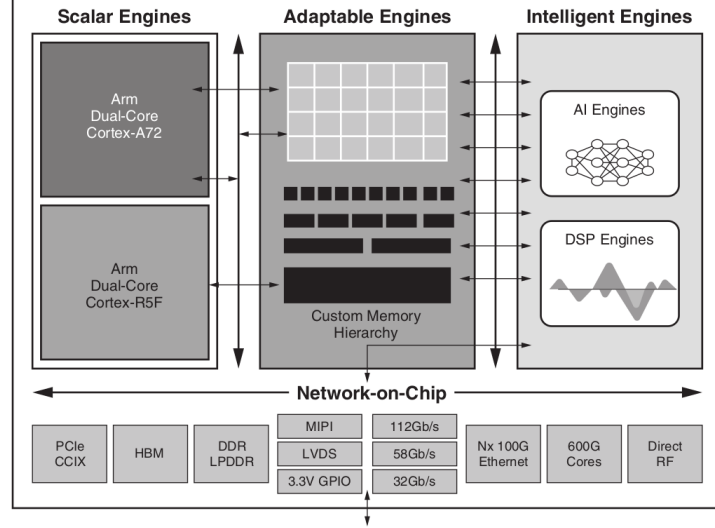


Figure 3.1: High-level block diagram of a Versal Adaptive SoC [48].

3.1 Adaptable Intelligent Engines Architecture

Versal AI Core Series devices include the AI engine array, which is a 2D array of AI engine tiles. It also includes interface tiles that connect the array with the programmable NoC and the PL. In this section we first provide an overview of the tile architecture[50], and then we showcase in detail the AI engine processor, the AI engine interfaces and the tile memory module. Finally, we comment on the parallelism levels achieved thanks to AI engines architecture and features.

3.1.1 AI Engine Tile Overview

AI engine tiles are the computing nodes of the array. In order to fully exploit the SoC, it is key to leverage the array so that single tiles can crunch data and sustain a continuous data flow. Each tile contains an interconnect module, a memory module and one AI engine. A schematic view of the tile is visible in 3.2.

The tile interconnect serves both AXI4-Stream and memory-mapped AXI4 input/output transactions. The memory-mapped AXI4 interconnect is generally needed whenever it is necessary to write/read to/from memory banks of different tiles. The AXI4-Stream interconnect instead is a fully-programmable, 32-bit crossbar that allows for direct AXI4-Stream transaction as well as AXI4-Stream to memory-mapped and vice-versa data transfers.

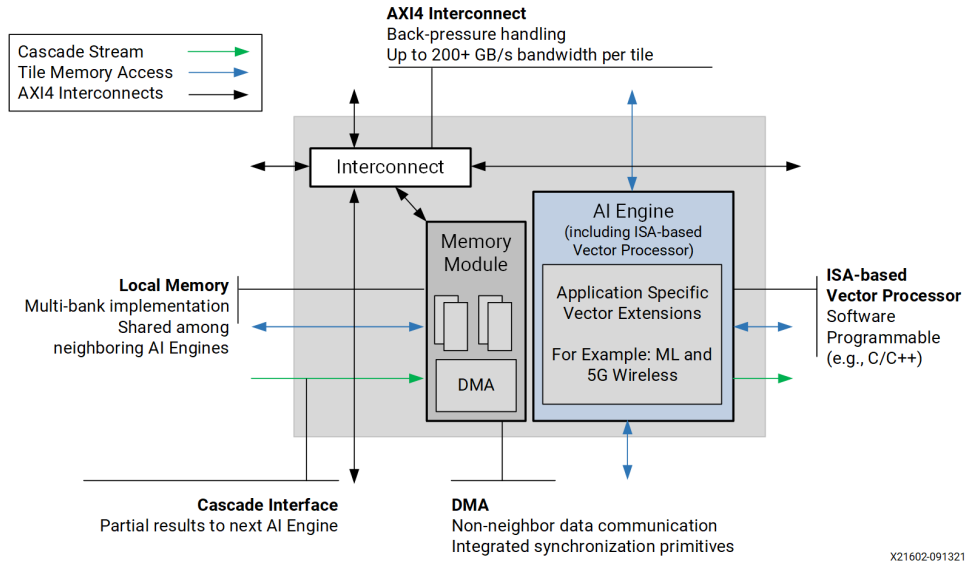


Figure 3.2: Schematic view of an AI engine tile[50].

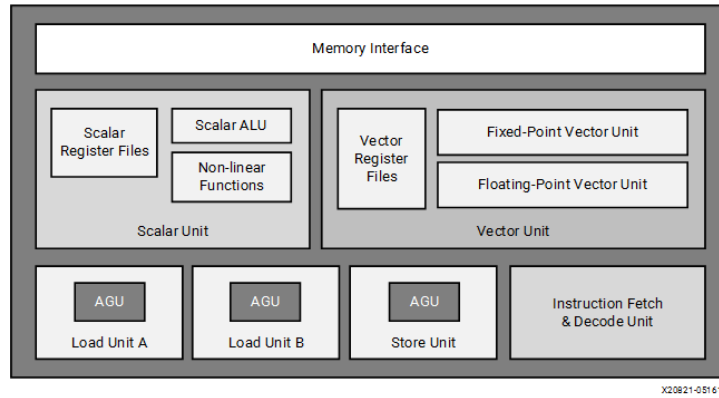


Figure 3.3: Diagram of AI engine processor architecture[50].

The memory module consists of eight banks providing a total of 32 KB available space, together with a memory interface, DMA and locks.

Finally, the proper AI engine is a hard-silicon processor that includes both a scalar and vector core, each one equipped with dedicated registers. The AI engine comes with its own program memory and a rich set of interfaces.

3.1.2 AI Engine Processor

The AI engine processor is a Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) processor equipped with a scalar unit, a vector unit, two load units (load AGUs), a single store unit (store AGU) and an instruction fetch and decode unit, as shown in Figure 3.3.

X Operand	Z Operand	Output	MACs/Cycle
8 real	8 real	48 real	128
16 real	8 real	48 real	64
16 real	16 real	48 real	32
16 real	16 complex	48 complex	16
16 complex	16 real	48 complex	16
16 complex	16 complex	48 complex	8
16 real	32 real	48/80 real	16
16 real	32 complex	48/80 complex	8
16 complex	32 real	48/80 complex	8
16 complex	32 complex	48/80 complex	4
32 real	16 real	48/80 real	16
32 real	16 complex	48/80 complex	8
32 complex	16 real	48/80 complex	8
32 complex	16 complex	48/80 complex	4
32 real	32 real	80 real	8
32 real	32 complex	80 complex	4
32 complex	32 real	80 complex	4
32 complex	32 complex	80 complex	2
32 SPFP	32 SPFP	32 SPFP	8

Table 3.1: Peak MAC operations per cycle varying input operand data type[50].

Scalar Processing Unit

The scalar unit is a RISC processor that serves as a general-purpose compute unit. It supports 32-bit scalar operations and also non-linear functions, like sin/cos, and direct/inverse square root. It also provides conversion between fixed point and floating point data types. The scalar unit is targeted whenever the program instructs conditional branches or needs to evaluate comparisons. It is important to specify that the scalar unit does not have a floating point unit, thus floating point operations are emulated using the SoftFloat library. It is then recommended to use scalar floating point operations only when strictly necessary.

Vector Processing Unit

The vector unit is the key component of the AI engine architecture, since it allows to compute operations on multiple values at the same time, i.e. in the same clock cycle. It can be considered as a powerful Digital Signal Processor (DSP) equipped with 128 8-bit fixed-point multipliers and a floating-point unit with 8 single-precision floating-point multipliers. The performance in terms of MACs is summarized in Table 3.1.

The operations performed by the fixed-point unit follow three different data paths, i.e. Multiply Accumulator (MAC) path, the Upshift path and the Shift-round Saturate (SRS) path.

These paths consist of staged pipelines that orchestrate vector multiplication, pre- and post-addition, vector permutation, and accumulator-to-vector transfers.

The floating-point unit includes a single MAC pipeline that is an adapted version of the fixed-point one. It also supports vector element-wise comparison, fixed-to-float and float-to-fixed vector conversion.

All these operations rely on 128-bit wide vector registers that can be concatenated in order to get 256, 512 and 1024-bit registers. Vector registers are a valuable resource of AI engines, so an effective handling of registers becomes key in order to extract performance from the vector unit. The AI engine has also 384-bit wide accumulator registers used to store the results of the vector data path. A typical use case is to store the result of 32-bit integer multiply and accumulate operations. The grouping and naming of vector and accumulator registers is reported in Figure 3.4.

128-bit	256-bit	512-bit	1024-bit		
vrl0	wr0	xa	ya	N/A	
vrh0					
vrl1	wr1				
vrh1					
vrl2	wr2				
vrh2					
vrl3	wr3				
vrh3					
vcl0	wc0	xb	yd (MSBs)		
vch0					
vcl1	wc1				
vch1					
vd0	wd0	xc		N/A	N/A
vdh0					
vd1	wd1				
vdh1					

(a)

384-bit	768-bit
aml0	bm0
amh0	
aml1	bm1
amh1	
aml2	bm2
amh2	
aml3	bm3
amh3	

(b)

Figure 3.4: AI engine vector registers (a) and accumulator registers (b)[51].

3.1.3 AI Engine Interfaces

In order to fully exploit its computational power, the AI engine processor is equipped with several interfaces. In the following we provide the details of the most important ones. The full interfaces diagram is shown in Figure 3.5.

Data Memory Interface

This interface allows the AI engine to access data from all four directions in the 2D array as if it were a contiguous block of $4 \times 32 \text{ KB} = 128 \text{ KB}$ of local memory. It is important to stress the fact that the memory hierarchy of the AI engine does not include any cache-levels schema of traditional, off-the-shelf CPUs: once data is available on the local memory, it can be loaded directly into the scalar or vector registers in order to perform the required computation. Data

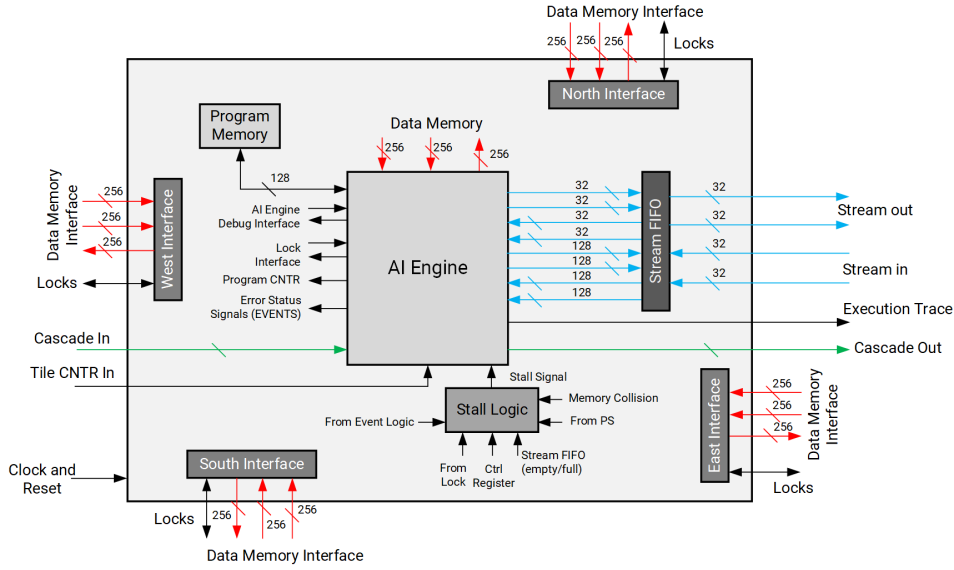


Figure 3.5: Schematic view of AI engine interfaces[50].

memory access is based on two 256-bit wide load units and one 256-bit wide store units, each one supporting 128-bit or 256-bit alignment.

Direct AXI4-Stream Interface

The direct AXI4-Stream interface allows for direct data transfers between tiles, without the need to pass through the memory banks. The AI engine has two 32-bit input AXI4-Stream and two 32-bit output AXI4-Stream interfaces, each one equipped with an input and output FIFO.

Cascade Stream Interface

This interface allows to forward a 384-bit wide accumulator register directly to an adjacent tile, thus creating a chain that can potentially extend row-wise across the whole array.

3.1.4 AI Engine Memory Module

Each AI engine tile has 32 KB of data memory organized in eight 256×128 -bit banks. However, each AI engine can also access the data memory of its north, south and east/west (mutually exclusive depending on the location) neighboring tiles, thus providing a total available memory of 128 KB. Memory arbitrators manage and schedule memory access requests for each bank.

A key unit inside the memory module is the DMA controller that—thanks to its two sub-modules S2MM and MM2S—manages two incoming and two outgoing 32-bit streams. This allows to move data between non-neighboring tiles by using the AXI4 tile interconnect.

AI engine tiles also have a dedicated 16 KB program memory, corresponding to 1024 in-

structions 128-bit wide. This is also a feature to take into account while developing code for AI engines, since the available amount of program memory poses a limit on the size of a program that is supposed to run inside a single tile.

3.1.5 AI Engine Parallelism

We have seen that the architecture and features of AI engines are prompted towards achieving parallelism, both at software and hardware level. In particular, AI engines provide the following:

Instruction Level Parallelism

It is possible to exploit Instruction Level Parallelism (ILP) thanks to the VLIW-based machine code of the AI engine. Indeed instruction words are 128-bit wide, unless differently instructed by compiler optimization. This allows to execute 7 instruction per clock cycle, as shown in Figure 3.6.

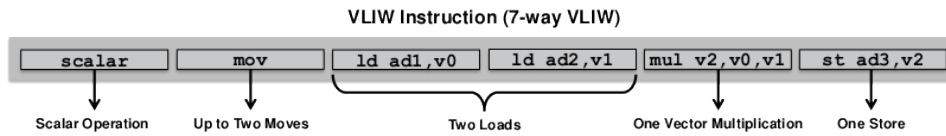


Figure 3.6

Data Level Parallelism

Data Level Parallelism (DLP) comes from the SIMD-based vector processor, since it allows to execute the same operation concurrently on all vector elements (*lanes*). The number of possible operations per clock cycle depends on the vector data type, so it is higher for smaller data types (e.g., 8-bit integers) while it decreases for bigger data types (e.g., 32-bit integers or single precision floats).

Multicore

AI engine acceleration can leverage 400 distinct computing tiles. This is key for scaling an application to work at a fixed rate. Indeed, once the latency of a single-tile program is known,

3.2 AI Engine Coding and Programming Model

In order to extract performance out of AI engines, one has deal with their programming model[51, 52]. The atomic programming unit of AI engines is the *kernel*, which consists of a C/C++ function with arguments specifying the input/output data involved in the computation. Since a kernel will run on one of the multiple AI engine tiles, it must be embedded

inside a *graph*, which consists of a class specifying the interconnections between different AI engine kernels and between AI engine kernels and PL/NoC interfaces. Given the innovative programming paradigm provided by the AI engine architecture, AMD developed a series of custom templated data types and APIs[51, 53] that target the scalar and vector cores, thus enabling the user to exploit the power of AI engines through a relatively simple set of tools. In the source code, these tools are accessible by including the two `aie_api/aie.hpp` and `aie_api/aie_adf.hpp` header files. These libraries are accompanied by a dedicated compiler that maps APIs to scalar/vector operations, scalar/vector data movements and schedules such instructions in a way that maximally exploits the 7-way VLIW power.

In this section we start by showcasing the AI engine data types, we then provide an overview of the most relevant vector operations and related APIs and we conclude by describing the AI engine programming model.

