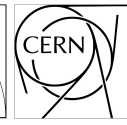
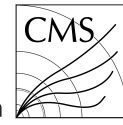


**TECHNISCHE
UNIVERSITÄT
DRESDEN**



Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Master's Thesis

Acceleration of Physics Algorithms on FPGA Platforms to perform Online Analysis within the CMS Phase-2 Level-1 Data Scouting System

Leah-Louisa Sieder

Matriculation number: 5032935

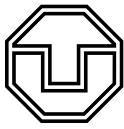
18th June 2025

Supervisor

Dr. Giovanni Petrucciani

Supervising professor

Prof. Dr.-Ing. Diana Göhringer



Task for the preparation of a Master Thesis

Course:	Master Thesis
Name:	Leah-Louisa Sieder
Matriculation number:	5032935
Title:	Acceleration of Physics Algorithms on FPGA Platforms to perform Online Analysis within the CMS Phase-2 Level-1 Data Scouting System

Objectives of work

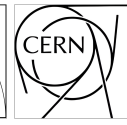
Collision events in the CMS experiment at the LHC are first processed by a hardware-based Level-1 Trigger System (L1T) to perform event reconstruction and select the most promising ones for data acquisition and further processing by the High-Level Trigger. The upcoming high luminosity run of the LHC entails a major upgrade of the CMS detector and the L1T event reconstruction. This enables the introduction of an alternative Data Scouting approach, where physics analysis is performed on all reconstructed events without a preselection. The L1 Data Scouting System (L1DS) collects and analyses trigger objects produced by the L1 processors at the accelerator bunch crossing rate of 40MHz. This project aims to create a hardware implementation of common physics algorithms to run online selection on scouting data at the full bunch crossing rate of 40MHz. The design should be able to act as an accelerator running next to the CMSSW software framework on the server processing farm (e.g. on an Alveo U55C HPC Card) but also to be integrated into a hw design targeting a standalone platform within the DAQ pipeline (e.g. the Versal HBM Series VHK158 Evaluation Kit).

Focus of work

- Get familiar with the provided hardware and understand common computations in physics analyses
- Search for existing implementations and related work
- Design hardware modules for common physics computations (e.g. isolation, delta-r)
- Start with an implementation of the $W \rightarrow 3\pi$ analysis and develop generic modules for common computations in physics analyses.
- Document and evaluate the implementation and results

First referee:	Prof. Dr.-Ing. Diana Göhringer
Second referee:	Dr.-Ing. Lester Kalms
Supervisor:	Dr. Giovanni Petrucciani
Issued on:	15th January 2025
Due date for submission:	18th June 2025

Prof. Dr.-Ing. Diana Göhringer
Supervising professor



Statement of authorship

I hereby certify that I have authored this document entitled *Acceleration of Physics Algorithms on FPGA Platforms to perform Online Analysis within the CMS Phase-2 Level-1 Data Scouting System* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. During the preparation of this document I was only supported by the following persons:

Giovanni Petrucciani

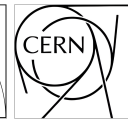
Additional persons were not involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 18th June 2025

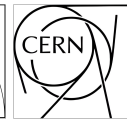
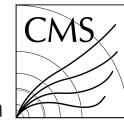
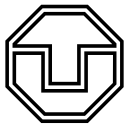
Leah-Louisa Sieder



**TECHNISCHE
UNIVERSITÄT
DRESDEN**



Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems



Abstract

The upcoming High-Luminosity Large Hadron Collider (HL-LHC) era will present unprecedented computational challenges for particle physics research, with instantaneous luminosities reaching up to $7.5 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$ and pile-up of up to 200 simultaneous collisions per bunch crossing. The CMS Phase-2 upgrade introduces the Level-1 Data Scouting (L1DS) system, which proposes to analyze all reconstructed events at the full bunch crossing rate of 40 MHz without preselection, requiring revolutionary approaches to real-time data processing that go beyond traditional CPU-only server architectures.

This thesis investigates the feasibility of Field-Programmable Gate Array (FPGA) acceleration for particle physics online analysis algorithms within the L1DS framework. Specifically, this work presents the design, implementation, and evaluation of FPGA-based realizations of the $W \rightarrow 3\pi$ and $W \rightarrow \pi\gamma$ decay analyses on the AMD Alveo U55C platform. The research investigates whether FPGAs can meaningfully complement and extend CPU-based systems to accommodate additional computational functions and enhance processing capacity for real-time particle physics analysis while maintaining necessary computational precision and algorithmic flexibility.

The implementation shows the development of modular, scalable hardware functions for fundamental computational primitives, including angular distance calculations, isolation computation, and combinatorial operations. It is demonstrated that through the implementation of iterative optimization cycles, the final FPGA implementation achieves competitive performance metrics. The results show that the FPGA implementation delivers the best total per-orbit runtime for bulk processing and nearly matches GPU performance levels for analysis kernel execution time. Performance evaluation reveals that the FPGA implementation consumes significantly less power than the GPU version while utilizing approximately 30% of available computational resources, highlighting substantial energy efficiency benefits. Additionally, a small library with ROOT-like user interface was developed to generate code for both software and hardware-accelerated analysis execution, enabling seamless comparison and validation between computational approaches.

This work provides practical insights into the viability of FPGA acceleration for next-generation trigger and particle physics data acquisition systems, demonstrating that specialized hardware can successfully enhance computational performance for high-energy physics applications while offering energy-efficient alternatives to traditional processing paradigms.

Contents

List of Figures	IV
List of Tables	VI
List of Listings	VII
Acronyms	VIII
1 Introduction	1
2 Background	4
2.1 Fundamentals of Particle Physics	4
2.1.1 Standard Model	4
2.1.2 Experimental Side	6
2.2 CMS Experiment at the CERN Large Hadron Collider	6
2.2.1 Large Hadron Collider	7
2.2.2 CMS Detector	9
2.2.3 Trigger and Data Acquisition System	13
2.3 CMS Phase-2 Upgrade for the HL-LHC	15
2.3.1 Sub-detector Upgrades	16
2.3.2 Level-1 Trigger System Upgrade	16
2.4 CMS Level-1 Data Scouting	18
2.4.1 Baseline Architecture	19
2.4.2 Demonstrator System	19
2.4.3 Physics Case	21
3 Related Work	23

4	Hardware-Software Design	25
4.1	Target Platform	25
4.1.1	AMD Alveo U55C	25
4.1.2	AMD Vitis Unified Software Platform	26
4.2	General Architecture	27
4.3	Implementation of the $W \rightarrow 3\pi$ Analysis	29
4.3.1	Input Data	30
4.3.2	Hardware Design	32
4.3.3	Combinatorics	33
4.3.4	Trigonometric Functions	34
4.4	Implementation of the $W \rightarrow \pi\gamma$ Analysis	35
4.4.1	Input Data	35
4.4.2	Hardware Design	36
4.5	Host Implementation	38
4.5.1	OpenCL API	38
4.5.2	XRT Native API	38
4.6	Further Design Space Exploration and Optimization	39
5	Library Implementation	45
5.1	Concept	45
5.1.1	ROOT Framework	46
5.1.2	Project Structure	47
5.2	Implementation	48
5.2.1	Interface	48
5.2.2	Software Analysis	49
5.2.3	Hardware Analysis	49
6	Results and Evaluation	52
6.1	Testing Environment	52
6.1.1	Datasets	53
6.1.2	Hardware Devices	53
6.2	Physics Performance	55
6.3	Computational Benchmarking	58
6.3.1	Runtime Comparison	59
6.3.2	Resource Utilization	62
6.3.3	Power Consumption	65
7	Conclusion and Future Work	67

Bibliography	70
A Appendix	76
A.1 HLS and ROOT Analysis Implementation	76
A.2 Power Measurements	81

List of Figures

2.1	Particles of the standard model [12].	5
2.2	The CERN accelerator complex [18].	8
2.3	Development of the pileup (left) and integrated luminosity (right) over the years [20].	9
2.4	CMS underground experimental (UXC) and service cavern (USC) [21].	10
2.5	The CMS coordinate system (left) and the range of η values (right) [22].	12
2.6	Structure of the CMS detector [23].	14
2.7	The CMS trigger chain made of L1T and HLT (left) and the structure of the L1T (right) [29, 21].	15
2.8	Structure of the Phase-2 L1T [40].	18
2.9	CMS Level-1 Data Scouting.	19
2.10	Baseline Architecture of the Phase-2 Level-1 Scouting System [44].	20
2.11	Structure of the Demonstrator System [44].	21
4.1	Floorplan of the Virtex XCU55 UltraScale+ FPGA [57].	27
4.2	General layout of the implementation using either A) one kernel for memory reads/writes and the analysis computation, or B) three kernels separating memory reads, writes and analysis computation.	29
4.3	Steps of the $W \rightarrow 3\pi$ analysis.	30
4.4	Format of the PUPPI input data with A) event header, B) charged particles and C) neutral particle; and how the data is streamed D).	31
4.5	Hardware design of the $W \rightarrow 3\pi$ analysis implementation.	32
4.6	Process of combining data in the reworked combinatorics computation module. . .	33
4.7	Vivado device view showing the congested areas of the old implementation (left) and the resource usage of the new (right) combinatorics module.	34
4.8	Steps of the $W \rightarrow \pi\gamma$ analysis.	36
4.9	Format of the e/γ input data with A) header, B) photons (γ) and C) electrons (e); and how the data is streamed D).	37
4.10	Hardware design of the $W \rightarrow \pi\gamma$ analysis implementation.	37