3.2.1 Scalar and Vector Data Types

Programming the AI engine scalar unit limits to basic arithmetic operations based on the supported 8-bit, 16-bit and 32-bit integer data types. Operations on 32-bit float and 64-bit double data types—including nonlinear functions—are emulated using the SoftFloat library.

Regarding the vector unit, the most important data types are:

- **`aie::vector`**. A vector is an object that groups together multiple elements, called *lanes*, of the same type. Each vector is mapped to a vector register of the same bit size. Vectors are parametrized by the element type and number of lanes, provided that the total bit size of the combination is 128, 256, 512 or 1024. Supported combinations are shown in Table 3.2.

Vector Data Types	No. Lanes
<code>int8</code>	16/32/64/128
<code>int16</code>	8/16/32/64
<code>int32</code>	4/8/16/32
<code>uint8</code>	16/32/64/128
<code>float</code>	4/8/16/32
<code>cint16</code>	4/8/16/32
<code>cint32</code>	2/4/8/16
<code>cfloat</code>	2/4/8/16

Table 3.2: Supported vector data types and number of lanes.

- **`aie::accum`**. An accumulator is conceptually similar to a vector since it gathers elements of the same type that are obtained as a result of multiply and accumulate operations. Indeed, each accumulator lane has a larger bit width than the correspondent

vector, allowing for multiple chained operations without incurring in overflow. Accumulators are parameterized by the element type and the number of lanes, where the element type specifies the minimum number of accumulation bits available for each lane. As for vectors, accumulators are mapped to accumulator registers. Supported accumulator types and lanes are reported in Table 3.3.

Type	acc48	cacc48	acc80	cacc80	accfloat	caccfloat
Native accumulation bits	48		80		32	
Lanes	8 – 128	8 – 64	2 – 64	2 – 32	4 – 32	4 – 16

Table 3.3: Supported accumulator data types, number of lanes and accumulation bits[51].

- **aie::mask**. A mask is similar to a vector, with the difference that each element is boolean. Thus, a mask is parametrized only by the number of lanes, whose possible values are the same as vectors. Masks are returned by vector APIs that perform comparison operations and can be used for vector filtering.

3.2.2 Vector Operations

AI engine APIs provide a rich variety of vector operations, including initialization, load/store, arithmetic, reduction and others. Discussing the details of all possible vector operations is beyond the scope of this work. In the following, we will rather provide an overview of the important ones, that are also the most useful in every “standard” programming scenario.

The basics of vectors in AI engines are load and initialization operations. Loading an `aie::vector` involves fetching data from a producer and assign it to a vector register. Initializing an `aie::vector`, instead, corresponds to writing data into a vector register that has already been assigned to the vector itself. Loading operations can happen by accessing buffers (windows), streams or data memory. In particular, data memory loads are supposed to happen on 16-byte aligned memory, otherwise extra clock cycles are used in order to deal with the misalignment. To initialize a vector, one can leverage dedicated APIs that write a specific initialization constant to a vector register, or copy/concatenate the content of other vector registers and write it to the register at hand. An example is provided in Listing 3.1.

Listing 3.1: Vector definition examples.

```
// load vector from aligned array
alignas(aie::vector_decl_align) const int16 values[8] =
    {48, 14, -17585, 0, -955, 24, 0, -469};
aie::vector<int16, 8> vec_values = aie::load_v<8>(values);
```

```
// load vector from a 16-bit wide stream
aie::vector<int16, 8> vec_stream = readincr_v<8>(stream);
```

AI engine APIs allow for the user to perform element-wise operations on vector lanes. Some of them return another vector object, while others return an accumulator object in order to prevent overflow. The former type includes, for example, `aie::add`, `aie::abs`, `aie::conj`, `aie::neg`, while the latter includes, for example, `aie::mul`, `aie::mul_square`, `aie::mac` and `aie::mac_square`, which are extremely useful tools for calculating physical quantities like ΔR^2 . Other APIs, dedicated to vector reduction, like `aie::reduce_add`, `aie::reduce_max` and `aie::reduce_mul`, return a single scalar value. Comparison APIs return `aie::mask` object that tell whether a vector lane satisfies a condition specified by the user or not. These include, for example, `aie::lt`, `aie::gt` and `aie::eq`. Such APIs are useful if used together with reshaping APIs, like `aie::select`, that combine the content of multiple vectors into another vector based on some user-defined criteria, which is specified by a mask object. In our case, reshaping APIs come useful for selecting in-cone p_T values that have to be summed for calculating the isolation of a particle. Vector operations examples are provided in Listing 3.2.

Listing 3.2: Vector operation examples.

```
// multiply two vectors element-wise, obtaining an accumulator
aie::vector<int32, 8> v1, v2;
aie::accum<acc64,8> vmul = aie::mul(v1, v2);

// multiply and accumulate other two vectors
aie::vector<int32, 8> v3, v4;
vmul = aie::mac(vmul, v3, v4);

// obtain result vector from accumulator and extract max value
aie::vector<int64, 8> vres = vmul.to_vector<int64>(0);
int64 max = aie::reduce_max(vres);
```

3.2.3 Programming Model

The innovative characteristics of AI engines has lead to the definition of a new programming paradigm[52], which answers the question about what does it mean to run a program on an array of multiple interconnected cores.

The fundamental building unit of an AI engines program is a kernel, which is a function that consumes data from input buffers or streams and produces outputs that are flushed through buffers or streams. Kernels do not run in isolation: instead, they are connected within a data flow graph, which is implemented as a Kahn Process Network. It is then possible to depict an AI engine graph as a set of nodes (kernels), in which the actual computation is per-

formed, and edges (buffers or streams) that move data between nodes. This implies that the execution order of kernels is driven only by data availability: a kernel “fires” if and only if it fetches from its input ports the expected amount of data to consume, otherwise it will stall. Even if some kernels take longer to execute due to computational delays or varying communication latencies, the final output remains deterministic, thus ensuring repeatable, predictable results irrespective of hardware execution timing.

An AI engine graph is created as a C++ class, inheriting from the graph class defined in the adf library: kernels are specified as private attributes, graph I/O interfaces are specified as public attributes and all nodes/edges are specified into the class constructor. An example is provided in Listing 3.3. Other than kernel instantiations and connections, Programmable Logic Input-Output (PLIO) declarations specify the interfaces of the graph with the PL. Alternatively, Global Memory Input-Output (GMIO) declarations are used in case the interface is with the global memory. Another important parameter associated with each kernel of the graph is the runtime<ratio>: it specifies the percentage of cycle budgeted belonging to a kernel in a specific tile. A runtime ratio of 0.5 means that half of the total execution cycles will be occupied by the considered kernel, while the remaining half is available for other kernels to run in the same tile. The AI engine compiler then tries to assign multiple kernels to the same tile provided that the sum of the runtime ratios does not exceed 1. In such case, kernels will be mapped to different tiles so that their runtime ratio can be satisfied.

Listing 3.3: Example of AI engine graph (graph.h).

```
#include <adf.h>
#include "kernels.h"

using namespace adf;

class simpleGraph : public graph {
private:
    kernel first;
    kernel second;
public:
    input_plio in;
    output_plio out;

    simpleGraph(){

        first = kernel::create(simple);
        second = kernel::create(simple);

        in = input_plio::create(plio_32_bits, "data/input.txt");
        out = output_plio::create(plio_32_bits, "data/output.txt");
```

```

connect(in.out[0], first.in[0]);
connect(first.out[0], second.in[0]);
connect(second.out[0], out.in[0]);
dimensions(first.in[0]) = {128};
dimensions(first.out[0]) = {128};
dimensions(second.in[0]) = {128};
dimensions(second.out[0]) = {128};

source(first) = "kernels.cc";
source(second) = "kernels.cc";

runtime<ratio>(first) = 0.1;
runtime<ratio>(second) = 0.1;
}
};

```

Eventually, the AI engine tools expect the graph object to be created inside a `graph.cpp` top-level file, in which the `main()` function calls the runtime APIs that manage the execution of the graph, as shown in Listing 3.4. Usually, the `main()` function is encapsulated inside a `#if defined(__AIESIM__) || defined(__X86SIM__) ... #endif` statement in order to control the graph execution only during functional or hardware simulation, while deferring the graph hardware execution to the PS/CPU host program.

Listing 3.4: Example of AI engine graph top-level file (`graph.cpp`).

```

#include "graph.h"

simpleGraph mygraph;

int main(void) {
    adf::return_code ret;
    mygraph.init();
    ret=mygraph.run(<number_of_iterations>);
    if(ret!=adf::ok){
        printf("Run failed\n");
        return ret;
    }
    ret=mygraph.end();
    if(ret!=adf::ok){
        printf("End failed\n");
        return ret;
    }
    return 0;
}

```

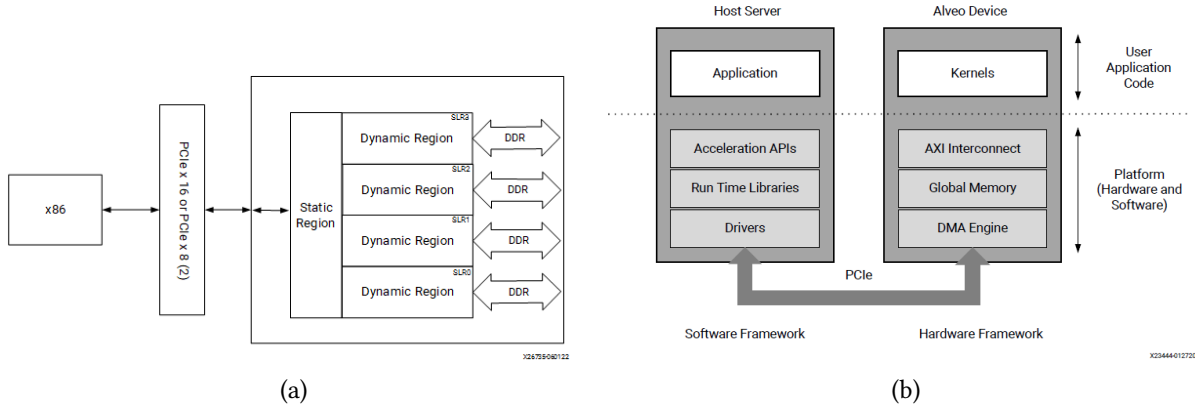



Figure 3.7: (a) Generic AMD data center platform block diagram[54]. (b) Software stack required for running Vitis accelerated application on AMD data center platforms[56].

3.3 Vitis Tools

AI engine programming does not come as an independent task. Indeed, AI engines are only a component of Versal Adaptive SoCs, so, rather than developing an AI engine application, the user targets the whole Versal acceleration platform. In order to guide the user through the application development, AMD provides a set of tools—AMD Vitis unified software platform[54]—consisting of a comprehensive development environment designed to facilitate heterogeneous computing on AMD devices. It enables software applications to run across host processors while leveraging PL regions and AI engines for hardware acceleration, providing a seamless integration of hardware and software components.

Vitis provides a complete tool chain for software development, including compilers and cross-compilers for building applications, debugging tools for identifying and resolving system issues, and program analyzers for profiling and performance optimization. It also includes the Xilinx RunTime library (XRT)[55], which offers APIs and drivers to facilitate communication between software applications and hardware components, managing transactions and data transfers efficiently. By integrating both hardware and software development within a unified environment, Vitis streamlines the deployment of heterogeneous systems using standard C/C++ programming models. The platform supports a variety of design methodologies, including data center application acceleration, RTL kernel design, and embedded system development, ensuring flexibility for different computational needs.

In this work we focus mainly on applications for data center accelerator devices, since it is the solution that best fits the current Phase-2 scouting demonstrator architecture. In particular, in this section we discuss the fundamentals of data center applications on Vitis and showcase the steps needed to compile the application for hardware target.

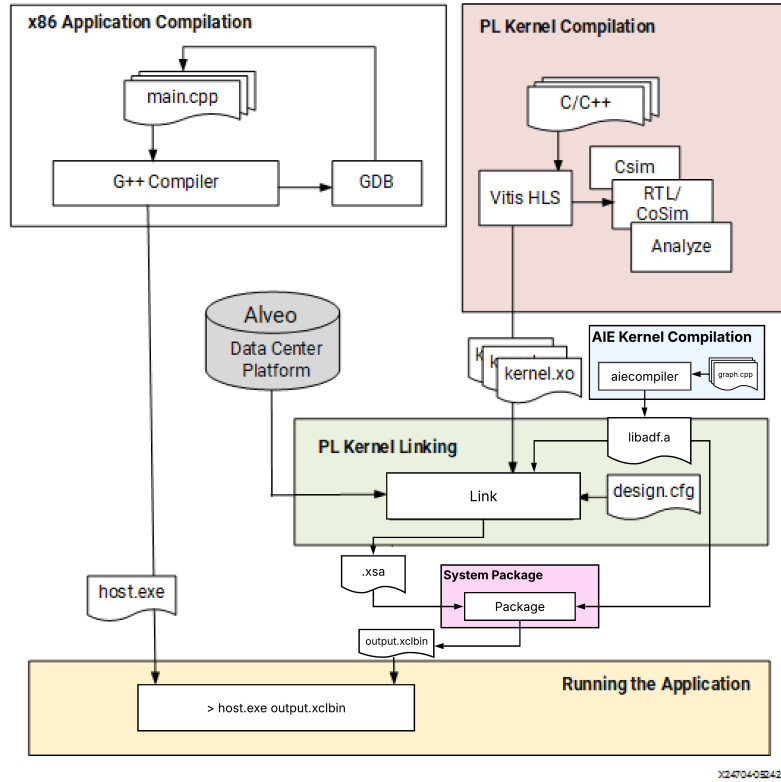


Figure 3.8: Vitis tools flow chart, adapted from[54].

3.3.1 AMD Data Center Acceleration

In the context of data center applications, AMD developed a dedicated product line together with tools that allow for the creation of accelerated applications independently of the specific platform at hand, provided that it embeds the Versal SoC (or other AMD SoC like Zynq). Such data center platforms share the abstract internal structure depicted in Figure 3.7 (a): the platform is divided into a static and a Dynamic Function Exchange (DFX) region, referred to also as *shell* and *user* partitions respectively. During the application development process, the user targets the DFX region using the Vitis tools[54], populating it with PL and AI engine kernels, while the static region provides the basic infrastructure needed by the kernels to operate, like PCIe connectivity, board management, DMA, clocking and resets. It is then important to specify that Figure 3.7 (a) does not represent the internal hardware blocks, but rather the *logical* partitioning of a generic board as seen from a user point of view. Another crucial remark is that the accelerated application is split between the host CPU and the accelerated kernels running on hardware: the communication between these two entities happens through XRT APIs calls made by the host code, which drives control transaction and data transfers across the PCIe bridge. In other words, the acceleration paradigm involves a “master” host code that orchestrates the computation by offloading to the hardware accelerator the most compute-intensive task. A graphic depiction of host/kernels interplay is visible in Figure 3.7 (b).

The development flow includes the following sequential steps:

1. PL kernels are written in HLSs[54] (C++) and compiled using the `v++ -c` command.

This outputs a Xilinx Object (`.xo`) file, which corresponds to a RTL version of the same kernel, where all the necessary code for the required interfaces has been automatically added by the compiler. Alternatively, the user can develop a PL kernel directly in RTL, but this takes extra steps as all control interfaces need to be added and validated manually.

2. AI engine kernels are compiled using the `aiecompiler` command, which generates a `libadf.a` file containing all the configuration metadata and executable files for the AI engine tiles.
3. PL and AI engine kernels are linked with the hardware target platform using the `v++ -l` command. Here the user specifies the interconnections among kernels as well as between kernels and NoC or kernels and DDR. This is the most important and compute-intensive step, as the Vitis tools run Vivado synthesis and place and route functions in the background, implementing kernels into the DFX region of the system. This step generates a fixed platform (`.xsa`) file.
4. The `.xsa` and `libadf.a` files are packaged using the `v++ -p` command, obtaining a `.xclbin` file, consisting of a container that wraps together the binary file, needed to reconfigure the FPGA fabric of the DFX region, and the `libadf.a` file, which instead configures the AI engine array.
5. Finally, the host code manages device memory allocation and kernel execution via XRT APIs. It takes as command-line input the `.xclbin` file, reconfiguring the DFX region of the platform at runtime. The host code is compiled using the standard `g++`, provided that all XRT headers and libraries are properly linked. The output of the accelerated application is also retrieved by the host code, that can then continue the data processing by forwarding such output to other compute units.

Figure 3.8 shows a schematic summary of the Vitis tool chain. In the case of embedded Versal devices, the tool chain is similar, but extra steps are needed in order to run the PS host application on the embedded processor, as a custom Linux installation is needed.

3.4 Deploying Accelerated Applications on Hardware

In this section, we provide a detailed discussion of how to run Vitis accelerated applications on hardware. Indeed, in the previous section we outlined the steps that the user has to follow in order to develop a Vitis accelerated application for a data center platform, and now we delve into the practical aspects of running the final `.xclbin` container on a real hardware device. We begin by presenting the device used in this work, i.e. the VCK5000 Versal Development Card, we then continue with discussing the software stack needed to effectively use the board, and we conclude by showcasing the execution flow of the accelerated application.

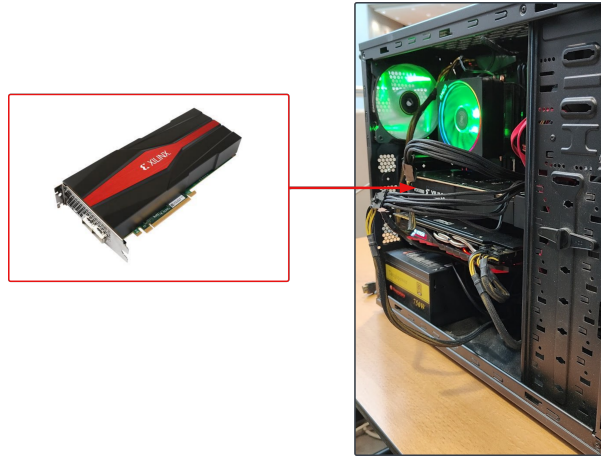


Figure 3.9: Experimental VCK5000 setup on a standalone desktop tower. The board is mounted on the available slot between the GPU and the dissipator fan. It is possible to see the AUX cables hanging out of the case.

3.4.1 AMD VCK5000 Versal Development Card

The AMD VCK5000 Versal Development Card[57] is the first data-center-oriented device commercialized by AMD featuring AI engines. More specifically, the VCK5000 is equipped with a VC1902 ACAP chip that includes 400 AI engine tiles, 1968 DSP engines and a powerful FPGA fabric. More specifications are reported in Table 3.4. Apart from the Versal SoC, the card is equipped with a 16GB Micron DDR off-chip memory organized in 4×4 GB modules. This component plays a crucial role in operating the card, since all the data coming from the host must be stored in the DDR before being consumed by the compute engines. Moreover, the card integrates $2 \times$ QSFP28 and PCIe Gen. 3×16 / Gen. 4×8 interfaces. The PCIe interface, in particular, allows for communication and data transfers between the host and the card itself. Another key component of the device is the Satellite Controller (SC), a low-power microcontroller that keeps track of the card status and continuously reads temperature and power sensors. When the host requests status information about the device, such request is forwarded through PCIe interface to the Card Management Controller (CMC)—usually a Microblaze soft processor inside the FPGA—which eventually queries the SC.

3.4.2 Installing the Card and the Deployment Software

In order to effectively use the card, the first step is to mount the device on the host system. This requires a PCIe slot to be present and available on the host motherboard. The card plugs into the PCIe slot in a GPU-like fashion. Moreover, depending on the characteristics of the host system, the board fan assembly could be necessary. In our case, since the host system is a tower desktop with a single backplate fan, the fan assembly was mounted at the right end of the board to provide adequate cooling. The fan assembly should not be necessary in a server-like setup, where each server rack has already its own fan assembly. Our assembled setup is visible in Figure 3.9.

Feature	Resource Size or Count
AI engines	400
AI engine data memory blocks	3,200
AI engine data memory (Mb)	100
DSP engines	1,968
System logic cells	1,968,400
CLB flip-flops	1,799,680
CLB LUTs	899,840
Distributed RAM (Mb)	27.5
Block RAM blocks	967
Block RAM (Mb)	34.0
UltraRAM blocks	463
UltraRAM (Mb)	130.2
NoC master/slave Ports	28
DDR bus width	256
DDR memory controllers	4
CCIX & PCIe (CPM)	1x Gen3 x16, 2x Gen4 x8, 2x CCIX x8
Multirate Ethernet MAC	4
GTY transceivers (28.125 Gb/s)	24

Table 3.4: VC1902 hardware resources specifications.

Once the board is fixed into place, all the required software stack[57, 56] must be installed both on the host and device side, as shown in Figure 3.7 (b):

- The Xilinx RunTime library (XRT) must be installed on the host in order to provide the necessary libraries and drivers to communicate and exchange data with the board via PCIe. XRT comes with two fundamental programs: `xbmgmt` provides management utilities that, most importantly, allow the user to flash a specific firmware inside the device. `xutil`, instead, allows for monitoring the status of an application running inside the board.
- The development platform (.xpfm file) must be installed on the host in order to allow the Vitis tools to compile and synthesize the application for the specific platform at hand. This is necessary in order to tell the compilation tools which are and how are organized the target device resources.
- The deployment platform (.xsabin file) must be flashed on the board using the `xbmgmt` program command. This sets up the static and dynamic DFX regions inside the device, shown in Figure 3.10. The percentage available inside the DFX partition is almost 100% for all types of chip resources. Regarding the off-chip resources, the selected platform provides 12 out of 16 GB of RAM, two scalable clocks, two QSFP28 I/O and a Xilinx Direct Memory Access (XDMA) engine for PCIe data transfers.

Software	Version
Host OS	Ubuntu 20.04.6 LTS
Host Kernel	5.4.0-26-generic
XRT	2.13.478 (2022.1)
Development Platform	xilinx_vck5000_gen4x8_xdma_2_202210_1.xpfm
Deployment Platform	xilinx_vck5000_gen4x8_xdma_base_2.xsabin
Vitis & Vivado	2022.1

Table 3.5: VCK5000 software stack specifications used in this work.

The setup used in this work is reported on Table 3.5. Once the software stack has been installed, it is possible to use the `xbmgmt examine` command to inspect the status of the board, as shown in Listing 3.5: we can see that the flashable partition SC version corresponds to the one installed in the host system, hence the card is active and ready to run applications.

Listing 3.5: `xbmgmt examine -d 0000:08:00.0` command output

```

-----
1/1 [0000:08:00.0] : xilinx_vck5000_gen4x8_xdma_base_2
-----

Flash properties
  Type           : ospi_xgq
  Serial Number  : XFL1JECDFOAL

Device properties
  Type           : vck5000
  Name           : VCK5000-P

Flashable partitions running on FPGA
  Platform       : xilinx_vck5000_gen4x8_xdma_base_2
  SC Version     : 4.4.33
  Platform UUID  : 04624343-B44B-B0A1-3CD4-8A411789FF20
  Interface UUID : AE9AF7BC-A57D-DE8A-3BA8-A3F9262D78BB

Flashable partitions installed in system
  Platform       : xilinx_vck5000_gen4x8_xdma_base_2
  SC Version     : 4.4.33
  Platform UUID  : 04624343-B44B-B0A1-3CD4-8A411789FF20

Bootable Partitions:
  Default        : ACTIVE
  Backup         : INACTIVE

Mac Address      : 00:0A:35:0D:DE:7C

```

: 00:0A:35:0D:DE:7D

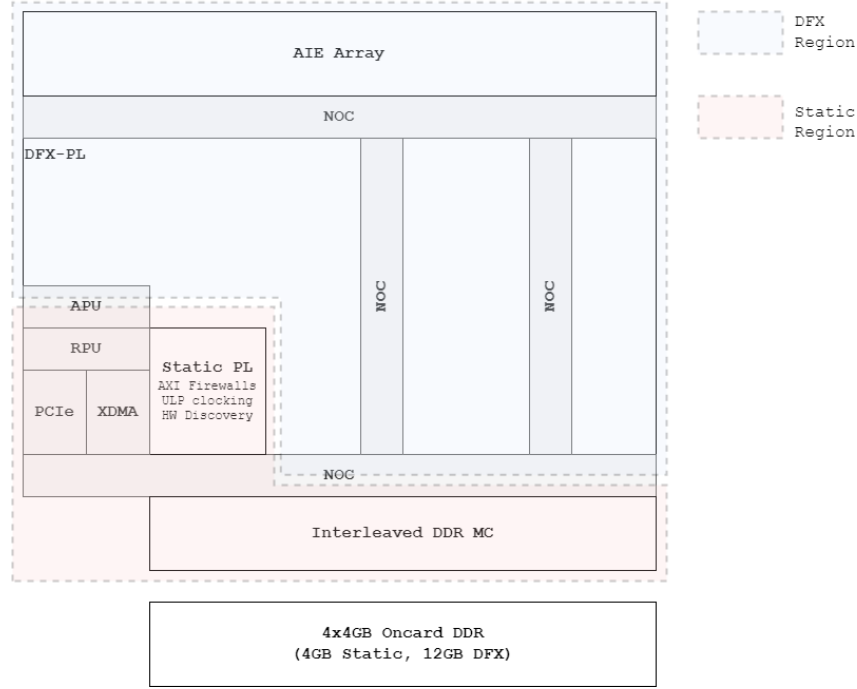


Figure 3.10: Schematic view of the static and dynamic partitions of the `xilinx_vck5000_gen4x8_xdma_base_2.xsabin` deployment platform[56].

3.4.3 Programming the Host and Running the Application

In order to run the application on hardware one must first go through all the Vitis tools pipeline, as discussed in Section 3.3. The final stage is obtaining the `.xclbin` file, which is a container that gathers all the components that will run on the device.

Being an accelerator board, the most straightforward way to program the VCK5000 is through the host application. This means that on the host side we have to write a C++ program that loads the `.xclbin` file into the card, creates kernel object instances, allocates kernel data, runs the kernel and finally reads back the result of the computation to the host memory. All these operations are possible thanks to the native XRT C++ APIs that operate on the XRT objects summarized in Table 3.6. The communications between host and device and vice versa occur via PCIe interface, whose low-level details are hidden from the user, thanks to the XDMA engine hard-coded inside the static region of the device. The host program is compiled using `g++` and can be run directly from terminal. During hardware execution it is also possible to include a `xrt.ini` file in the same directory of the executable, in order to instruct XRT to dump specific trace data that can be used for debugging, profiling or benchmarking purposes—provided that the same profiling options have been specified in the linking phase. A complete working example of host program will be provided in Section 4.3, when discussing the $W \rightarrow$

	C++ Class	Main function
Device	<code>xrt::device</code>	Loads .xclbin file on the device.
Buffer	<code>xrt::bo</code>	Memory r/w operations in both directions.
Kernel	<code>xrt::kernel</code>	Creates kernel instances.
Run	<code>xrt::run</code>	Runs a kernel with a given set of arguments.
Graph	<code>xrt::graph</code>	Creates and runs a graph instance.

Table 3.6: Summary of the most relevant XRT C++ classes.

3π algorithm hardware implementation.

Chapter 4

The Rare $W \rightarrow 3\pi$ Decay Online Selection on AI Engines

Recently, a set of feasibility studies have been performed in order to explore the physics performance of L1 trigger scouting in the HL-LHC era. These studies are based on both Monte Carlo simulated signals and minimum-bias samples as background. So far, the physics processes that have been addressed include W boson decays[45] ($W \rightarrow 3\pi$, $W \rightarrow \pi\gamma$, $W \rightarrow D_s\gamma$), exclusive rare Higgs boson decays[45, 46] ($H \rightarrow QQ$, $H \rightarrow Q\gamma$) as well as other rare channels. In particular, the $W \rightarrow 3\pi$ decay is an excellent candidate for such analyses since its signature is not selected by any general-purpose Phase-1 L1T menu, owing to excessively high p_T thresholds set for hadronically-decaying taus or jets. Moreover, in the context of research and development for Phase 2, it has been studied not only from a physics point of view, but also from a computational perspective with the goal of testing the performance obtained by running such online analysis on traditional CPUs or on hardware accelerators. This is a key point to address to make the scouting Processing Units (cf. Section 2.4) work at 40 MHz.

In this chapter we showcase a complete implementation of the $W \rightarrow 3\pi$ online selection algorithm on AI engines. We start by providing an overview of the cut-based selection, we then discuss the features of the dataset used to test the algorithm, and eventually delve into the hardware implementation exploiting the Versal chip on the AMD VCK5000 Development Platform. Efficiency results, performance benchmarks and a comparison with related works are discussed at the end of the chapter.

4.1 The Cut-Based $W \rightarrow 3\pi$ Selection Algorithm

The rare $W \rightarrow 3\pi$ decay has already been investigated in previous works, where an upper limit for the branching ratio of the 10^{-6} order has been established[58]. Other works followed with the goal of adapting the offline analysis to the online selection scenario of L1T scouting[59].

The current version of the selection algorithm consists in applying a series of cuts/filters on L1 PUPPI candidates features in a sequential fashion until a pion triplet is identified in the event at hand. While some cuts are necessary in order to enforce conservation laws,

other can be optimized to maximize the efficiency of the algorithm. As such optimization has been already addressed by previous efforts, it will not be discussed in this work. The cuts are performed as follows:

1. **Particle ID.** The particle ID is filtered to match the one of charged hadrons ($\text{pid} = 2, 3$) or electrons/positrons ($\text{pid} = 4, 5$), since it is likely to misidentify a pion for an electron.
2. **Transverse momenta.** The p_T cut plays a fundamental role in the analysis since it is the main background discriminator. Three different p_T thresholds—7, 12 and 15 GeV—are established and it is required that there are at least three candidates above the low cut, two above the medium cut and one exceeding the high cut.
3. **Isolation.** Each pion of the triplet must have an isolation value ≤ 2 . This is required in order to suppress QCD background. The isolation is calculated considering a cone around the particle defined by $0.01 < \Delta R < 0.25$. It is important to recall that the isolation of a particle that passes the previous filters requires the information of all the other particles, even those that do not survive the selection so far.
4. **Angular separation.** This is the step of the selection in which triplet candidates are identified. Indeed, up to step 3. we have only filtered out a list of isolated hadrons with a sufficiently high p_T . Now instead we check the possible triplet combinations that satisfy the requirement that each pion has an angular distance of $\Delta R > 0.5$ from the others.
5. **Charge.** Once we have a candidate triplet, we have to make sure that charge is conserved. The W boson comes with ± 1 charge, thus two out of the three charged pions must have opposite charge. A possible combination may be $(-1, +1, +1)$.
6. **Invariant mass.** The last step is to calculate the invariant mass $m_{3\pi}$ of the selected triplet and see if it falls within a reasonable window centered around the W mass, which is approximately 80 GeV. The established window is $60 \text{ GeV} < m_{3\pi} < 100 \text{ GeV}$.

It is important to stress that the order in which cuts are performed depends on the specific architecture for which the selection algorithm is designed: the current baseline is optimized for single-core CPU performance, which differs significantly from the AI engine programming paradigm. Indeed, modern CPUs work at very high frequencies and can leverage powerful branching prediction systems that are particularly fit to our scenario, since a lot of conditions must be evaluated sequentially.