4.11	Development of the per-obit runtimes during design space exploration. The runtimes below $10^4\mu s$ are plotted on a linear scale and above $10^4\mu s$ on a logarithmic scale. . .	40
4.12	Hardware design of the $W \rightarrow 3\pi$ analysis using 5 HBM channels.	41
4.13	$W \rightarrow 3\pi$ analysis computation pipeline stages with A) low pt-cut before isolation and B) low pt-cut after isolation.	42
5.1	Concept of the library implementation.	47
6.1	Invariant mass distribution of the $W \rightarrow 3\pi$ analysis compared to the reference implementation on CPU and with the Vitis software emulation results.	56
6.2	Invariant mass distribution of the $W \rightarrow \pi\gamma$ compared to the reference implementation on CPU and with the Vitis software emulation results.	58
6.3	Analysis runtime measurements on CPU, GPU, AIE and FPGA.	60
6.4	I/O time measurements on CPU, GPU, AIE and FPGA.	61
6.5	Runtime distribution showing the ratio of i/o time on the total execution time across CPU, GPU, AIE and FPGA platforms.	62
6.6	Power consumption over the entire application runtime of the FPGA bulk processing version, taken with a sample rate of ~ 20 ms.	65

List of Tables

2.1	Overview of LHC collision specifications.	9
3.1	Comparison of the $W \rightarrow 3\pi$ Analysis project features of the implementations from the University of Colorado, the University of Milan and the AnaAccel implementation developed in this work, as well as the implementations on AIE and GPU.	24
4.1	Mapping of the particle's ID (PID) to PDGID.	31
6.1	Details of the datasets used.	53
6.2	Efficiency of the $W \rightarrow 3\pi$ analysis on different hardware architectures.	57
6.3	Efficiency of the $W \rightarrow \pi\gamma$ analysis on CPU and FPGA.	57
6.4	Execution times (per orbit) of the $W \rightarrow 3\pi$ analysis on different hardware devices. .	59
6.5	Execution times (per orbit) of the $W \rightarrow \pi\gamma$ analysis comparing the monolithic kernel design with the three-kernel (input, analysis, output) approach.. . . .	62
6.6	Resource usage on different hardware platforms.	63
6.7	Resource usage on FPGA by SLR.	64
6.8	Resource usage of the $W \rightarrow \pi\gamma$ analysis, comparing the monolithic kernel design with the three-kernel (input, analysis, output) approach.	64
6.9	Power statistics over the entire application runtime of the FPGA bulk processing implementation compared to the GPU implementation on the Nvidia L4.	66

List of Listings

4.1	HLS C++ function to perform the low pt-cut with a <code>while(true)</code> loop and break condition.	43
5.1	Minimal non-functional code example of the $W \rightarrow 3\pi$ analysis steps in ROOT.	46
5.2	Main function to define the analysis steps.	48
5.3	Function computing the angular-separation computation in software.	49
5.4	Function computing the angular-separation computation on hardware.	50
5.5	Generated HLS C++ entry function of the analysis computation.	51
6.1	Kernel runtime measurement of the single kernel implementation.	52
A.1	HLS C++ entry function of the analysis computation.	76
A.2	ROOT version of the analysis computation.	79
A.3	Alveo U55C idle power measurement obtained from <code>xbutil</code>	81
A.4	Alveo U55C power measurement during kernel execution obtained from <code>xbutil</code>	82

Acronyms

AIE	AI Engine	24, 39, 43, 52, 53, 55, 56, 58–63, 68
BDT	Boosted Decision Tree	23
BMTF	Barrel Muon Track Finder	14
BNL	Block Nested Loops	33
BRAM	Block RAM	26, 55, 63, 64, 66
BX	Bunch Crossing	7–9, 16–19, 30, 31, 35
CERN	European Organization for Nuclear Research	6, 7, 15, 57
CL	Correlator Layer	17, 19
CLI	Command Line Interface	26
CMS	Compact Muon Solenoid	9–11, 13, 16, 18–20, 23, 24, 27, 28, 53, 59, 67
CSC	Cathode Strip Chamber	13, 14
CT	Correlator Trigger	17
CU	Compute Unit	28, 36, 38, 39, 41– 43
DAQ	Data Acquisition	19
DNN	Deep Neural Network	23
DSP	Digital Signal Processing	23, 26, 55, 63, 64
DT	Drift Tube	13, 14
DTH	DAQ and Timing Hub	19
ECAL	Electromagnetic Calorimeter	12, 13, 16
EMTF	Endcap Muon Track Finder	14
FF	Flip-Flop	23

FPGA	Field Programmable Gate Array	1, 2, 17, 19–21, 23–28, 35, 36, 38, 41, 42, 45, 48, 52, 53, 55–63, 65–69
GCT	Global Calorimeter Trigger	17, 19
GEM	Gas Electron Multiplier	13
GMT	Global Muon Trigger	14, 17, 19
GT	Global Trigger	14, 15, 17, 19
GTT	Global Track Trigger	17, 19, 23, 24
HBM	High Bandwidth Memory	19, 25, 26, 28, 29, 32, 36, 38–44, 48, 63–65
HCAL	Hadron Calorimeter	13
HEP	High Energy Physics	4
HGCAL	High Granularity Calorimeter	16, 17
HL	High Luminosity	16, 18
HLS	High Level Synthesis	27, 32, 45, 47–50
HLT	High-Level Trigger	13–15, 18
IDE	Integrated Development Environment	26
IU	Ingestion Unit	19
L1DS	Level-1 Data Scouting	2, 18, 19, 21, 23, 24, 27, 61, 68
L1T	Level-1 Trigger	13–18, 20, 21, 23, 24
LEIR	Low Energy Ion Ring	7
LHC	Large Hadron Collider	6–11, 15, 16, 19
LINAC3	Linear Accelerator 3	7
LINAC4	Linear Accelerator 4	7
LUT	Look-Up Table	23, 26, 34, 55, 56, 63, 64
ML	Machine Learning	4, 17, 23, 24
MTD	MIP Timing Detector	16
OMTF	Overlap Muon Track Finder	14
PC	Pseudo-Channel	26
PDGID	Particle Data Group Identifier	21, 29
PF	Particle Flow	17, 19
PS	Proton Synchrotron	7
PSB	Proton Synchrotron Booster	7

PUPPI	Pileup Per Particle Identification	17, 19–21, 24, 29, 30, 34–36, 53
RPC	Resistive Plate Chamber	13, 14
RTL	Register Transfer Level	27
SCX	Surface Data Center	10, 19
SLR	Super Logic Region	25, 42, 63
SM	Streaming Multiprocessor	54, 60, 62, 63
SPS	Super Proton Synchrotron	7
TCDS2	Trigger Control and Distribution System for Phase-2	20
TDP	Thermal Design Power	26, 54, 55
TF	Track Finder	17
TP	Trigger Primitives	13, 14
URAM	Ultra RAM	26, 55
USC	Underground Service Cavern	10, 19, 20
UXC	Underground Experimental Cavern	10
XRT	Xilinx Runtime	26, 65

1 Introduction

Motivation

The upcoming High Luminosity Large Hadron Collider (HL-LHC) era [1] presents unprecedented challenges and opportunities for particle physics research. With instantaneous luminosities reaching up to $7.5 \times 10^{34} \text{ cm}^{-2}\text{s}^{-1}$ and a pile-up of up to 200 simultaneous collisions per bunch crossing, the HL-LHC will generate enormous volumes of data that must be processed in real-time. This dramatic increase in data throughput necessitates revolutionary approaches to event selection and analysis, pushing the boundaries of current computing paradigms in high-energy physics.

Traditional trigger systems employ a hierarchical approach where the Level-1 Trigger (L1T) [2] performs initial event reconstruction and selection, retaining only the most promising collision events for further processing. While this approach has proven successful, it inherently introduces selection bias and potentially discards rare physics processes that could lead to groundbreaking discoveries. The CMS Phase-2 upgrade introduces a novel, parallel and complementary approach: the Level-1 Data Scouting (L1DS), which proposes to analyze all reconstructed events at the full bunch crossing rate of 40 MHz without preselection. This approach promises to preserve the complete physics content of collision data, enabling searches for rare processes and previously inaccessible physics signatures. Implementing such a system requires advanced processing capabilities that make it particularly interesting to explore the feasibility of heterogeneous architectures beyond CPU-only server setups to perform online analysis.

Field Programmable Gate Arrays (FPGAs) emerge as a promising solution to this computational challenge. Their parallel processing architecture, high-throughput streaming capabilities, and ability to implement custom algorithms at the hardware level make them ideally suited for testing real-time particle physics analysis implementations. While FPGA acceleration has been successfully applied to various particle physics computations, including machine learning inference and anomaly detection algorithms [3, 4], the specific implementation of rare decay analyses such as the $W \rightarrow 3\pi$ [5] represents largely uncharted territory as these analyses have so far been carried out primarily

offline. Although parallel development efforts are ongoing within the CMS collaboration, exploring different approaches or hardware architectures [6, 7, 8, 9], no complete FPGA-based solution for $W \rightarrow 3\pi$ analysis has been demonstrated to date. This work, therefore, represents a pioneering effort in bringing rare physics analyses to hardware acceleration platforms.