4.2 Signal and Background Dataset

The datasets used in this work are Monte-Carlo-based simulations of the CMS detector in the Phase-2 configuration. The physics process is generated using PYTHIA 8.212, the interaction with the detector is handled by GEANT4 and the event reconstruction is performed by all the

dedicated algorithms in the experiment’s code base.

Both datasets consist of a collection of events grouped in orbits: the background dataset includes minimum-bias, neutrino-gun events (i.e. soft pp interactions) while in the signal dataset each event contains a generated triplet of pions given by the decay of a W boson. It is important to stress that since our goal is developing a L1 search, the objects of our interest stored for each event are PUPPI candidates, which indeed are available in the Phase-2 Correlator Trigger (CT) layer 2 (cf. Section 2.3). The details of signal and background datasets are reported in Table 4.1.

Dataset	Details
Signal	ROOT File: 11Nano_WTo3Pion_PU200.root (gen-matched)
	Binary (PUPPI): 11Nano_WTo3Pion_PU200.dump
	Total Events: 50,000 (~ 14 orbits)
Background	ROOT File: 11Nano_SingleNeutrino_PU200.125X_v1.0.root
	Binary (PUPPI): puppi_SingleNeutrino_PU200.125X_v1.0.dump
	Total Events: 147,600 (~ 42 orbits)

Table 4.1: Summary of Signal and Background Datasets

4.2.1 PUPPI Data Format

In the current setup of the Phase-2 L1T scouting system, the format of PUPPI candidates is a 64-bit word, of which the first 40 are common to all types of candidates and the last 24 vary for charged or neutral candidates. Since for our selection only the common bits are relevant, we will not mention the variable ones.

The features encoded in the first 40 bits are the transverse momentum p_T , the pseudo-rapidity η , the azimuthal angle ϕ and particle identifier pid. An important detail is that all features except the pid are real values, which is however an inconvenient data format to deal with in the context of trigger and data acquisition boards, since they are mainly powered by FPGAs that handle efficiently only integer values. For this reason, the LSB of p_T corresponds to 0.25 GeV, while the LSB of η and ϕ corresponds to $\pi/720$ radians. In this way, we have hard-coded in the integer representation the sensitivity of our features, i.e. the minimum absolute variation needed to distinguish two neighboring values. The encoding is summarized in Table 4.2. The pid can be converted to `pdg_id`, which is used in offline analysis. The mapping is provided in Table 4.3.

Each event in the dataset starts with a 64-bit header that precedes the list of PUPPI candidates. It reports some general information about the event itself, like the number of PUPPI candidates, the bunch crossing number and the orbit number, as shown in Table 4.4.

Range	Field	Bits	Format	LSB
13-0	p_T	14	unsigned int	0.25 GeV
25-14	η	12	signed int	$\pi/720 = 1/4$ deg
36-26	ϕ	11	signed int	$\pi/720 = 1/4$ deg
39-37	PID	3	unsigned int	

Table 4.2: Common features for all PUPPI candidates.

PID	Binary	Particle	PDG ID
0	000	h^0 neutral hadron	130
1	001	γ photon	22
2	010	h^- hadron of charge $-$	-211
3	011	h^+ hadron of charge $+$	+211
4	100	e^- electron	+11
5	101	e^+ positron	-11
6	110	μ^- muon	+13
7	111	μ^+ anti-muon	-13

Table 4.3: pid and pdg_id encoding.

Range	Size	Info
11-00	12	Number of Puppi candidates
23-12	12	Bunch crossing number (0-3563)
55-24	32	Orbit number
60-56	6	Run number
61	1	Error bit
63-62	2	10 = Valid event header

Table 4.4: PUPPI header features.

4.2.2 Gen-matching and Target Triplets

In order to assess its performance, it is important to define an objective that the algorithm is supposed to target efficiently. In our scenario, this translates in checking that the reconstruction algorithm is able to identify the L1 objects associated to the pion triplet originated from the decay of the W boson on a per-event basis.

To help with this, the dataset is subject a gen-matching process, which consists in identifying, for each event, the L1 PUPPI candidates that are closest—in terms of η and ϕ —to the generated pion triplet. This is backed by the fact that, at least theoretically, a generated pion, once detected by the experiment, should appear among the L1 reconstructed objects. In practice however, this is not always verified. Indeed, one of the pions may have been generated outside the detector acceptance, or simply could have escaped the reconstruction algorithms

running on the trigger. For this reason, the first step towards defining our selection target is filtering out the events with a generated triplet that is outside the detector acceptance. By applying this filter, in the dataset there will be only events that we should expect to reconstruct at L1.

However, it may happen that a reconstructed and gen-matched triplet contains a particle that is not actually a pion. This could be explained by the fact that the gen-matching depends only on geometrical features of L1 reconstructed particles (i.e. their proximity to the generated ones), not their actual identification (i.e. hadron, muon, etc.). It is thus important to filter the proper gen-matched events, i.e. those with three gen-matched hadrons, from all the other ones. We will refer to these correctly gen-matched triplets as “good” triples or “target” triplets. This is done by checking that all `pdg_id` of the gen-matched triplet are ± 211 or ± 11 (or equivalently that all `pids` are between 2 and 5). Table 4.5 summarizes the gen-matching results for the signal dataset.

Metric	Ratio	On Acceptance
Acceptance	20383/49896 = 40.9%	–
Matched	10372/49896 = 20.8%	50.9%
Good	9757/49896 = 19.6%	47.9%

Table 4.5: Summary of acceptance, matched, and good events ratios. The ratio is taken out of 49896 events, corresponding to 14 full orbits.

4.2.3 Dataset Format and Reprocessing

The Monte Carlo simulation outputs two different dataset formats, both for signal and background. The root file contains the full-granularity dataset and is meant to be used for offline reconstruction. Indeed, all the L1 PUPPI features are already unpacked and converted to floating point precision. In the case of signal dataset, other than the L1 PUPPI candidates, it also stores p_T , η and ϕ of the generated pions and the results of the gen-matching process. The dump file instead is a binary file containing only headers and PUPPI candidates in the format discussed in previous paragraphs.

The choice of the dataset eventually landed on the dump file: in this way, we can work with smaller integer variables that allow us to use larger vector sizes in the AI engine, thus enhancing parallelization and reducing compute time. Indeed, from Table 4.2 we can see that all PUPPI features can be stored using 16-bit integers. A side effect of this choice is that the features need to be unpacked out of each 64-bit PUPPI row, which adds on top of the algorithm compute load. It is important to recall from Section 3.2 that the vector lanes available when programming the AI engine scale as powers of 2 depending on the variable size. This means that, for example, it is not possible to declare a 16-bit integer vector with an arbitrary number of lanes. This clashes with the variable number of PUPPI candidates present in each event

of the dataset. As a solution, we decided to reprocess the dump file by removing the headers and zero-padding each event so that they always contain 224 rows. In other words, this corresponds to a reverse zero suppression. Assuming 224 PUPPI candidates per event allows us to store their features into 7×32 -lanes, 16-bit integer vectors, and also to exploit the 32-lanes accumulator for intermediate calculations. This is also a conservative choice if we take into account that the CT layer 2 is expected to deliver at most 208 PUPPI candidates per event (cf. Section 2.4). It is important to remark that such reprocessing introduces an inefficiency in our algorithm, i.e. we deliberately process more data than what is actually needed.

4.3 Hardware Implementation

In this section, we provide a detailed discussion of the online selection algorithm's implementation on the Versal Adaptive SoC, focusing specifically on the AI engine array. Starting from an overview of the algorithm, we discuss the PL kernels, the AI engine kernel and the orchestration provided by the host program. We then run the algorithm on the datasets presented in Section 4.2 and we comment on the computational and physics performance of our design.

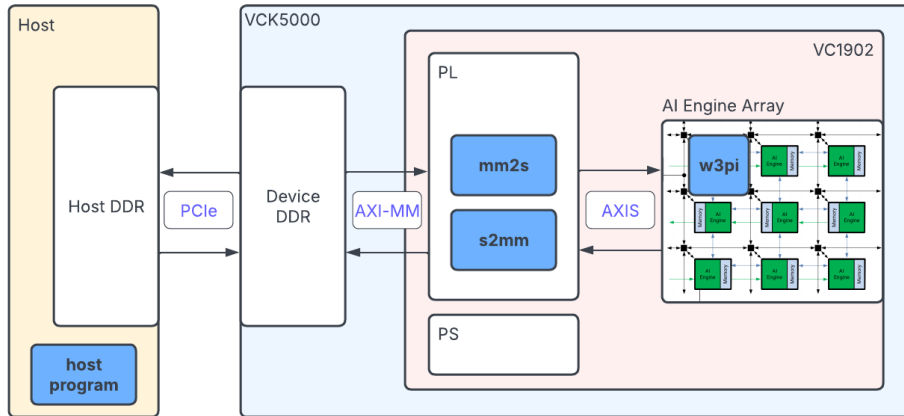


Figure 4.1: High-level diagram of the Versal SoC application.

4.3.1 Design Overview

Our hardware design for the $W \rightarrow 3\pi$ search involves PL kernels, AI engine kernels and a host CPU program¹. This reflects the heterogeneous nature of Versal SoCs. Before discussing the role of each component, it is important to designate the high-level operations that the design is supposed to perform on a per-event basis. These involve

1. Allocating data memory on the device,
2. Unpacking the 64-bit PUPPI rows into single features,

¹The source code is available upon request to the author.

3. Feeding the unpacked data to the AI engine tiles,
4. Applying the selection,
5. Retrieving the result from the AI engines,
6. Sending the result back to the host.

While steps 1. and 6. are managed by the XRT[55] APIs in the host code, all the others could potentially fit in the AI engine array. On one hand this would simplify the design, since data could be fed to the array directly from the NoC without the need of PL kernels, but on the other hand this would excessively increase the computational load of the tiles, thus forcing the use of two or more tiles per algorithm instance. This results in at least a $\times 2$ factor in the AI engine tile usage when calculating the total number of compute units needed to run the design at 40 MHz. It is then a reasonable self-constraint using one kernel instance per AI engine tile. To achieve this, steps 2., 3. and 5. have been implemented as PL kernels. Indeed, the FPGA fabric is well suited for interfacing the device DDR memory, unpacking the data and streaming it to the AI engines.

Other key aspects to consider are memory management and kernel execution. The AI engine kernel—running on a single tile—applies the selection on a per-event basis and by default the `.xclbin` file enables the graph to run forever, without the need of explicit calls from the host program. This means that the kernel is executed as soon as its interfaces find enough data to consume in order to start the computation. Analogously, the kernel is released once the output interfaces are completely filled with the result of the computation. On the contrary, PL kernels usually need to be called from the host program, and if the number of subsequent calls is high, e.g. a full orbit of events, this will introduce a significant overhead resulting in breaking the stream feeding the AI engine kernel. As a first approach, we then decided to run the PL kernels on a per-orbit basis, thus requiring just a single kernel call for 3564 events.

The final design pipeline involves the host code, two PL kernels (`mm2s` and `s2mm`) and one AI engine kernel (`w3pi`) running on a single tile:

- The host code loads the `.xclbin` file and reconfigures the DFX region of the board to include the `mm2s` and `s2mm` kernels in the PL and the `w3pi` kernel on one tile of the AI engine array. It then instantiates a buffer for the input data—i.e. a full orbit—that the `mm2s` kernel will stream to the `w3pi` kernel. At the same time it also instantiates a second buffer that the `s2mm` kernel will use to store the results provided by the AI engine. The host code follows a standard structure whose details have been already discussed in Section 3.4.
- The `mm2s` kernel reads blocks of data from the DDR memory of the device, unpacks them into feature vectors and starts a stream directed to the `w3pi` kernel.

- The w3pi kernel consumes the incoming stream of data while it runs indefinitely.
- The s2mm collects the output from the AI engine tile and stores it in its dedicated buffer.
- Once all the input data has been consumed, the host code copies the results back to the host memory.

An high-level functional diagram of the whole application is visible in Figure 4.1.

Listing 4.1: Signature of mm2s and s2mm kernels.

```

1 void mm2s(ap_int<64 * BLOCK_SIZE>* mem, hls::stream<qdma_axis<32,0,0,0>>& s0,
  ↪ hls::stream<qdma_axis<32,0,0,0>>& s1);
2
3 void s2mm(ap_int<32>* mem, hls::stream<qdma_axis<32,0,0,0>>& s);

```

4.3.2 PL Kernels

The mm2s and s2mm kernels have been written in HLS[60], since it facilitates the design of software-controllable kernels. Indeed, the Vitis v++ compiler automatically generates the interfaces needed to communicate with the host via XRT APIs. These interfaces include any number of AXI4 memory-mapped or AXI4-Stream ports, clock and resets, and at most one AXI4-Lite slave port.

The mm2s kernel signature is reported in Listing 4.1: the three interfaces consist of a AXI4 master mem port and two AXI4-Stream s0 and s1 ports. The mem port is the one interfacing the global DDR memory of the board and its bit width depends on the BLOCK_SIZE constant. This has been set to 16 in order to maximize the width of the word that the PL is able to read from the DDR in one clock cycle, i.e. $16 \times 64 \text{ bit} = 1024 \text{ bit}$. Indeed, the clock frequency of the DDR is significantly higher than the one of the FPGA and this implies that, during one clock cycle, the latter can fetch multiple adjacent DDR memory words and concatenate them into a larger one (*burst* read). As a 1024-bit block is read from memory, each of the 16 64-bit word is sliced in parallel in order to extract the PUPPI features into vectors. The parallelization of the slicing is provided by the HLS UNROLL pragma. This process is repeated 14 times (NUM_BLOCKS), since an event of $224 \times 64\text{-bit}$ words consists of $14 \times 1024\text{-bit}$ blocks. The code is reported in Listing 4.2.

Listing 4.2: mm2s data unpacker block.

```

1 ... // Defining and initializing variables
2
3 for (unsigned int event_idx = 0; event_idx < NUM_EVENTS; event_idx++)
4 {
5     #pragma HLS LOOP_TRIPCOUNT min=1 max=3564

```



```

6
7  ap_uint<16> is_filter_idx = 0;
8  for (unsigned int i=0; i<N_MIN; i++)
9  {
10     #pragma HLS UNROLL
11     is_filter[i] = 0;
12 }
13
14 for (unsigned int i=0; i<NUM_BLOCKS; i++)
15 {
16     #pragma HLS PIPELINE II=1
17
18     ap_int<64 * BLOCK_SIZE> burst_data = mem[mem_offset + i];
19     ap_int<64> buffer[BLOCK_SIZE];
20
21     for (unsigned int j=0; j<BLOCK_SIZE; j++)
22     {
23         #pragma HLS UNROLL
24
25         buffer[j] = burst_data.range(64 * (j + 1) - 1, 64 * j);
26
27         pt    [i * BLOCK_SIZE + j] = buffer[j].range(13, 0);
28
29         eta    [i * BLOCK_SIZE + j] = buffer[j].range(25, 14);
30         eta    [i * BLOCK_SIZE + j] = eta[i * BLOCK_SIZE + j] | ((eta[i *
31             ↪ BLOCK_SIZE + j][11]) ? 0xF000 : 0x0000);
32
33         phi    [i * BLOCK_SIZE + j] = buffer[j].range(36, 26);
34         phi    [i * BLOCK_SIZE + j] = phi[i * BLOCK_SIZE + j] | ((phi[i *
35             ↪ BLOCK_SIZE + j][10]) ? 0xFC00 : 0x0000);
36
37         pdg_id[i * BLOCK_SIZE + j] = buffer[j].range(39, 37);
38         pdg_id[i * BLOCK_SIZE + j] = pdg_id[i * BLOCK_SIZE + j];
39     }
40 }
41
42 for (unsigned int j=0; j<EV_SIZE; j++)
43 {
44     #pragma HLS PIPELINE II=1
45
46     bool is_filter_cond = (pt[j] >= MIN_PT) && ((pdg_id[j] >= 2) &&
47         ↪ (pdg_id[j] <= 5));
48     is_filter[is_filter_idx] = (is_filter_cond) ? ap_uint<16>(j + 1) :
49         ↪ is_filter[is_filter_idx];
50     is_filter_idx += is_filter_cond;
51 }
52

```

```

49     qdma_axis<32,0,0,0> x_pt, x_eta, x_phi, x_pdg_id;
50
51     for (unsigned int j=0; j<EV_SIZE; j+=2)
52     {
53         #pragma HLS PIPELINE II=1
54
55         x_pt.data = (pt[j + 1], pt[j]);
56         x_eta.data = (eta[j + 1], eta[j]);
57
58         x_pt.keep_all();
59         x_eta.keep_all();
60
61         s0.write(x_pt);
62         s1.write(x_eta);
63     }
64     ... // Stream other variables to the AI engine
65 }

```

Once the feature extraction is complete, the kernel creates another vector, called `is_filter`, which stores the indexes—in the range $[1, 224]$ —of the first 16 candidates that pass the low p_T cut. This corresponds to the last for loop in Listing 4.2. The computation of `is_filter` should have been included in the `w3pi` kernel but we were forced to move it here to avoid exceeding the AI engine tile program memory. The `w3pi` kernel heavily relies on this index vector and further details will be provided while discussing the AI engine kernel. The remaining part of the `mm2s` kernel streams data to the AI engine via the two 32-bit wide streaming ports. The data type of the `hls::stream` directed to the AI engine kernel is `qdma_axis`, which is dedicated to AXI4-Stream transaction that involve crossing hardware domain boundaries (PL \leftrightarrow AI engines, PL \leftrightarrow DMA controller). Since the PUPPI features fit in 16-bit integers, two values per clock cycle can be streamed out at the same time. The whole process is embedded inside a loop that iterates over the 3564 events that build up one orbit.

The `s2mm` kernel is much simpler, as it just needs to write 4×32 -bit floats—the `w3pi` kernel output—to the DDR memory of the device. Its signature is reported in Listing 4.1: the values read through the stream port `s` are directly written to memory using the 32-bit wide AXI4 master mem port. As in the previous case, the whole process is iterated for 3564 times in order to process an entire orbit. The main code block of `s2mm` is reported in Listing 4.3

Listing 4.3: `s2mm` main block.

```

1  ... // Declaring variables and interfaces
2
3  ap_int<32> mem_offset = 0;
4

```

```

5 for (unsigned int event_idx = 0; event_idx < NUM_EVENTS; event_idx++)
6 {
7     for (unsigned int i = 0; i < TRIPLET_VSIZE; i++)
8     {
9         #pragma HLS PIPELINE II=1
10
11         qdma_axis<32,0,0,0> x = s.read();
12         ap_uint<32> temp = x.get_data();
13         mem[i + mem_offset] = temp;
14     }
15
16     mem_offset += TRIPLET_VSIZE;
17 }

```

4.3.3 AI Engine Kernel

The w3pi kernel is the core compute unit of the whole design, as it leverages the key features provided by AI engines. In the current implementation, the net result is that a single AI engine tile performs all the selection steps discussed in Section 4.1 except of the first one. As previously stated, a crucial constraint that we wanted to achieve in our design was running the full analysis on a single tile, in order to maximize the possibility of scaling the computation on the whole array.

The kernel code relies on two fundamental assumptions. The first and most important one is that the number of charged hadrons candidates with $p_T > 7$ GeV (lower cut) is at most 16. The need of introducing this assumption arises from a computational inefficiency encountered during the code development on AI engines: isolation has to be calculated for charged hadron candidates exceeding the lower p_T cut w.r.t. all the other candidates (hadrons or not). Looping over the candidates with $p_T > 7$ GeV is not efficient on AI engines, since there is no hardware support for filtering a vector depending on its elements. One solution could have been defining masks using the $p_T > 7$ GeV and $\text{pid} \in \{2, 3, 4, 5\}$ condition and then looping over the vector elements corresponding to non-zero entries, but this also would have been extremely inefficient because all filtered candidates would have been spread across the 7×32 -lanes vectors needed to store the features of the 224 candidates. In essence, such approach would have implied running much more iterations on the candidates than what actually needed in order to calculate the isolation. A simple solution is collecting the indexes—in the range $[1, 224]$ —of all charged hadrons with $p_T > 7$ GeV into a single vector, which we will refer to as `is_filter`. This allows the retrieval of the filtered particle features through their index within the event, while the vectors with the features of all the particles remain unchanged and can be used to calculate the isolation. The length of `is_filter`—16 entries—has been chosen by looking at the distribution of the number of PUPPI candidates per event that satisfy the `pid` and p_T filter, shown in Figure 4.2.

The second assumption is that the `is_filter` vector is not computed by the kernel but it

is provided as input by the PL mm2s kernel, as specified in the previous paragraph. The main reason for this assumption is avoiding to exceed the program memory of the AI engine tile. Additionally, the PL is more suited than the AI engine for such task, since it provides more flexibility in terms of data slicing and reshaping.

The w3pi kernel first reads the incoming 32-bit wide streams into vectors, while keeping track of the number of charged hadrons that exceed the three p_T thresholds. If the count does not pass the filter, then the event is skipped, i.e. it will output a zero triplet. On the contrary, if the count filter is satisfied, it follows the calculation of the isolation for each candidate inside `is_filter`. This is done exploiting the vector processor of the tile. Indeed, the isolation requires computing ΔR between a charged hadron candidate belonging to `is_filter` and all the other particles, and this plays perfectly as a vector APIs use case. The isolation code block is reported in Listing 4.4. It is possible to see that the calculation of $\Delta\phi$ goes through multiple steps owing to the 2π degeneracy of the ϕ feature. Indeed, being $\phi \in [-\pi, \pi]$, we have that $\Delta\phi \in [-2\pi, 2\pi]$, but the physical values for $\Delta\phi$ actually fall within $[-\pi, \pi]$. The code is required to correct such degeneracy by doing:

$$\Delta\phi \mapsto \begin{cases} \Delta\phi - 2\pi & \text{if } \Delta\phi > \pi \\ \Delta\phi + 2\pi & \text{if } \Delta\phi < -\pi \\ \Delta\phi & \text{otherwise.} \end{cases} \quad (4.1)$$

From the code block, we also see that the candidates among the 16 pre-filtered ones that survive the isolation step are added to another vector called `is_iso_filter`, which is then iterated over for identifying the candidates that are angularly separated from each other above the the required threshold. This is done in a similar fashion w.r.t. the isolation, but using scalars instead of vectors. The reason behind this choice resides in the fact that this passage involves three nested loops—one for each particle of the triplet—and a series of conditions (e.g. ΔR threshold between two out of the three particles, avoid considering the same particle twice, skip zeroes) that need to be evaluated sequentially. The scalar core is preferable than the vector core for such type of tasks. As three angularly separated pions are identified, charge and mass conservation are also checked. Notice how for the computation of p_z a rough approximation of the sinh function, $\sinh x = x + x^3/3! + O(x^5)$, has been necessary as the AI engine libraries do not support it. The corresponding code block is reported in Listing 4.5. Eventually, the kernel streams out a 4×32 -bit integer vector containing the three indexes of the selected candidates and the correspondent invariant mass (casted from float).

Listing 4.4: Compute isolation block on AI engine.

```
1 for (int i=0; i<N_MIN; i++)
2 {
3     // Skip event flag is checked; skip zero values
```

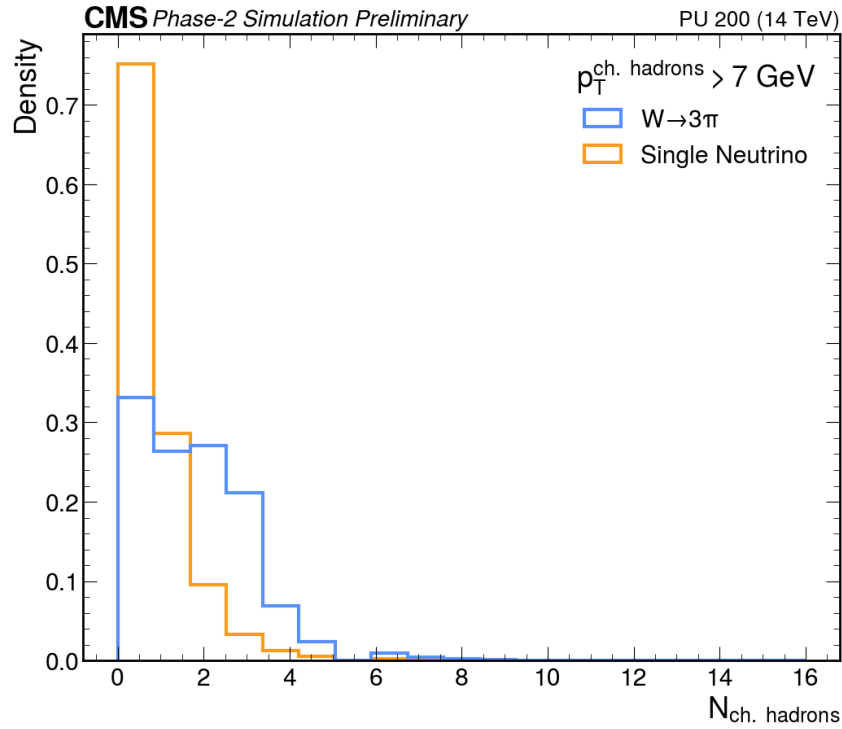


Figure 4.2: Distribution of the number of charged hadron PUPPI candidates with $p_T > 7 \text{ GeV}$ for signal and background datasets.

```

4   if (skip_event) continue;
5   if (!is_filter[i]) continue;
6
7   int16 pt_sum = 0;
8
9   // Retrieve indexes of the current candidate
10  int16 out_idx = (is_filter[i] - 1) / V_SIZE;
11  int16 in_idx = (is_filter[i] - 1) % V_SIZE;
12
13  // Iterate over the 7 32-lanes 16-bit integer vectors
14  for (int k=0; k<P_BUNCHES; k++)
15  chess_prepare_for_pipelining
16  {
17      // Delta eta
18      d_eta = aie::sub(etas[out_idx][in_idx], etas[k]);
19
20      // Delta phi
21      d_phi = aie::sub(phis[out_idx][in_idx], phis[k]);
22      d_phi_ptwopi = aie::add(d_phi, twopi_vector);
23      d_phi_mtwopi = aie::add(d_phi, mtwopi_vector);
24      is_gt_pi = aie::gt(d_phi, pi_vector);
25      is_lt_mpi = aie::lt(d_phi, mpi_vector);
26      d_phi = aie::select(d_phi, d_phi_ptwopi, is_lt_mpi);
27      d_phi = aie::select(d_phi, d_phi_mtwopi, is_gt_pi);

```

```

28
29     // Multiply-accumulate the squared values
30     // to get the squared Delta R
31     acc = aie::mul_square(d_eta);
32     acc = aie::mac_square(acc, d_phi);
33     dr2 = acc.to_vector<int32>(0);
34     dr2_float = aie::to_float(dr2, 0);
35     acc_float = aie::mul(dr2_float, F_CONV2);
36     dr2_float = acc_float.to_vector<float>(0);
37
38     // Filter in-cone particles
39     is_ge_mindr2 = aie::ge(dr2_float, MINDR2_FLOAT);
40     is_le_maxdr2 = aie::le(dr2_float, MAXDR2_FLOAT);
41     pt_cut_mask = is_ge_mindr2 & is_le_maxdr2;
42
43     // Sum pt of in-cone particles
44     pt_to_sum = aie::select(zeros_vector, pts[k], pt_cut_mask);
45     pt_sum += aie::reduce_add(pt_to_sum);
46 }
47
48 // Apply the isolation cut on the current candidate
49 float pt_cur_float = fix2float(pts[out_idx][in_idx]);
50 int16 pt_sum_thresh = float2fix(pt_cur_float * MAX_ISO);
51
52 if (pt_sum <= pt_sum_thresh)
53 {
54     pts_iso_filter[i] = pts[out_idx][in_idx];
55     etas_iso_filter[i] = etas[out_idx][in_idx];
56     phis_iso_filter[i] = phis[out_idx][in_idx];
57     pdg_ids_iso_filter[i] = pdg_ids[out_idx][in_idx];
58     is_iso_filter[i] = is_filter[i];
59 }
60 }

```

Listing 4.5: Angular separation block on AI engine.

```

1  for (int i0=0; i0<N_MIN; i0++)
2  {
3      if (skip_event) continue;
4      if (!is_iso_filter[i0]) continue;
5
6      pt_hig_pt_target0 = pts_iso_filter[i0];
7      eta_hig_pt_target0 = etas_iso_filter[i0];
8      phi_hig_pt_target0 = phis_iso_filter[i0];
9
10     for (int i1=0; i1<N_MIN; i1++)

```

```

11     {
12         if (i1 == i0) continue;
13         if (!is_iso_filter[i1]) continue;
14
15         d_eta_scalar = etas_iso_filter[i1] - eta_hig_pt_target0;
16         d_phi_scalar = phis_iso_filter[i1] - phi_hig_pt_target0;
17         d_phi_scalar = (d_phi_scalar <= PI) ? ((d_phi_scalar >= MPI) ?
            ↪ d_phi_scalar : d_phi_scalar + TWOPI) : d_phi_scalar + MTWOPI;
18
19         dr2_scalar = d_eta_scalar * d_eta_scalar + d_phi_scalar *
            ↪ d_phi_scalar;
20         dr2_float_scalar = dr2_scalar * F_CONV2;
21
22         if (dr2_float_scalar < MINDR2_ANGSEP_FLOAT) continue;
23
24         pt_hig_pt_target1 = pts_iso_filter[i1];
25         eta_hig_pt_target1 = etas_iso_filter[i1];
26         phi_hig_pt_target1 = phis_iso_filter[i1];
27
28         for (int i2=0; i2<N_MIN; i2++)
29         {
30             if (i2 == i0) continue;
31             if (i2 == i1) continue;
32             if (!is_iso_filter[i2]) continue;
33
34             d_eta_scalar = etas_iso_filter[i2] - eta_hig_pt_target1;
35             d_phi_scalar = phis_iso_filter[i2] - phi_hig_pt_target1;
36             d_phi_scalar = (d_phi_scalar <= PI) ? ((d_phi_scalar >= MPI) ?
            ↪ d_phi_scalar : d_phi_scalar + TWOPI) : d_phi_scalar +
            ↪ MTWOPI;
37
38             dr2_scalar = d_eta_scalar * d_eta_scalar + d_phi_scalar *
            ↪ d_phi_scalar;
39             dr2_float_scalar = dr2_scalar * F_CONV2;
40
41             if (dr2_float_scalar < MINDR2_ANGSEP_FLOAT) continue;
42
43             pt_hig_pt_target2 = pts_iso_filter[i2];
44             eta_hig_pt_target2 = etas_iso_filter[i2];
45             phi_hig_pt_target2 = phis_iso_filter[i2];
46
47             d_eta_scalar = pt_hig_pt_target2 - eta_hig_pt_target0;
48             d_phi_scalar = pt_hig_pt_target2 - phi_hig_pt_target0;
49             d_phi_scalar = (d_phi_scalar <= PI) ? ((d_phi_scalar >= MPI) ?
            ↪ d_phi_scalar : d_phi_scalar + TWOPI) : d_phi_scalar +
            ↪ MTWOPI;
50