Objectives

This thesis aims to design, implement, and evaluate a FPGA-based realization of the $W \rightarrow 3\pi$ and other online analyses to investigate the feasibility of FPGA-equipped accelerator cards within the Level-1 Data Scouting (L1DS) system. The work addresses the fundamental question of whether reconfigurable hardware can meaningfully complement and extend CPU-based systems to accommodate additional computational functions and enhance processing capacity for real-time particle physics analysis while maintaining necessary computational precision and algorithmic flexibility. This thesis therefore explores hardware design approaches for complex and combinatorial computations essential in the field of particle physics analyses.

Technical Objectives:

1. **Algorithm Analysis and Decomposition:** Analyze the computational structure of particle physics online analysis algorithms, primarily for the $W \rightarrow 3\pi$ decay, to identify parallelization opportunities, data dependencies, and optimization strategies suitable for hardware implementation.
2. **Hardware-Software Design:** Develop modular, scalable hardware functions for fundamental computational primitives, including angular distance calculations, isolation computation, and combinatorial operations commonly used in scientific computing.
3. **Implementation and Optimization:** Implement a complete processing pipeline for $W \rightarrow 3\pi$ analysis that demonstrates the practical integration of multiple computational modules, addressing challenges in data flow management, pipeline synchronization, and system-level optimization.
4. **Performance Evaluation:** Conduct comprehensive performance analysis by measuring runtime, resource utilization, and power consumption, and benchmarking against implementations on different hardware platforms.

By achieving these objectives, this thesis provides practical insights into the viability of FPGA acceleration for the L1DS system, offering evidence-based conclusions about the potential for hardware acceleration in next-generation trigger and particle physics data acquisition systems.

Structure

This thesis is organized into seven chapters that provide an introduction to the topic, present the concepts and implementation details of the work conducted, and discuss and evaluate the results obtained. The following section provides a brief overview of the structure of the thesis and the content of the individual chapters.

Chapter 2 provides the necessary background knowledge, introducing the fundamentals of particle physics and the Standard Model, followed by a detailed overview of the CMS experiment at CERN's Large Hadron Collider. The chapter concludes with an examination of the planned Phase-2 upgrade for the High-Luminosity LHC and the conceptual framework of CMS Level-1 Data Scouting, establishing the physics motivation and technical context for this work.

Chapter 3 reviews related work in the field, positioning this research within the broader landscape of hardware acceleration approaches for high-energy physics online analysis applications and identifying gaps that this thesis aims to address.

Chapter 4 presents the core technical contribution of this work: the hardware-software design of the FPGA-based particle physics analysis implementation. This chapter introduces the target AMD Alveo U55C platform and the Vitis development environment, outlines the general architecture, and provides insight into the implementation details of two specific physics analyses: $W \rightarrow 3\pi$ and $W \rightarrow \pi\gamma$ decay channels. The chapter also explores design space optimization strategies to maximize performance.

Chapter 5 describes the development of a small library that combines the hardware implementations of the $W \rightarrow 3\pi$ analysis with the ROOT framework commonly used in particle physics. This chapter explains the library's concept, project structure, and implementation details, including both the user interface and the underlying software and hardware analysis components.

Chapter 6 evaluates the developed system through comprehensive testing and benchmarking. The chapter presents physics performance validation using signal and background datasets, followed by detailed computational benchmarking that includes runtime comparisons, resource utilization analysis, and power consumption evaluations across different hardware platforms.

Chapter 7 summarizes the key findings and contributions of this work, discusses the implications for future physics analysis acceleration approaches, and outlines potential directions for continued research and development.

2 Background

This chapter conveys the basic context of this work and should help to understand the concept and implementation aspects. It covers a short introduction to particle physics, providing an overview of the theoretical and experimental foundations to the science aimed at understanding the fundamental constituents of matter. Subsequently, it explains where this work can be placed within the entire accelerator complex at CERN. This is accomplished by providing an overview of the structure and functionality of the CMS experiment at the Large Hadron Collider covering the detector itself as well as the trigger and data acquisition chain. The chapter further describes what upgrades the accelerator and the CMS experiment will undergo in the near future, what the novel data scouting approach is and how it will be integrated into the experiment.

2.1 Fundamentals of Particle Physics

Particle physics, also known as High Energy Physics (HEP), is a branch of physics that deals with the fundamental constituents of matter and the forces controlling their interactions. As it is tightly coupled with quantum physics, which provides the theoretical foundation for many phenomena in the behavior and interaction of particles on a subatomic scale, many processes are of probabilistic nature and cannot be predicted deterministically. Consequently, particle physics research relies primarily on the statistical analysis of large datasets and on advanced computational techniques, including large-scale data processing, computer-aided simulation, and Machine Learning (ML).

2.1.1 Standard Model

At the core of particle physics stands the *standard model* [10, 11]. This theoretical framework categorizes all known elementary particles and describes three of the four known fundamental forces: the electromagnetic, weak and strong interaction (excluding gravity). Figure 2.1 presents a table showing the organization of particles according to the Standard Model. Elementary particles

are grouped into *fermions*, which are the constituents of matter, and *bosons*, which are force carriers. Each particle has specific properties like mass, charge, and spin. The interplay between these particles is described using quantum field theories, such as *quantum electrodynamics* for electromagnetism and *quantum chromodynamics* for the strong force. The Standard Model divides fermions into three generations with increasing mass. Each generation consists of two *quarks*, one $2/3$ positively and one $1/3$ negatively charged, and two *leptons*, one charged and one neutral. The bosons can be assigned to the fundamental forces they mediate. The electromagnetic force is carried by *photons*, the weak force by *W* and *Z bosons* and the strong force by *gluons*. The *higgs boson*, discovered 2012 at CERN, gives mass to other particles via the *higgs field*. All elementary particles have a corresponding anti-particle, such as the anti-photon or the positron, that have the same properties but opposite charge characteristics.

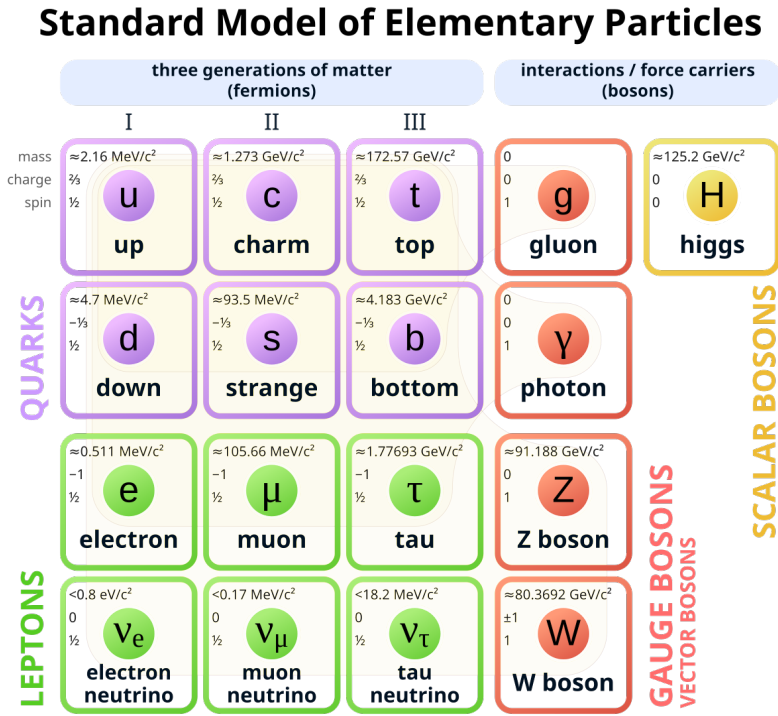


Figure 2.1: Particles of the standard model [12].

In addition to the elementary particles described in the standard model, there are composite particles that consist of multiple elementary particles often strongly interacting with each other. The best known composite particles are *hadrons*, made of quarks that are held together by gluons via the strong interaction. There are two types of hadrons: *baryons* such as protons and neutrons, that consist of three quarks, and *mesons*, such as *pions* (π) and *kaons* (K), that consist of a quark and an anti-quark. Composite particles can form further composite particles such as atomic nuclei.

2.1.2 Experimental Side

To study and experimentally validate the calculated theories, large particle accelerators [11] are built to bring charged particles close to the speed of light and collide them together or with a fixed target. The particles need to be accelerated to such high energies, that the energy of these collisions is transformed into mass or matter in the form of new particles, especially heavy particles that do not exist freely in nature. This phenomenon is described by Einstein's equation $E = mc^2$, which indicates that mass is a concentrated form of energy and both are interchangeable. During particle collisions, the energy of the particles colliding is concentrated in such small volume that conditions similar to those moments after the big bang are created. In such environments, the energy can transform into various particles as described in the standard model. These quantum processes are probabilistic, so it is never certain exactly which interactions are taking place. Many particles that are created are unstable and immediately decay into lighter, more stable particles. The role of the detectors is then to track these decay products back to the origin of their interaction, the primary or one of the secondary *vertices*.