```

```

51      dr2_scalar = d_eta_scalar * d_eta_scalar + d_phi_scalar *
      ↪ d_phi_scalar;
52      dr2_float_scalar = dr2_scalar * F_CONV2;
53
54      if (dr2_float_scalar < MINDR2_ANGSEP_FLOAT) continue;
55
56      charge0 = (pdg_ids_iso_filter[i0] >= 4) ?
      ↪ ((pdg_ids_iso_filter[i0] == 4) ? -1 : 1) :
      ↪ ((pdg_ids_iso_filter[i0] == 2) ? -1 : 1);
57      charge1 = (pdg_ids_iso_filter[i1] >= 4) ?
      ↪ ((pdg_ids_iso_filter[i1] == 4) ? -1 : 1) :
      ↪ ((pdg_ids_iso_filter[i1] == 2) ? -1 : 1);
58      charge2 = (pdg_ids_iso_filter[i2] >= 4) ?
      ↪ ((pdg_ids_iso_filter[i2] == 4) ? -1 : 1) :
      ↪ ((pdg_ids_iso_filter[i2] == 2) ? -1 : 1);
59
60      charge_tot = charge0 + charge1 + charge2;
61
62      if ((charge_tot != 1) & (charge_tot != -1)) continue;
63
64      px0 = pt_hig_pt_target0 * PT_CONV * aie::cos(phi_hig_pt_target0
      ↪ * F_CONV);
65      py0 = pt_hig_pt_target0 * PT_CONV * aie::sin(phi_hig_pt_target0
      ↪ * F_CONV);
66      x = eta_hig_pt_target0 * F_CONV;
67      sinh = x + ((x * x * x) / 6);
68      pz0 = pt_hig_pt_target0 * PT_CONV * sinh;
69      e0 = aie::sqrt(px0 * px0 + py0 * py0 + pz0 * pz0 + MASS_P *
      ↪ MASS_P);
70
71      px1 = pt_hig_pt_target1 * PT_CONV * aie::cos(phi_hig_pt_target1
      ↪ * F_CONV);
72      py1 = pt_hig_pt_target1 * PT_CONV * aie::sin(phi_hig_pt_target1
      ↪ * F_CONV);
73      x = eta_hig_pt_target1 * F_CONV;
74      sinh = x + ((x * x * x) / 6);
75      pz1 = pt_hig_pt_target1 * PT_CONV * sinh;
76      e1 = aie::sqrt(px1 * px1 + py1 * py1 + pz1 * pz1 + MASS_P *
      ↪ MASS_P);
77
78      px2 = pt_hig_pt_target2 * PT_CONV * aie::cos(phi_hig_pt_target2
      ↪ * F_CONV);
79      py2 = pt_hig_pt_target2 * PT_CONV * aie::sin(phi_hig_pt_target2
      ↪ * F_CONV);
80      x = eta_hig_pt_target2 * F_CONV;
81      sinh = x + ((x * x * x) / 6);
82      pz2 = pt_hig_pt_target2 * PT_CONV * sinh;

```



```

83         e2 = aie::sqrt(px2 * px2 + py2 * py2 + pz2 * pz2 + MASS_P *
      ↪ MASS_P);
84
85         px_tot = px0 + px1 + px2;
86         py_tot = py0 + py1 + py2;
87         pz_tot = pz0 + pz1 + pz2;
88         e_tot = e0 + e1 + e2;
89
90         invariant_mass = aie::sqrt(e_tot * e_tot - px_tot * px_tot -
      ↪ py_tot * py_tot - pz_tot * pz_tot);
91
92         if ((invariant_mass < MIN_MASS) | (invariant_mass > MAX_MASS))
      ↪ continue;
93
94         triplet[0] = is_iso_filter[i0] - 1;
95         triplet[1] = is_iso_filter[i1] - 1;
96         triplet[2] = is_iso_filter[i2] - 1;
97         triplet[3] = invariant_mass;
98
99         exitLoop = true;
100        break;
101    }
102    if (exitLoop) break;
103 }
104 if (exitLoop) break;
105 }

```

4.3.4 Host Program

The host program runs on the host CPU and its role is to orchestrate the interplay between memory, PL kernels and AI engine kernels. To do so, it uses XRT APIs that allow for transferring data and provide instructions to reconfigure the dynamic partition of the VCK5000 deployment platform.

The host program follows a standard structure, whose summarized version is reported in Listing 4.6. As a first step, the host program loads the .xclbin file into the device. This is a key step since it reconfigures the dynamic partition at runtime. The `xclbin_uuid` object that is returned after the reconfiguration allows the instantiation of PL kernels on the device. Then, buffer objects are created for data transfer between host and device and vice-versa. Notice how the buffer constructor takes as input `mm2s.group_id(0)`, which corresponds to the memory bank associated with the first argument of the kernel. The following step is to fill the buffers with data and allocate them on the device: this is performed using the `write` and `sync` methods of buffer objects. Once buffers are ready, PL kernels are started and the program waits for `s2mm` to finish executing, i.e. when it has received `w3pi` outputs from all the events in the current orbit. It is possible to see that the `w3pi` kernel running on the AI engine

does not appear at all in the host code. The reason is that once the AI engine graph is loaded into the array, the execution starts automatically as soon as the kernel interfaces find enough available data. Eventually, the output buffer containing the result is synchronized and copied to host memory.

In our development setup, the input data is loaded directly from the binary file and the result of the computation is written to disk as a csv file, ready to be processed for the analysis.

Listing 4.6: Structure of host code.

```

1 // get .xclbin file as a command line argument
2 char* xclbinFile = argv[1];
3
4 // detect device and reconfigure it
5 auto device = xrt::device(0);
6 auto xclbin_uuid = device.load_xclbin(xclbinFile);
7
8 // create kernel instances
9 auto mm2s = xrt::kernel(device, xclbin_uuid, "mm2s");
10 auto s2mm = xrt::kernel(device, xclbin_uuid, "s2mm");
11
12 // allocate buffer for the input of the two mm2s kernels
13 auto events_buf = xrt::bo(device,
14                             NUM_EVENTS * EV_SIZE * sizeof(int64_t),
15                             xrt::bo::flags::normal,
16                             mm2s.group_id(0));
17
18 auto triplets_buf = xrt::bo(device,
19                             NUM_EVENTS * TRIPLET_VSIZE * sizeof(ap_int<32>),
20                             xrt::bo::flags::normal,
21                             s2mm.group_id(0));
22
23 // load dataset
24 std::ifstream bin_file("/path/to/data.dump", std::ios::binary);
25
26 // set the start position of the stream to the first event of the orbit
27 std::streampos start = start_event_idx * EV_SIZE * sizeof(ap_int<64>);
28 bin_file.seekg(start, std::ios::beg);
29
30 // fill array with data of an entire orbit
31 ap_int<64 * BLOCK_SIZE> mem[NUM_EVENTS * NUM_BLOCKS];
32 bin_file.read(reinterpret_cast<char*>(mem),
33              NUM_EVENTS * EV_SIZE * sizeof(ap_int<64>));
34 bin_file.close();
35
36 // write and sync the array to the device
37 events_buf.write(mem);

```

```

38 events_buf.sync(XCL_BO_SYNC_BO_TO_DEVICE);
39
40 // create run instances of the kernels
41 auto mm2s_run = mm2s(events_buf, nullptr, nullptr);
42 auto s2mm_run = s2mm(triplets_buf, nullptr);
43
44 // wait for kernels to finish execution
45 // AI engine kernel is running under the hood
46 mm2s_run.wait();
47 s2mm_run.wait();
48
49 // read output buffer content from device to host
50 int32_t triplets[NUM_EVENTS * TRIPLET_VSIZE];
51 triplets_buf.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
52 triplets_buf.read(triplets);

```

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined
+ mm2s	-	0.00	566	1.572e+03	-	567	-	no
+ mm2s_Pipeline_VITIS_LOOP_34_3	-	0.00	17	47.209	-	17	-	no
o VITIS_LOOP_34_3	-	2.03	15	41.655	3	1	14	yes
+ mm2s_Pipeline_VITIS_LOOP_67_5	-	0.13	228	633.156	-	228	-	no
o VITIS_LOOP_67_5	-	2.03	226	627.602	4	1	224	yes
+ mm2s_Pipeline_VITIS_LOOP_78_6	-	0.91	114	316.578	-	114	-	no
o VITIS_LOOP_78_6	-	2.03	112	311.024	2	1	112	yes
+ mm2s_Pipeline_VITIS_LOOP_92_7	-	0.91	114	316.578	-	114	-	no
o VITIS_LOOP_92_7	-	2.03	112	311.024	2	1	112	yes
+ mm2s_Pipeline_VITIS_LOOP_108_8	-	0.98	10	27.770	-	10	-	no
o VITIS_LOOP_108_8	-	2.03	8	22.216	2	1	8	yes

Figure 4.3: mm2s kernel performance estimates.

4.3.5 Compiling, Linking and Packaging the Design

As all the components were ready, the Vitis tools[54] have been used to get the final .xclbin container. The compilation report of the mm2s kernel, visible in Figure 4.3, shows how all the HLS pragmas have been correctly enforced. Moreover, the total latency of the kernel is estimated to be approximately $1.5 \mu\text{s}$. The aiecompiler generated the AI engine array floorplan reported in Figure 4.4 (d), in which it is possible to see the two ingoing streams and the single outgoing stream passing through the same tile interconnect module and AI engine interface tile.

Listing 4.7: system.cfg file for the linking step.

```

[connectivity]
nk = mm2s:1:mm2s

```

```

nk = s2mm:1:s2mm

slr = mm2s:SLR0
slr = s2mm:SLR0

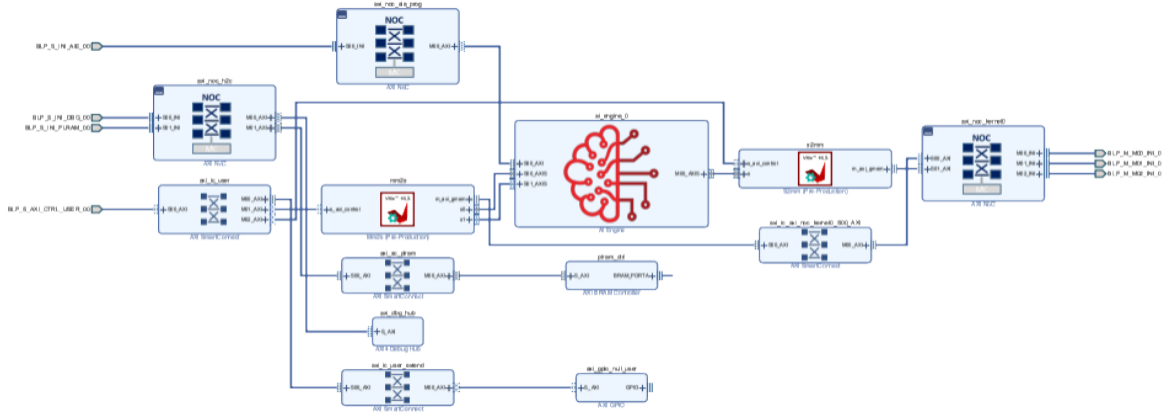
sp = mm2s.m_axi_gmem:MC_NOC0
sp = s2mm.m_axi_gmem:MC_NOC0

stream_connect = mm2s.s0:ai_engine_0.in0
stream_connect = mm2s.s1:ai_engine_0.in1
stream_connect = ai_engine_0.out:s2mm.s

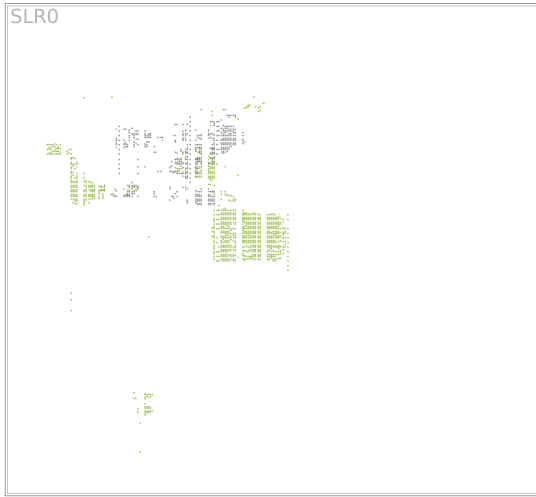
[vivado]
# use following line to improve the hw_emu running speed affected by platform
prop=filesset.sim_1.xsim.elaborate.xelab.more_options={-override_timeprecision
↪ -timescale=1ns/1ps}

```

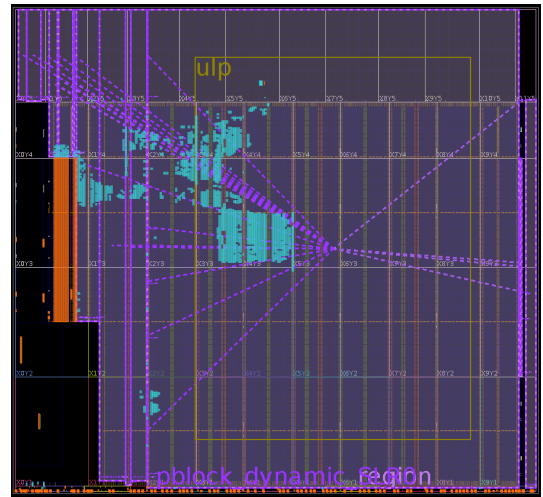
The linking step took approximately one hour to run all the Vivado synthesis, place and route tools. The v++ compiler has been provided the `system.cfg` file reported in Listing 4.7: the connectivity options instruct the compiler to create one instance of both the `mm2s` and `s2mm` kernels and to place them into the available SLR of the platform. The platform used for this design has only a single SLR is available (SLR0). Notice how the AI engine kernel does not need to be instantiated here. Indeed, its system ports are already hardened inside the board, so just the connections have to be specified. Then the system port (`sp`) option specifies the memory resources connected to the memory interfaces of the instantiated kernels. As previously discussed, both `mm2s` and `s2mm` kernels have one global memory interface (`m_axi_gmem`) which is connected to the NoC memory controller (`MC_NOC0`) available in the platform. The `streaming_connect` option instead specifies the connections that do not involve memory resources. This is the case of the input/output ports of the AI engine kernel, whose correspondent graph is labeled as `ai_engine_0` by default. The `vivado` options include advanced configurations for managing the low-level details of design place and route phase. The reported one provides a speed up in the hardware emulation flow, which can be useful for debugging. Figure 4.4 (a) shows the interface view of design's block diagram: it is possible to see the AI engine interfaces IP and the two Vitis HLS IPs connected to it. The PL floorplan is reported in Figure 4.4 (b): the interconnections are marked in light green, while the kernel resources are marked in gray. In Figure 4.4 (c) it is possible to see also the fixed region in orange, while the full DFX region is shaded in purple. The generated system diagram is visible in Figure 4.4 (e), together with the resources utilization next to each PL kernel, which turns out to be minimal except for the BRAMs (3.21%).