Emerging from the point of collision, a key phenomenon often observed is so-called *Jets*. Jets are collimated sprays of particles, namely quarks and gluons, that are produced in the initial state of the collision. Due to the fact that quarks and gluons cannot exist freely, a cascade of hadrons is formed in the process of hadronization. These hadrons all travel roughly in the same direction, along with many other particles created in subsequent decays of unstable hadrons.

2.2 CMS Experiment at the CERN Large Hadron Collider

Near Geneva, on the border between France and Switzerland, the European Organization for Nuclear Research (CERN), is located. CERN operates the world's largest laboratory for particle physics. It is currently operated by 24 member states and is part of a worldwide network dedicated to particle physics research. CERN is particularly renowned for operating the largest and most powerful particle accelerator in the world, the Large Hadron Collider (LHC) [13].

The LHC is a 27 km long circular collider that is located between 45 m and 170 m underground. It accelerates beams of protons (or, for only a few weeks each year, heavy ions) to velocities close to the speed of light in both, clock- and counterclockwise directions and make them collide at four designated locations. At these collision points four particle detector experiments have been constructed to observe and record the collisions. These four experiments comprise two general-purpose detectors CMS [14] and ATLAS [15] and two detectors specialized in investigating specific physics phenomena: LHCb [16], which focuses particularly on b-quarks and the differences between matter and antimatter; and ALICE [17], a detector dedicated to heavy-ion physics.

The LHC is merely one component among numerous machines in the entire accelerator complex [11] depicted in figure 2.2. It serves as the last element in a multi-step accelerator chain designed to accelerate protons to velocities approaching the speed of light and enable high-energy collisions with a center-of-mass energy of 13.6 TeV. The acceleration process begins with the Linear Accelerator 4 (LINAC4), which accelerates protons to 160 MeV before they enter the Proton Synchrotron Booster (PSB), where they are further accelerated to 2 GeV. Subsequently, they are accelerated to 26 GeV in the Proton Synchrotron (PS) and then injected into the Super Proton Synchrotron (SPS), where they achieve energies of up to 450 GeV. Finally, the two beam pipes of the LHC are filled with these pre-accelerated protons and the particles are further accelerated to 6.5 TeV. For ion runs, the procedure follows a similar pattern, except that the Linear Accelerator 3 (LINAC3) and the Low Energy Ion Ring (LEIR) are used in the initial stages of the accelerator chain.

In addition to the four main experiments at the LHC, CERN hosts numerous additional smaller fixed-target experiments for various research purposes in particle physics, as well as interdisciplinary projects. These include, for example, research in medical applications such as radiotherapy.

2.2.1 Large Hadron Collider

In 2008, the Large Hadron Collider (LHC) [13] was commissioned as the new final stage of CERN's accelerator complex, and it is now in its third operational run. The LHC consists of two separate beam pipes in which two beams circulate in opposite directions. The beam pipes are surrounded by thousands of superconducting dipole and quadrupole magnets to focus and steer the beam along its circular orbit. To become superconducting, the magnets are made of a special coil Niobium Titanium (Nb-Ti) and are cooled down to 1.9 K (-271.3° C) using liquid helium. Before the beam is made to collide at the four major experiments - ATLAS, CMS, ALICE, and LHCb - it is squeezed by special magnets to achieve a higher probability of collisions.

It takes 4 min and 20 s to completely fill each pipe of the LHC and another 20 min to accelerate the protons to their maximum energy of 6.5 TeV. Once at maximum energy, a particle needs approximately $89.1\mu\text{s}$ to travel the full 27 km circle of the accelerator at 99.999999% of the speed of light.

Beam Structure and Collisions

The beam in the LHC is structured into *bunches* of approximately 10^{11} protons. One revolution of the LHC is called an *orbit* and it can fit up to 3564 bunches. The temporal spacing between two bunches is of 25 ns. Therefore, theoretically, every 25 ns two bunches collide at the experimental interaction points, corresponding to a Bunch Crossing (BX) rate of 40 MHz. However, there are

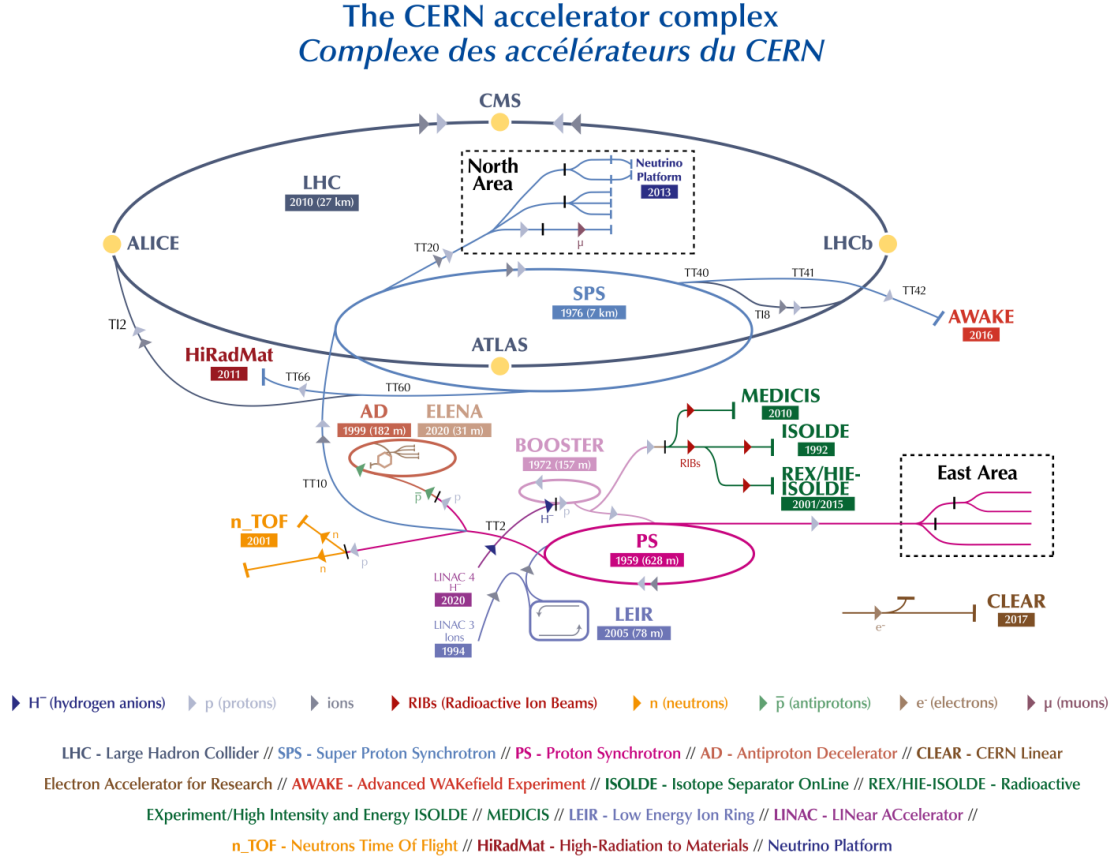


Figure 2.2: The CERN accelerator complex [18].

gaps in the beam structure, leading to an effective average BX rate of 31.6 MHz. The arrangement of the bunches and gaps within the LHC follows a fixed filling scheme that is optimized to maximize the accelerators performance within its operational limitations. For example, this includes a $3\mu s$ abort gap at the end of every orbit to ensure safe beam dumping in case of an emergency, as well as time needed to synchronize and calibrate the front-end electronics [19]. Two beams are circulating and made to collide inside the LHC for many hours before being dumped and another refill is initiated. Table 2.1 provides an overview of the aforementioned beam specifications and collision rates.

A central quantity characterizing the performance of the LHC is the *instantaneous luminosity* \mathcal{L} , which describes the number of collisions per surface unit (cross section) at a specific time. The current average instantaneous luminosity is $\mathcal{L} = 2 \cdot 10^{34} cm^{-2}s^{-1}$. By integrating the instantaneous luminosity over a desired time interval, the *integrated luminosity* $\int_{t_0}^{t_1} \mathcal{L}(t) dt$ is obtained. It is measured in *inverse femtobarns* (fb^{-1}) and is used to quantify the number of collisions recorded over a specific

Beam Specifications	
max. BX rate	40 MHz
eff. BX rate	31.6 MHz
max. #bunches per orbit	3564

Table 2.1: Overview of LHC collision specifications.

time period and is therefore an indicator of the amount of data produced by the detector. For example, by the end of 2018, after approximately one decade of operation, the LHC had produced more than 160fb^{-1} of data [11].

To be able to draw well-founded statistical conclusions and to study rare physics processes, the LHC is designed to achieve the highest possible luminosity. This, however, leads to an increasing number of simultaneous proton-proton collisions in a single BX, a phenomenon known as *pileup*. Currently, an average pileup of up to 60 is reached. The development of pileup and integrated luminosity over the years can be seen in figure 2.3.

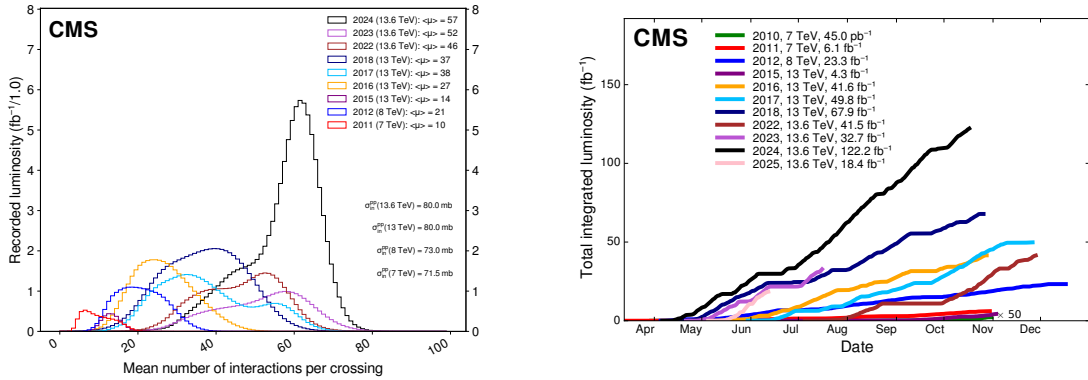


Figure 2.3: Development of the pileup (left) and integrated luminosity (right) over the years [20].

2.2.2 CMS Detector

The Compact Muon Solenoid (CMS) experiment [14] at the LHC is a general-purpose experiment, built to perform standard model and especially higgs-physics research as well as investigations into physics beyond the standard model and the search for new particles that could constitute dark-matter. It serves to investigate almost the same physics phenomena as the ATLAS experiment, but is constructed differently and therefore uses different detection techniques compared to ATLAS. The purpose of having two general-purpose detectors with the same scientific goals is to

be able to independently verify discoveries. Compared to ATLAS, CMS is built more compactly, ensuring the most robust muon detection system possible.

The detector itself is enclosed within a large superconducting solenoid magnet, capable of generating a magnetic field of 3.8 T. Within this magnetic field, the trajectories of charged particles are bent. This allows the detector to identify the charge of a particle (since negatively and positively charged particles bend in opposite directions) and to measure the momentum of a particle (since high-momentum particles trajectories are bent less than low-momentum particles).

The CMS detector is located in Cessy, France, in an Underground Experimental Cavern (UXC) approximately 100 m below the surface along the LHC ring. Adjacent to the UXC, there is a nearby Underground Service Cavern (USC) that is shielded from the radiation inside the UXC and houses much of the operational infrastructure, such as the power supply and cooling systems and the electronics for the trigger and data acquisition system. In addition to the underground caverns, there is the surface control room and additional service rooms, like the Surface Data Center (SCX), which houses a server farm for the final step of the data acquisition. Figure 2.4 illustrates the structure of the CMS underground facilities.

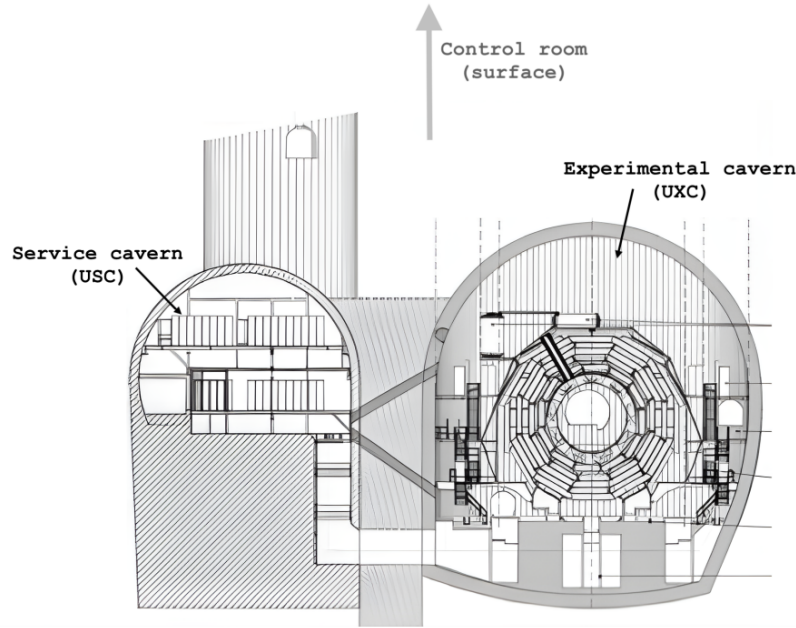


Figure 2.4: CMS underground experimental (UXC) and service cavern (USC) [21].

Coordinate System

To have a reference frame for describing the detector geometry and particle positions after collisions, the CMS experiment defines a right-handed cartesian coordinate system (shown in figure 2.5). It is centered at the bunch crossing point such that the z-axis aligns with the beam line and the x-axis points toward the center of the LHC ring. In addition, due to the cylindrical shape of the detector, a spherical polar coordinate system is employed. This system takes the beam line as the polar axis and defines the usual three polar coordinates $(|\vec{p}|, \theta, \varphi)$ as illustrated in figure 2.5. However, instead of θ , the *pseudo-rapidity*

$$\eta = -\ln \left[\tan \left(\frac{\theta}{2} \right) \right] \quad (2.1)$$

is used, as in relativistic regimes it provides a more uniform representation of the distribution of particles produced in a collision. This can also be seen on the right side of figure 2.5, with η values of $\pm\infty$ representing particles very close to the beam line. Furthermore, the *transverse momentum*

$$\vec{p}_T = \sqrt{p_x^2 + p_y^2} \quad (2.2)$$

is widely used. It describes the momentum projected to the transverse plane (the x-y plane). Additionally, the *angular distance*

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

between two particles is often needed to describe how far two particles produced in a collision are separated. Another important quantity is the *invariant mass*

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

which allows one to describe the properties of a particle as a combination of the energies E_i and momenta \vec{p}_T of its decay products.

Sub-Detectors

The CMS detector has a layered structure consisting of different sub-detectors as can be seen in figure 2.6 [14, 23]. It is divided into the cylindrical part around the beam pipe, called the barrel, and the two end-caps, perpendicular to the beam pipe, which close the cylinder on either side. Around

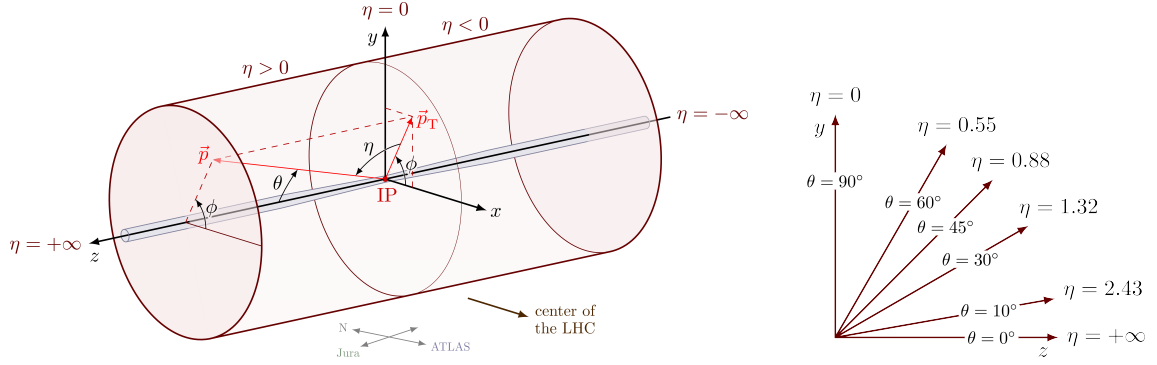


Figure 2.5: The CMS coordinate system (left) and the range of η values (right) [22].

the inner part of the detector is the superconducting solenoid creating a magnetic field of 3.8 T. Inside this magnetic field, charged particles moving outside from the collision point are bent in different directions and in different ways, depending on whether they are positively or negatively charged and on how much momentum they have, since high-momentum particle trajectories are less affected by the magnetic field. Therefore the main tasks of the detector are to identify tracks and determine the trajectory of a measured particle and to obtain the energy of the particle. All of that is achieved by the various sub-detectors, which, from inside out, are:

Silicon Tracker

The silicon tracker [24] consists of semiconducting material, that becomes ionized when a charged particle passes through. It is the innermost layer of the detector, positioned closely adjacent to the interaction point to enable high resolution track reconstruction. It is divided into a pixel detector and a strip detector. The pixel detector is made of $100 \mu\text{m} \times 150 \mu\text{m}$ silicon pixels arranged in four layers around the beam pipe and three disks in the end-caps. The strip detector is located around the pixel detector and consists of narrow silicon strips in multiple layers. Particles traveling through the layers of these detectors pass through different pixels or strips and thus create data points from which the tracks of even short-lived particles can be reconstructed.

Electromagnetic Calorimeter (ECAL)

Around the tracker system, the ECAL [25] is positioned to measure the energy of electrons and photons by stopping them within the very dense material. It is made of scintillating lead tungstate crystals. When photons or electrons pass through this dense material, the crystals produce light that is captured and converted into an electrical signal by photo-detectors positioned at the back of the crystals. From these signals, the energy of the particle can then be computed. The ECAL is divided into a barrel and two end-cap sections with 61,200 and 15,000 crystals, respectively, in total.

Hadron Calorimeter (HCAL)

Outside the ECAL is the HCAL [26] with the purpose of stopping and measuring the energy of hadrons (e.g., protons, neutrons, or pions). It is structured as a so-called sampling calorimeter with alternating layers of very dense absorber material and fluorescent scintillator material. The scintillating material produces light when a particle passes through. This light is captured by photo-detectors and converted into an electric signal to compute the energy of that particle by summing the amount of light over all transversed layers.