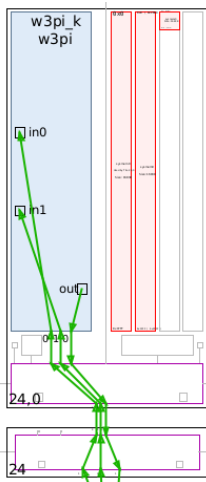
(a)



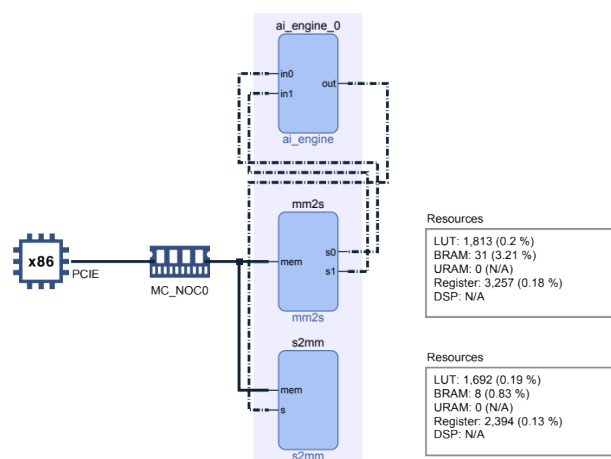
(b)



(c)



(d)



(e)

Figure 4.4: Floorplans and diagrams obtained after running the Vitis tools.

Orbit	Allocation Time (ms)	Execution Time (ms)	Readout Time (ms)
0	1.270	7.080	0.0275
1	0.957	7.120	0.0234
2	0.959	6.970	0.0219
3	0.955	6.950	0.0223
4	0.955	6.970	0.0216
5	0.956	6.920	0.0220
6	0.956	7.030	0.0218
7	0.958	7.090	0.0237
8	0.955	7.180	0.0230
9	0.957	7.110	0.0215
10	0.956	7.150	0.0214
11	0.975	7.130	0.0223
12	0.980	7.020	0.0295
13	0.957	7.030	0.0303

Table 4.6: Processing time statistics for each orbit (background dataset).

Orbit	Allocation Time (ms)	Execution Time (ms)	Readout Time (ms)
0	0.985	14.400	0.0325
1	0.977	14.200	0.0299
2	0.976	13.800	0.0277
3	0.978	14.100	0.0293
4	0.981	13.800	0.0287
5	0.981	14.100	0.0291
6	0.979	14.300	0.0313
7	0.975	13.900	0.0259
8	0.976	14.200	0.0293
9	0.977	14.000	0.0290
10	0.976	14.100	0.0293
11	0.974	14.100	0.0291
12	0.976	14.000	0.0292
13	0.976	13.700	0.0282

Table 4.7: Processing time statistics for each orbit (signal dataset).

Device	ms/orbit	Resources/CU	CUs @ 31.5 MHz	Devices/CUs
VCK5000	7	0.25%	62	16% of one board
Alveo U55C	4.6	2.3%	41	94% of one board
NVIDIA L4	1.5	50%	13	7 GPUs

Table 4.8: Rough comparison of hardware-accelerated implementation of the $W \rightarrow 3\pi$ selection.

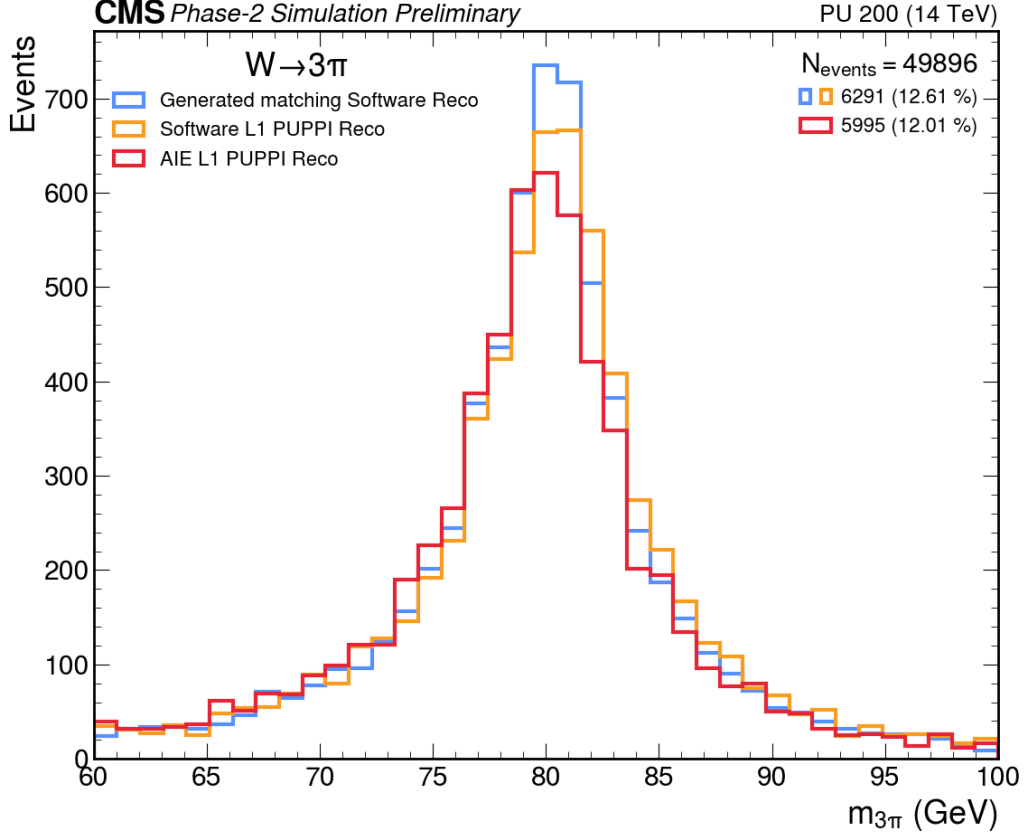


Figure 4.5: Invariant mass distribution of AI-engine reconstructed triplets, compared with the invariant mass of software reconstructed ones.

4.3.6 Physics Performance

The accelerated selection has been run over the 14 available orbits of the signal dataset and the first 14 orbits of the background dataset, in order to collect the same amount of statistics. The used cuts' thresholds are those reported in Section 4.1. Figure 4.5 shows the results by comparing the generated and the reconstructed invariant mass distribution for the events selected through the algorithm both on software and on our AI engine implementation. It is possible to see that the AI engine implementation misses a 5% of events w.r.t. the software one: despite apparently small, this discrepancy must be addressed since perfectly identical results are expected. As a first improvement, a look-up table for the sinh function could be implemented in order to replace the current rough approximation.

4.3.7 Computational Performance

The computational performance of our design is shown in Table 4.6 for the background dataset and in Table 4.7 for the signal dataset: on background, the selection runs on average in ~ 7 ms/orbit, which corresponds to $\sim 1.96 \mu\text{s}/\text{event}$, while on signal it runs on average in ~ 14 ms/orbit, which corresponds to $\sim 3.93 \mu\text{s}/\text{event}$. As expected, the selection runs much slower

on signal. It is important to specify that the measured execution time involves all three kernels, not only the AI engine one. Measuring the AI engine latency alone on a full orbit is not possible on the host code, because of the fact that the `w3pi` kernel runs automatically without being called by the host. The allocation and readout time refer to the time taken to transfer one orbit data from host to device DDR and the time taken to transfer the result the other way around respectively. Despite being an important factor to consider, we did not delve into optimizing it. A in-depth study of the host/device memory management will be needed once the current design will be integrated in the Phase-2 demonstrator.

A crucial estimation that can be done with the measured execution times is how our current system scales at the full bunch crossing rate. It is important to specify here that the maximum number of filled bunches foreseen for Phase 2 is 2808 out of the 3564 possible, thus providing an effective BX rate of 31.5 MHz[33]. Considering an average of 7 ms/orbit as execution time on background, we have that the number of Compute Units (CUs) to work at the 31.5 MHz rate is:

$$n_{CU} = \frac{7 \cdot 10^{-3} \frac{s}{orbit}}{3564 \frac{events}{orbit} \times (\frac{1}{31.5 \cdot 10^6}) \frac{s}{event}} \simeq 62 \text{ compute units}$$

which correspond roughly to 16% of the total AI engine array resources. This means that it should be theoretically feasible to have at least one or two online analyses—requiring a similar compute demand—working at 31.5 MHz in a single VCK5000 board. We stress that this is only a rough estimate, and the true feasibility study requires an accurate evaluation of the throughput between the host and the device, as well as between the PL and the AI engine array. This result looks promising for future developments of the Phase-2 scouting demonstrator, especially working towards scaling it up to 31.5 MHz. Accelerating selection algorithms on hardware provides a deterministic latency in a way that is not feasible with CPUs and this could help to implement a more resource-efficient system.

It is possible to compare VCK5000 results with the ones obtained by analogous designs running on Alveo U55C FPGA² and NVIDIA L4 GPU³, as shown in Table 4.8. The considered resources are AI engine tile usage per compute unit, LUTs per compute unit and `nvidia-smi`-estimated GPU occupancy per compute unit respectively. Despite achieving the highest latency per orbit, the VCK5000 seems to outperform the other accelerators in terms of resources per compute unit, which is a key factor to ensure scalability. Also in this case we specify that the comparison is based on rough estimates so it should be taken as a preliminary result. Moreover, one has to take into account the specific design goals originally foreseen for each type of accelerator: VCK5000 and NVIDIA L4 have been developed and optimized to achieve outstanding performance on high-throughput AI inference task, while Alveo U55C is designed to enhance servers employed in large big-data computing clusters. With the rising interest for machine-learning based physics algorithms, devices like VCK5000 and NVIDIA L4 could

²Results provided by [Leah-Louisa Sieder](#) (Technische Universität Dresden)

³Results provided by [Lukasz Artur Michalsky](#) (Wrocław University of Science and Technology).

be more suited for the evolving scenarios in the following years.

Chapter 5

Conclusions

This thesis presented the design, implementation, and benchmarking of an accelerated on-line selection algorithm for the CMS Phase-2 Level-1 Trigger scouting system. The goal was to explore the feasibility and benefits of deploying real-time physics selection algorithms on heterogeneous accelerator platforms, with a specific focus on the AMD Versal Adaptive SoC.

The L1T scouting system is being developed to enhance the CMS experiment's ability to collect and process events that would otherwise be discarded by traditional trigger systems, especially under the extreme conditions foreseen at the High-Luminosity LHC. By recording reduced but meaningful information from the trigger level at the full 40 MHz bunch crossing rate, the scouting system opens new opportunities for real-time analyses of rare processes. Implementing this system requires an efficient and scalable computing farm, capable of running multiple online selection algorithms at once. To address this, the thesis investigated the use of the AMD VCK5000 Development Card, featuring a VC1902 Versal chip that combines scalar processors, FPGA fabric, and AI engines. A complete use case was developed for the rare $W \rightarrow 3\pi$ decay channel, which serves as a testbench for evaluating both physics performance and computational efficiency of the AI-engine implementation. The cut-based algorithm was fully implemented on hardware, with a design involving PL kernels for data unpacking and streaming, and a dedicated AI engine kernel running the physics selection. The host program, written using XRT APIs, orchestrates the execution and manages data transfers between the host system and the accelerator.

Benchmarking results show that the system achieves a latency of approximately $1.96 \mu\text{s}$ per event on background dataset, implying that the full algorithm could run at 31.5 MHz using only $\sim 16\%$ of the AI engine resources on a single VCK5000 board. This demonstrates a promising level of scalability and opens the path for deploying multiple analyses in parallel within the same hardware platform. Physics performance was also satisfactory, with a small efficiency loss ($\sim 5\%$) compared to the software reference. Future work will focus on further optimizations at both the PL level and the AI engine level. One initial step will be the implementation of variable event sizes to better adapt to realistic data taking scenarios. Additionally, developing a complete architecture with a sufficient number of compute units to sustain the full bunch crossing rate is a crucial step toward bringing the current prototype

closer to an operational deployment. Finally, promising directions also include leveraging AI engines for machine-learning-based selections and jet tagging algorithms, further expanding the scope and capabilities of hardware-accelerated online analysis.

Bibliography

- [1] Oliver Sim Brüning et al. *LHC Design Report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2004. DOI: [10.5170/CERN-2004-003-V-1](https://doi.org/10.5170/CERN-2004-003-V-1). URL: <https://cds.cern.ch/record/782076>.
- [2] Ewa Lopienska. “The CERN accelerator complex, layout in 2022. Complexe des accélérateurs du CERN en janvier 2022”. In: (2022). General Photo. URL: <https://cds.cern.ch/record/2800984>.
- [3] ATLAS Collaboration. “The ATLAS Experiment at the CERN Large Hadron Collider”. In: *JINST* 3 (2008). Also published by CERN Geneva in 2010, S08003. DOI: [10.1088/1748-0221/3/08/S08003](https://doi.org/10.1088/1748-0221/3/08/S08003). URL: <https://cds.cern.ch/record/1129811>.
- [4] CMS Collaboration. “The CMS experiment at the CERN LHC. The Compact Muon Solenoid experiment”. In: *JINST* 3 (2008). Also published by CERN Geneva in 2010, S08004. DOI: [10.1088/1748-0221/3/08/S08004](https://doi.org/10.1088/1748-0221/3/08/S08004). URL: <https://cds.cern.ch/record/1129810>.
- [5] ALICE Collaboration. “The ALICE experiment at the CERN LHC. A Large Ion Collider Experiment”. In: *JINST* 3 (2008). Also published by CERN Geneva in 2010, S08002. DOI: [10.1088/1748-0221/3/08/S08002](https://doi.org/10.1088/1748-0221/3/08/S08002). URL: <https://cds.cern.ch/record/1129812>.
- [6] LHCb Collaboration. “The LHCb Detector at the LHC”. In: *JINST* 3 (2008). Also published by CERN Geneva in 2010, S08005. DOI: [10.1088/1748-0221/3/08/S08005](https://doi.org/10.1088/1748-0221/3/08/S08005). URL: <https://cds.cern.ch/record/1129809>.
- [7] Tai Sakuma and Thomas McCauley. “Detector and Event Visualization with SketchUp at the CMS Experiment”. In: *Journal of Physics: Conference Series* 513.2 (June 2014), p. 022032. DOI: [10.1088/1742-6596/513/2/022032](https://doi.org/10.1088/1742-6596/513/2/022032). URL: <https://dx.doi.org/10.1088/1742-6596/513/2/022032>.
- [8] CMS Collaboration. *The CMS tracker system project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: <https://cds.cern.ch/record/368412>.
- [9] CMS Collaboration. *The CMS electromagnetic calorimeter project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: <https://cds.cern.ch/record/349375>.