Muon System

As it is also part of the name CMS, tracking muons is one essential component of the detector. Muons are charged particles much heavier than electrons, and they pass through material without losing much of their energy, which is why they are not stopped by the calorimeter systems. The muon system [27] is placed outside the solenoid magnet, so it becomes even more unlikely that other particles than muons have made their way to this point. In addition, the layers of the muon system alternate with steel return yokes so that only muons can create a clear signal in the entire multi-layered system. Together with the information from the silicon tracker, the trajectory of the muons can be reconstructed. The system is made up of different kinds of gas detectors: Drift Tubes (DTs), Cathode Strip Chambers (CSCs), Resistive Plate Chambers (RPCs) and Gas Electron Multipliers (GEMs). They all have different properties in terms of, e.g., response time, granularity, or radiation resistance and are placed at specific positions around the detector to build a robust and reliable tracking system.

2.2.3 Trigger and Data Acquisition System

After the readout and digitization of the detector signals are completed, the huge amount of data from all CMS subsystems needs to be processed and stored, so that offline physics analyses can be carried out. Assuming a data size of approximately 1 MB per event and a collision rate of 31.6 MHz, around 30 TB/s of collision data is produced. However, it is impractical to store all of this data due to technical limitations and also unnecessary because a large part of that data contains information about physics processes that are not of interest within the CMS physics program. To realize this online pre-selection, a two-tiered *trigger system* [2, 28, 21] was implemented. It consists of the Level-1 Trigger (L1T) and the High-Level Trigger (HLT) and is able to reduce the event rate from its theoretical maximum of 40 MHz to 1 kHz.

Level-1 Trigger

As illustrated on the left side of figure 2.7, the digitized data of one event from the detectors is temporarily stored in front-end pipelines. From a subset of this data, so-called Trigger Primitives

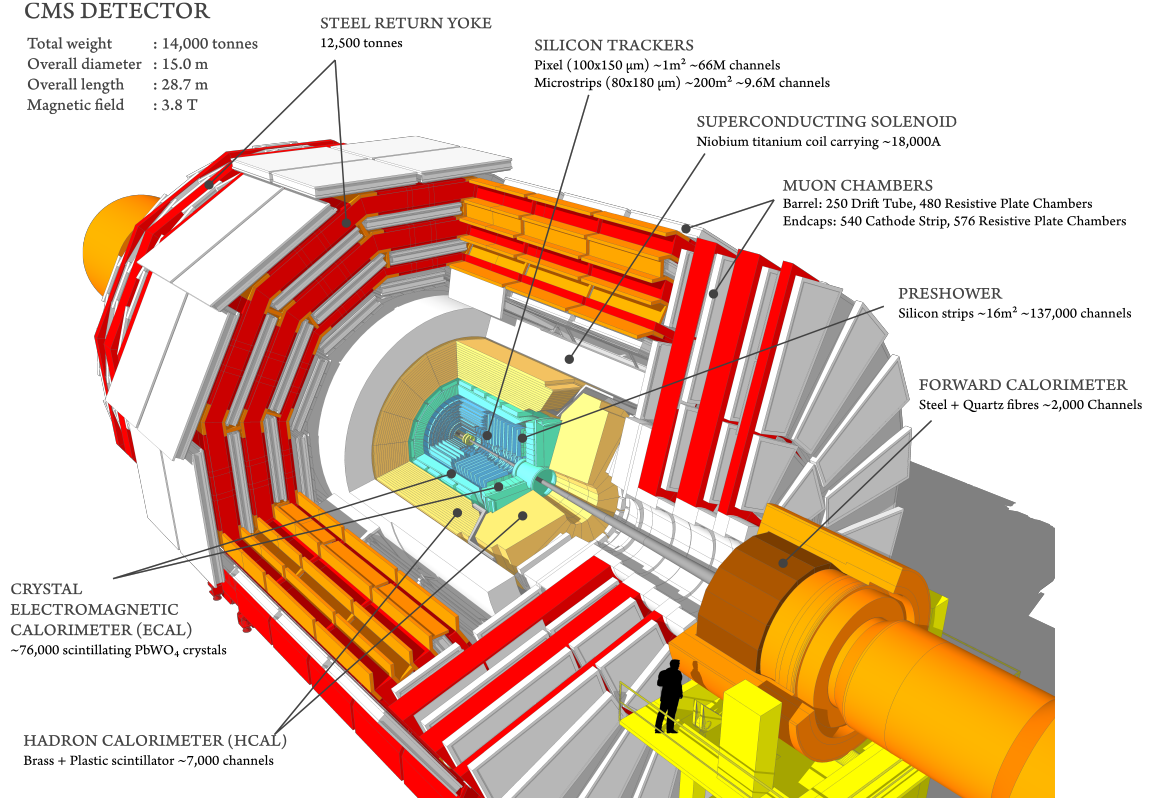


Figure 2.6: Structure of the CMS detector [23].

(TP) are generated, which give a rough picture of the event data. The TP serve as input for the L1T system [30] and form the basis on which the decision of whether an event will be kept or discarded is made. The L1T has a few microseconds to make this decision and reduces the data rate from 40 MHz to around 100 kHz. When an event is selected, the full event data is sent to the HLT. The current L1T system is implemented on custom electronic boards and is structured into two main parts: the muon trigger and the calorimeter trigger. The module structure of the L1T system can be seen on the right side of figure 2.7.

The muon trigger [31] takes TPs from the muon subsystems (CSC, RPC, DT) of the detector and is divided into three sub-modules: the Endcap Muon Track Finder (EMTF), Overlap Muon Track Finder (OMTF), and Barrel Muon Track Finder (BMTF), each covering different η -regions of the detector to find muon tracks in the data. It then sends the candidates to the Global Muon Trigger (GMT), which removes duplicates, sorts and selects the best candidates, and further forwards them to the Global Trigger (GT).

The calorimeter trigger [32] processes TPs from all calorimeters in a two-layered system to reconstruct electron, photon, tau and jet candidates and calculates global energy sums. The objects are

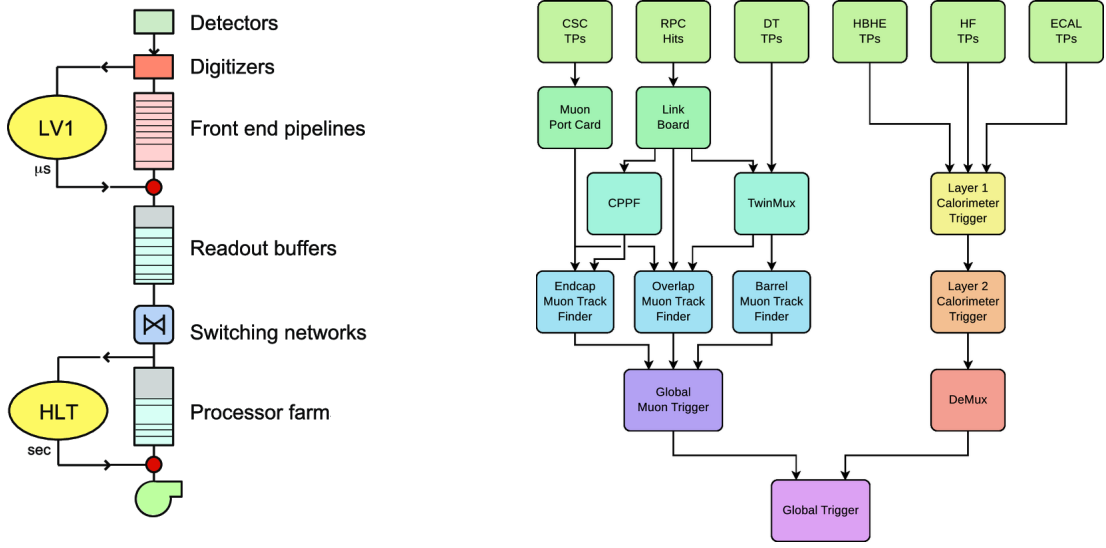


Figure 2.7: The CMS trigger chain made of L1T and HLT (left) and the structure of the L1T (right) [29, 21].

then sorted by their energy and also transferred to the GT.

Based on the best candidates from the muon and calorimeter triggers, the GT [33] ultimately decides whether an event should be accepted or not. To this end, multiple algorithms defined by the *Level-1 Trigger Menu* are executed, applying, e.g., p_T thresholds or making decisions on multi-object correlations.

High-Level Trigger

The HLT [29] receives a combination of raw data from all sub-detectors for the events selected by the L1T. It runs the full event reconstruction on commercial server processors equipped with GPUs and searches for specific signatures by running complex physics algorithms, e.g., fully matching muon tracks to hits in the muon chambers or precisely identifying photons. The HLT takes several hundred milliseconds to process one event and further reduces the event data rate to 1 kHz. The remaining events are then sent to the CERN computing center to be stored.

2.3 CMS Phase-2 Upgrade for the HL-LHC

The LHC has been running since 2008 and is now facing the end of *Phase-1*. During that time, it was upgraded twice in the long shutdowns 1 and 2 and is currently in *Run-3* of data taking. In the next

long shutdown, planned for 2026, the LHC will undergo a major upgrade for its second operational period, *Phase-2*, the High Luminosity (HL)-LHC [1]. The HL-LHC is going to collide particles at a center-of-mass energy of 14 TeV (currently: 13.6 TeV). Right at the beginning of Phase-2, which is planned to start in 2030, an instantaneous luminosity of $\mathcal{L} = 5 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$ should be reached and even increased to up to $\mathcal{L} = 7.5 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$. This also comes with significantly higher average pileup of up to 140 to 200 collisions per BX (currently 60). The integrated luminosity over the entire LHC period of approximately 14 years of data taking is going to be around 450 fb^{-1} . The HL-LHC is expected to produce up to 4000 fb^{-1} of data in 12 years. To be able to reach these high energies, especially the LHC magnets and acceleration cavities will be upgraded. Not only the LHC, but also the detectors around the accelerator ring, among them the CMS experiment, need to be upgraded in many places to be able to withstand the resulting higher amounts of radiation and to keep up with the read-out and processing of the increased amount of collision data [34].

2.3.1 Sub-detector Upgrades

The existing silicon tracker [35] will be replaced by a similar system consisting of pixel and strip sensors. However, the inner tracker will have higher granularity with thinner and even smaller pixel sensors equipped with faster and more radiation-tolerant electronics. The outer tracker will be rebuilt with macro pixels and stacked strip modules. This setup will for the first time allow track reconstruction at L1T level above a specific p_T threshold to meet the bandwidth limitations.

To introduce precise timing measurement to CMS, a completely new layer, the MIP Timing Detector (MTD) [36], will be added to the detector between the tracker and the calorimeters. It will help to match tracks to their corresponding vertices by providing a timing resolution of 30-60 ps - an essential feature when facing much more pileup.

The calorimeters [37, 38] will be equipped with new electronics for better granularity, noise suppression, and, for the ECAL, also timing information. In addition, the end-cap calorimeters will be completely replaced by the new High Granularity Calorimeter (HGCAL), which enhances the energy measurements with precise timing information.

The muon system [39], which was partly already updated in 2020, also gets new electronics to withstand the higher radiation and enable faster and more precise read-out to cope with the higher data rates.

2.3.2 Level-1 Trigger System Upgrade

Not only the components of the detector itself, but also the trigger system will be part of the phase-2 upgrade. The new L1T [40] will be built on new custom electronics boards and modern

FPGAs connected by fast optical links, and it will also have a higher output rate of 750 kHz with a longer possible latency of $12\mu\text{s}$ to be able to handle up to 200 pileup collisions per BX while retaining and even extending the same physics acceptance as the former one. Furthermore, the improved and more precise detector read-out, as described above, enables significantly better object reconstruction in the L1T. To realize all of that, the architecture of the system will undergo some changes, as illustrated in figure 2.8. There will still be a calorimeter trigger, now taking data from the new sub-detectors and combining them in the Global Calorimeter Trigger (GCT). The muon trigger basically retains its structure from the phase-1 system. Additionally, track stubs are sent to the GMT from the Track Finder (TF), which operates in the backend of the detector tracker system. The Global Track Trigger (GTT) will receive data directly from the TF to use this information, e.g., to identify the primary vertex of an event. The key difference, however, is the introduction of the Correlator Trigger (CT), which makes it possible to perform event reconstruction at the level-1. It is divided into two layers: the first Correlator Layer (CL) gets as input the results from the GCT, 3D HGCal information, reconstructed muons, and level-1 tracks from the TF and performs Particle Flow (PF) [41] reconstruction by combining the information from all the sub-detectors to analyze particle interactions and reconstruct high-level physics objects. It also executes the Pileup Per Particle Identification (PUPPI) algorithm [42] to significantly mitigate the pileup influence. In CL2, based on the so called PUPPI candidates, even more high-level physics algorithms, implemented in hardware, are run, such as global jet clustering or hadronic tau reconstruction. Finally, the GT analyzes all these objects by executing cut-based or ML-based algorithms from the new L1T menu to decide which events to keep and which to discard. The new GT is even capable of considering objects from multiple BX to better take long-lived particles or exotic signatures into account.

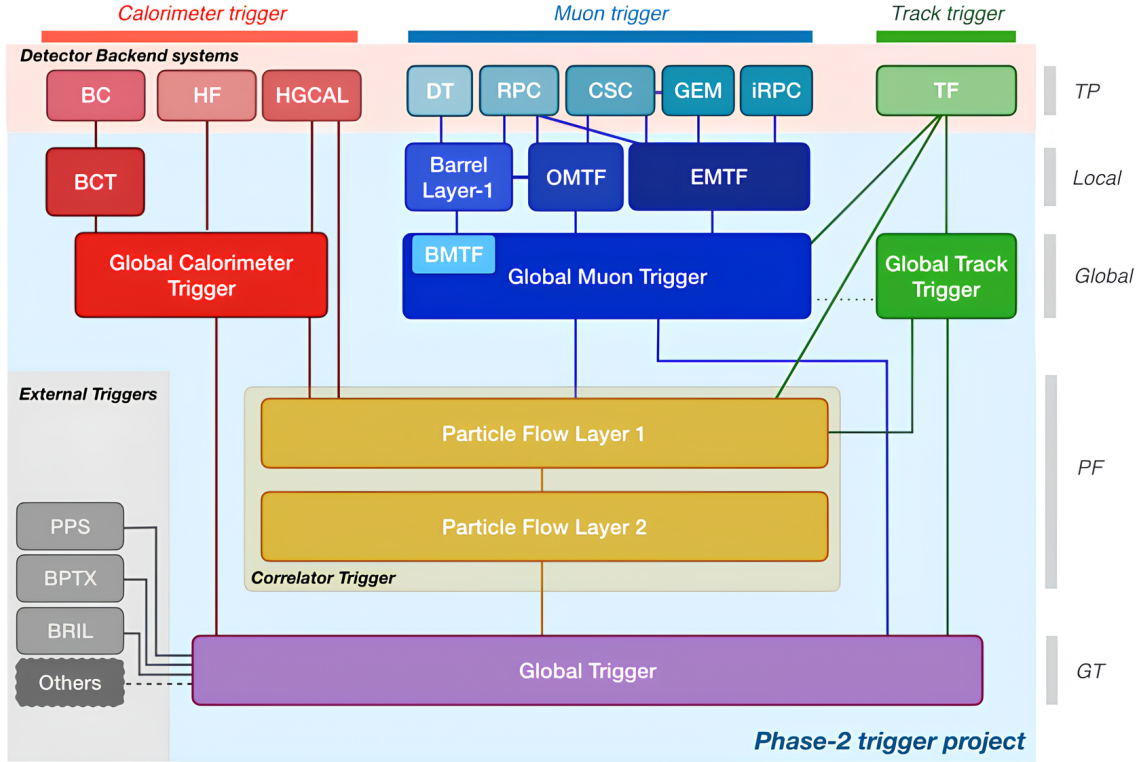


Figure 2.8: Structure of the Phase-2 L1T [40].

2.4 CMS Level-1 Data Scouting

The traditional CMS trigger system, consisting of L1T and HLT serves as a key element in the CMS data-taking chain and makes a large contribution to the success of the entire experiment. However, despite its upgrades and constantly improving performance, in order to meet the latency and maximum acceptance rate dictated by the read-out bandwidth, the offline storage capacity, and processing ability, it introduces bias to the data recorded for further physics analyses, as only those events that pass all predefined selection criteria are stored. To overcome these limitations due to the fear of unknowingly filtering out interesting physics data and to gain access to physics otherwise constraint by the trigger menu, the L1DS [43, 44, 45] approach has been developed. The L1DS acts as an independent system alongside the traditional trigger chain, as can be seen in figure 2.9. It collects and further processes L1T objects from the usual L1T boards at the full BX rate of 40 MHz and enables online physics analysis to be run on this unconstrained data; especially with the HL upgrade and the greatly improved object reconstruction of the L1T.

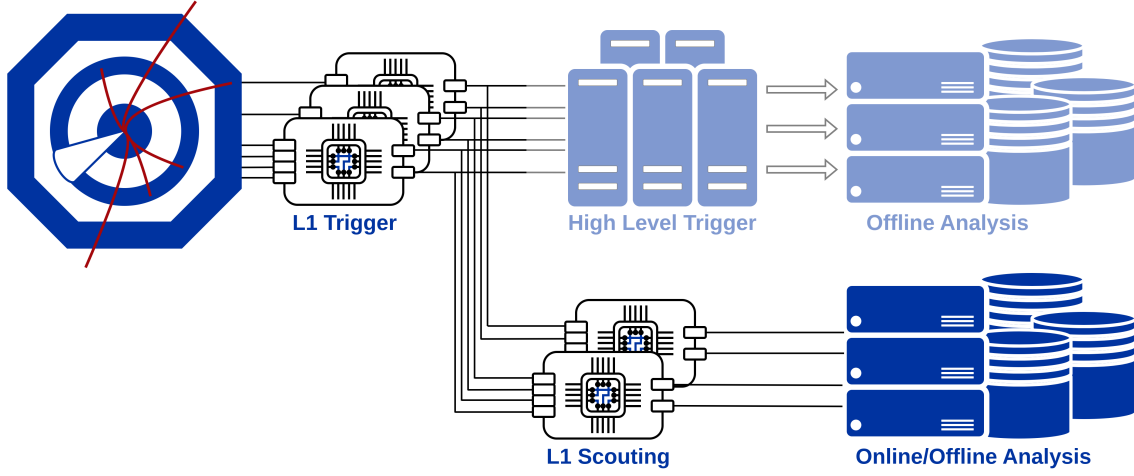


Figure 2.9: CMS Level-1 Data Scouting.