- [10] CMS Collaboration. *The CMS hadron calorimeter project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: <https://cds.cern.ch/record/357153>.
- [11] CMS Collaboration. *The CMS muon project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: <https://cds.cern.ch/record/343814>.
- [12] CMS Collaboration. *CMS TriDAS project: Technical Design Report, Volume 1: The Trigger Systems*. Technical design report. CMS. CERN, 2000. URL: <https://cds.cern.ch/record/706847>.
- [13] Sergio Cittolin, Attila Rácz, and Paris Sphicas. *CMS The TriDAS Project: Technical Design Report, Volume 2: Data Acquisition and High-Level Trigger. CMS trigger and data-acquisition project*. Technical design report. CMS. Geneva: CERN, 2002. URL: <https://cds.cern.ch/record/578006>.
- [14] CMS Collaboration. *CMS Technical Design Report for the Level-1 Trigger Upgrade*. Tech. rep. Additional contacts: Jeffrey Spalding, Fermilab, Jeffrey.Spalding@cern.ch Didier Contardo, Universite Claude Bernard-Lyon I, didier.claude.contardo@cern.ch. CERN, 2013. URL: <https://cds.cern.ch/record/1556311>.
- [15] CMS Collaboration. “The CMS Level-1 muon triggers for the LHC Run II”. In: *PoS ICHEP2018* (2019), p. 918. DOI: [10.22323/1.340.0918](https://doi.org/10.22323/1.340.0918).
- [16] CMS Collaboration. “Run 2 upgrades to the CMS Level-1 calorimeter trigger”. In: *Journal of Instrumentation* 11.01 (Jan. 2016), pp. C01051–C01051. ISSN: 1748-0221. DOI: [10.1088/1748-0221/11/01/c01051](https://doi.org/10.1088/1748-0221/11/01/c01051). URL: <http://dx.doi.org/10.1088/1748-0221/11/01/C01051>.
- [17] J. Wittmann et al. “The upgrade of the CMS Global Trigger”. In: *Journal of Instrumentation* 11.02 (Feb. 2016), p. C02029. DOI: [10.1088/1748-0221/11/02/C02029](https://doi.org/10.1088/1748-0221/11/02/C02029). URL: <https://dx.doi.org/10.1088/1748-0221/11/02/C02029>.
- [18] CMS Collaboration. “The CMS trigger system”. In: *Journal of Instrumentation* 12.01 (Jan. 2017), P01020–P01020. ISSN: 1748-0221. DOI: [10.1088/1748-0221/12/01/p01020](https://doi.org/10.1088/1748-0221/12/01/p01020). URL: <http://dx.doi.org/10.1088/1748-0221/12/01/P01020>.
- [19] CMS Collaboration. “The CMS Particle Flow Algorithm”. In: *EPJ Web Conf.* 191 (2018), p. 02016. DOI: [10.1051/epjconf/201819102016](https://doi.org/10.1051/epjconf/201819102016). URL: <https://cds.cern.ch/record/2678077>.
- [20] CMS Collaboration. “Particle-flow reconstruction and global event description with the CMS detector”. In: *Journal of Instrumentation* 12.10 (Oct. 2017), P10003–P10003. ISSN: 1748-0221. DOI: [10.1088/1748-0221/12/10/p10003](https://doi.org/10.1088/1748-0221/12/10/p10003). URL: <http://dx.doi.org/10.1088/1748-0221/12/10/P10003>.
- [21] Wolfgang Adam et al. *Track Reconstruction in the CMS tracker*. Tech. rep. Geneva: CERN, 2006. URL: <https://cds.cern.ch/record/934067>.

- [22] CMS Collaboration. “Performance of the CMS muon detector and muon reconstruction with proton-proton collisions at $s=13$ TeV”. In: *Journal of Instrumentation* 13.06 (June 2018), P06015–P06015. ISSN: 1748-0221. DOI: [10.1088/1748-0221/13/06/p06015](https://doi.org/10.1088/1748-0221/13/06/p06015). URL: <http://dx.doi.org/10.1088/1748-0221/13/06/P06015>.
- [23] CMS Collaboration. “Performance of the CMS muon detector and muon reconstruction with proton-proton collisions at $s=13$ TeV”. In: *Journal of Instrumentation* 13.06 (June 2018), P06015–P06015. ISSN: 1748-0221. DOI: [10.1088/1748-0221/13/06/p06015](https://doi.org/10.1088/1748-0221/13/06/p06015). URL: <http://dx.doi.org/10.1088/1748-0221/13/06/P06015>.
- [24] CMS Collaboration. “Electron and photon reconstruction and identification with the CMS experiment at the CERN LHC”. In: *Journal of Instrumentation* 16.05 (May 2021), P05014. ISSN: 1748-0221. DOI: [10.1088/1748-0221/16/05/p05014](https://doi.org/10.1088/1748-0221/16/05/p05014). URL: <http://dx.doi.org/10.1088/1748-0221/16/05/P05014>.
- [25] Matteo Cacciari, Gavin P Salam, and Gregory Soyez. “The anti-ktjet clustering algorithm”. In: *Journal of High Energy Physics* 2008.04 (Apr. 2008), pp. 063–063. ISSN: 1029-8479. DOI: [10.1088/1126-6708/2008/04/063](https://doi.org/10.1088/1126-6708/2008/04/063). URL: <http://dx.doi.org/10.1088/1126-6708/2008/04/063>.
- [26] O. Aberle et al. *High-Luminosity Large Hadron Collider (HL-LHC): Technical design report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2020. DOI: [10.23731/CYRM-2020-0010](https://doi.org/10.23731/CYRM-2020-0010). URL: <https://cds.cern.ch/record/2749422>.
- [27] CMS Collaboration. *Technical Proposal for the Phase-II Upgrade of the CMS Detector*. Tech. rep. Upgrade Project Leader Deputies: Lucia Silvestris (INFN-Bari), Jeremy Mans (University of Minnesota) Additional contacts: Lucia.Silvestris@cern.ch, Jeremy.Mans@cern.ch. Geneva: CERN, 2015. DOI: [10.17181/CERN.VU8I.D59J](https://doi.org/10.17181/CERN.VU8I.D59J). URL: <https://cds.cern.ch/record/2020886>.
- [28] CMS Collaboration. *The Phase-2 Upgrade of the CMS Tracker*. Tech. rep. Geneva: CERN, 2017. DOI: [10.17181/CERN.QZ28.FLHW](https://doi.org/10.17181/CERN.QZ28.FLHW). URL: <https://cds.cern.ch/record/2272264>.
- [29] CMS Collaboration. *A MIP Timing Detector for the CMS Phase-2 Upgrade*. Tech. rep. Geneva: CERN, 2019. URL: <https://cds.cern.ch/record/2667167>.
- [30] CMS Collaboration. *The Phase-2 Upgrade of the CMS Barrel Calorimeters*. Tech. rep. This is the final version, approved by the LHCC. Geneva: CERN, 2017. URL: <https://cds.cern.ch/record/2283187>.
- [31] CMS Collaboration. *The Phase-2 Upgrade of the CMS Endcap Calorimeter*. Tech. rep. Geneva: CERN, 2017. DOI: [10.17181/CERN.IV8M.1JY2](https://doi.org/10.17181/CERN.IV8M.1JY2). URL: <https://cds.cern.ch/record/2293646>.
- [32] CMS Collaboration. *The Phase-2 Upgrade of the CMS Muon Detectors*. Tech. rep. This is the final version, approved by the LHCC. Geneva: CERN, 2017. URL: <https://cds.cern.ch/record/2283189>.

- [33] CMS Collaboration. *The Phase-2 Upgrade of the CMS Level-1 Trigger*. Tech. rep. Final version. Geneva: CERN, 2020. URL: <https://cds.cern.ch/record/2714892>.
- [34] CMS Collaboration. “Design of the CMS calorimeter trigger upgrade from Phase I to Phase II of the LHC”. In: *Journal of Physics: Conference Series* 1162.1 (Jan. 2019), p. 012040. DOI: [10.1088/1742-6596/1162/1/012040](https://doi.org/10.1088/1742-6596/1162/1/012040). URL: <https://dx.doi.org/10.1088/1742-6596/1162/1/012040>.
- [35] Christopher Edward Brown. “Fast Machine Learning in the CMS Level-1 Trigger for the High-Luminosity LHC”. Presented 25 Jul 2023. Imperial College London, 2023. URL: <https://cds.cern.ch/record/2875830>.
- [36] Giovanni Petrucciani. *Particle Flow reconstruction in the CMS Level-1 trigger for the HL-LHC. Particle Flow reconstruction in the Level-1 trigger at CMS for the HL-LHC*. Tech. rep. Geneva: CERN, 2019. DOI: [10.1051/epjconf/201921401019](https://doi.org/10.1051/epjconf/201921401019). URL: <https://cds.cern.ch/record/2650974>.
- [37] C. Herwig and on behalf of the CMS collaboration. “Particle flow reconstruction for the CMS Phase-II Level-1 Trigger”. In: *Journal of Instrumentation* 18.01 (Jan. 2023), p. C01037. DOI: [10.1088/1748-0221/18/01/C01037](https://doi.org/10.1088/1748-0221/18/01/C01037). URL: <https://dx.doi.org/10.1088/1748-0221/18/01/C01037>.
- [38] Daniele Bertolini et al. “Pileup Per Particle Identification”. In: *JHEP* 10 (2014). Comments: v1 - 23 pages, 10 figures, p. 59. DOI: [10.1007/JHEP10\(2014\)059](https://doi.org/10.1007/JHEP10(2014)059). arXiv: [1407.6013](https://arxiv.org/abs/1407.6013). URL: <https://cds.cern.ch/record/1745357>.
- [39] Sioni Summers, Ioannis Bestintzanos, and Giovanni Petrucciani. *Reconstructing jets in the Phase-2 upgrade of the CMS Level-1 Trigger with a seeded cone algorithm*. 2023. arXiv: [2310.08062](https://arxiv.org/abs/2310.08062) [hep-ex]. URL: <https://arxiv.org/abs/2310.08062>.
- [40] G. Bortolato et al. “Architecture and prototype of the CMS Global Level-1 Trigger for Phase-2”. In: *Journal of Instrumentation* 18.01 (Jan. 2023), p. C01034. DOI: [10.1088/1748-0221/18/01/C01034](https://doi.org/10.1088/1748-0221/18/01/C01034). URL: <https://dx.doi.org/10.1088/1748-0221/18/01/C01034>.
- [41] Matteo Migliorini. “40MHz scouting at the CMS experiment”. Presented 16 Dec 2024. Padua U., 2024. URL: <https://cds.cern.ch/record/2920832>.
- [42] CMS Collaboration. *Enriching the physics program of the CMS experiment via data scouting and data parking*. 2024. arXiv: [2403.16134](https://arxiv.org/abs/2403.16134) [hep-ex]. URL: <https://arxiv.org/abs/2403.16134>.
- [43] D. S. Rabady et al. “A 40 MHz Level-1 trigger scouting system for the CMS Phase-2 upgrade”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 1047 (2023), p. 167805. ISSN: 0168-9002. DOI: <https://doi.org/10.1016/j.nima.2022.167805>. URL: <https://www.sciencedirect.com/science/article/pii/S016890022201097X>.

- [44] T. O. James et al. *The Level 1 Scouting system of the CMS experiment*. Tech. rep. Geneva: CERN, 2023. URL: <https://cds.cern.ch/record/2852916>.
- [45] CMS Collaboration. “Level-1 trigger scouting in Phase-2”. In: (2024). URL: <https://cds.cern.ch/record/2916191>.
- [46] Rocco Ardino. “Search for rare boson decays with the CMS detector at LHC and the CMS Level-1 trigger Data Scouting”. Presented 16 Dec 2024. Padua U., 2024. URL: <https://cds.cern.ch/record/2921503>.
- [47] CMS Collaboration. *The Phase-2 Upgrade of the CMS Data Acquisition and High Level Trigger*. Tech. rep. This is the final version of the document, approved by the LHCC. Geneva: CERN, 2021. URL: <https://cds.cern.ch/record/2759072>.
- [48] Inc. Xilinx. “Versal: The First Adaptive Compute Acceleration Platform (ACAP)”. In: (2020).
- [49] Inc. (AMD) Advanced Micro Devices. “AI Engines and Their Applications”. In: (2022).
- [50] Inc. (AMD) Advanced Micro Devices. *AM009 Versal Adaptive SoC AI Engine Architecture Manual*. (v1.3) August 18, 2023. 2023.
- [51] Inc. (AMD) Advanced Micro Devices. *UG1079 AI Engine Kernel Coding Best Practices Guide*. (v2022.1) May 25, 2022. 2022.
- [52] Inc. (AMD) Advanced Micro Devices. *UG1076 Versal ACAP AI Engine Programming Environment*. (v2022.1) May 25, 2022. 2022.
- [53] Inc. (AMD) Advanced Micro Devices. *AI Engine API User Guide*. https://www.xilinx.com/htmldocs/xilinx2022_1/aiengine_api/aie_api/doc/namespaceaie.html [Visited: (January 2025)].
- [54] Inc. (AMD) Advanced Micro Devices. *UG1393 Vitis Unified Software Platform Documentation Application Acceleration Development*. (v2022.2) October 19, 2022. 2022.
- [55] Inc. (AMD) Advanced Micro Devices. *Xilinx® Runtime (XRT) 2022.1*. <https://xilinx.github.io/XRT/2022.1/html/index.html> [Visited: (January 2025)].
- [56] Inc. (AMD) Advanced Micro Devices. *Alveo Versal Vitis Platforms*. <https://xilinx.github.io/Alveo-Versal-Platforms/alveoversalplatforms/build/html/index.html> [Visited: (January 2025)].
- [57] Inc. (AMD) Advanced Micro Devices. *AMD VCK5000 Versal™ Development Card*. <https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/vck5000.html> [Visited: (January 2025)].
- [58] CMS Collaboration. “Search for W Boson Decays to Three Charged Pions”. In: *Physical Review Letters* 122.15 (Apr. 2019). ISSN: 1079-7114. DOI: [10.1103/PhysRevLett.122.151802](https://doi.org/10.1103/PhysRevLett.122.151802). URL: <http://dx.doi.org/10.1103/PhysRevLett.122.151802>.

- [59] Pietro Cappelli. “On the measurement of W to 3 π with the Phase 2 L1 scouting system at CMS”. Padua U., 2023. URL: <https://hdl.handle.net/20.500.12608/59324>.
- [60] Inc. (AMD) Advanced Micro Devices. *UG1399 Vitis High-Level Synthesis User Guide*. (v2022.1) June 7, 2022. 2022.

Acronyms

ACAP	Adaptive Compute Acceleration Platform
AGU	Address Generator Unit
AI	Adaptable Intelligent
API	Application Programming Interface
AXI4	Advanced eXtensible Interface 4
BMFT	Barrel Muon Track Finder
BRAM	Block RAM
BSM	Beyond Standard Model
BX	bunch crossing
CERN	European Organization for Nuclear Research
CLB	Configurable Logic Block
CMC	Card Management Controller
CMS	Compact Muon Solenoid
CPU	Central Processing Unit
CU	Compute Unit
CSC	Cathode Strip Chamber
CT	Correlator Trigger
DAQ	Data Acquisition
DDR	Double Data Rate
DFX	Dynamic Function Exchange
DLP	Data Level Parallelism
DMA	Direct Memory Access
DQM	Data Quality Monitoring
DSP	Digital Signal Processor

DT	drift tube
ECAL	Electromagnetic Calorimeter
EMFT	Endcap Muon Track Finder
FPGA	Field Programmable Gate Array
GEM	Gas Electron Multiplier
GMIO	Global Memory Input-Output
GMT	Global Muon Trigger
GPU	Graphic Processing Unit
GT	Global Trigger
GTT	Global Track Trigger
HBM	High-Bandwidth Memory
HCAL	Hadron Calorimeter
HDL	Hardware Description Language
HF	Hadron Forward calorimeter
HGCAL	High-Granularity Calorimeter
HL-LHC	High-Luminosity LHC
HLS	High Level Synthesis
HLT	High-Level Trigger
ILP	Instruction Level Parallelism
IU	Ingestion Units
L1	Level-1
L1A	Level-1 Accept
L1T	Level-1 Trigger
LHC	Large Hadron Collider
LEP	Large Electron-Positron
LSB	Least Significant Bit
MAC	Multiply Accumulator
MTD	MIP Timing Detector
NGT	Next Generation Triggers
NoC	Network on Chip
OMFT	Overlap Muon Track Finder
PCIe	PCI express

PF	Particle Flow
PL	Programmable Logic
PLIO	Programmable Logic Input-Output
PS	Processing System
PU	pileup
PUPPI	Pile-Up Per Particle Identification
QCD	Quantum Chromo Dynamics
RISC	Reduced Instruction Set Computer
RPC	Resistive Plate Chamber
RTL	Register Transfer Level
SC	Satellite Controller
SIMD	Single Instruction Multiple Data
SLR	Super Logic Region
SM	Standard Model
SoC	System on Chip
SRS	Shift-round Saturate
SU	Storage Unit
TF	Track Finder
TP	Trigger Primitive
VLIW	Very Long Instruction Word
XDMA	Xilinx Direct Memory Access
XRT	Xilinx RunTime library