2.4.1 Baseline Architecture

The baseline architecture of the phase-1 L1DS system is illustrated in figure 2.10. The trigger objects are directly collected from the GCT, the GTT, and the GMT, as well as final trigger results from the GT and data from the two CLs. From the CL2 PF candidates with (PUPPI candidates) or without pileup subtraction can be obtained. The data is sent to DAQ800 ATCA boards [46] installed in the CMS USC. The boards are custom-designed for CMS phase-2 Data Acquisition (DAQ) with a focus on high throughput and one board can receive data on up to 48 high-speed optical input links at 25 Gb/s. They are equipped with two Xilinx VU35P FPGAs with High Bandwidth Memory (HBM) and perform interpretation and aggregation of the incoming trigger link protocol data, as well as pre-processing such as zero-suppression to reduce the data throughput before buffering it in the HBM and further transferring it to a commercial switch located in the SCX using TCP/IP on long-range optical links at 100 Gb/s. The data is further sent to the Ingestion Units (IUs) via 400 Gb/s links. The IUs generate sets of data containing information from all BXs of one LHC orbit (approx. $89\mu\text{s}$) and transmit them to a server processing farm. On these servers, the information from all data streams is combined to perform online analysis and send the results to permanent storage. In addition to the online analysis results, so-called zero-bias data is also saved: data from every BX of an orbit every N orbits.

2.4.2 Demonstrator System

To test and check the performance of the planned baseline architecture, a demonstrator system was set up in the laboratory as depicted in figure 2.11. It consists of a custom DAQ and Timing Hub

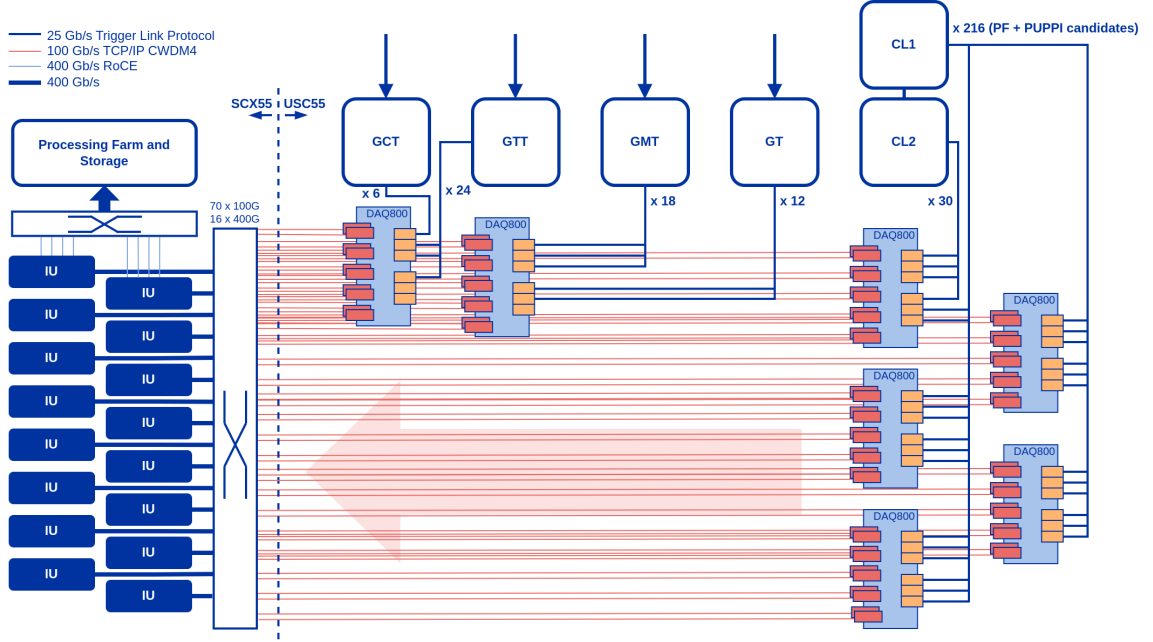


Figure 2.10: Baseline Architecture of the Phase-2 Level-1 Scouting System [44].

(DTH) ATCA board with two Xilinx KU15P FPGAs and it generates the Trigger Control and Distribution System for Phase-2 (TCDS2) clock and synchronization signal [46], as well as 6 data streams to simulate level-1 PUPPI inputs at 40 MHz. The data is further sent over 24 L1T links operating at 25 Gb/s to a Xilinx VCU128 development kit. The development kit is equipped with a VU37P FPGA and thus has approximately half of the bandwidth and FPGA resources of a DAQ800 board. It implements zero-suppression and the further transmission of level-1 PUPPI candidates via TCP/IP to a 100 Gb/s network switch. The switch connects the VCU128 outputs to 5 servers equipped with an AMD EPYC 9654, 768 GB of RAM and a 2x100 Gb/s network card. On these servers, TkEm generators producing e/γ candidates (electrons and photons) are running, and several prototype physics online analyses are being executed that are integrated into CMSSW [47]. CMSSW is a collection of software components that CMS uses to acquire, produce, process and analyze its data [48]. Since the physics performance of the scouting system improves as more online analyses can run efficiently in parallel and as more physics cases can be considered, this work investigates whether FPGA accelerator cards could serve as a useful extension to that system.

In addition to the described phase-2 demonstrator setup, there also exists a demonstrator system that operates on real L1T objects, collecting data from the current CMS Run-3 L1T. This demonstrator is already installed in the USC and connected to a surface compute cluster [49].

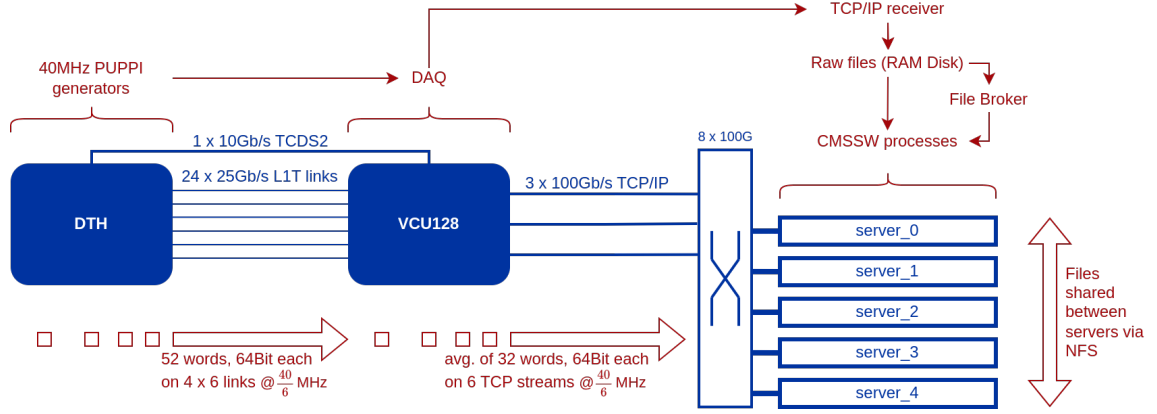


Figure 2.11: Structure of the Demonstrator System [44].

2.4.3 Physics Case

The L1DS allows for the analysis of rare and exotic signatures that would otherwise evade the traditional trigger chain. However, all computations are based on L1T objects that are optimized for triggering and not on fully reconstructed physics objects. So far, it has been successfully shown to analyze, for example, rare W decays such as the $W \rightarrow 3\pi$ and the $W \rightarrow \pi\gamma$ or rare higgs decays. The focus of this work is on the implementation of the $W \rightarrow 3\pi$ analysis on FPGA using PUPPI candidates as input data. This decay has a branching fraction of 10^{-12} to 10^{-8} with best upper limits at 10^{-6} [5]. The branching fraction describes the likelihood of the W boson decaying into three pions. The analysis steps of the $W \rightarrow 3\pi$ are briefly described below. A PUPPI candidate contains particle data like positional information η , φ and p_T ; the particle ID and the particle's charge.

$W \rightarrow 3\pi$ Analysis

For the online selection, the analysis is computed for each event by generating pairs and triplets from the input data and applying a set of cuts and filters to search for a triplet of pions that matches all criteria to be a potential decay product of the W boson.

- Particle ID:** The particle ID is derived from the Particle Data Group Identifier (PDGID) of a particle, which is a standardized integer number used to identify particle types. For the $W \rightarrow 3\pi$ analysis it must indicate a positively/negatively charged hadron or an electron/positron because charged hadrons are often misidentified as electrons/positrons.
- Isolation:** The p_T -based isolation value of a particle is computed and checked by considering all other particles in the same event with an angular distance

that is within a specified cone around that particle. The p_T sum of all particles within that cone divided by the p_T of the particle itself must not exceed a specified threshold.

Transverse Momentum: There are requirements for the p_T values of the particles eligible to form a triplet, which are checked by three different p_T cuts.

Angular Separation: The angular separation between the three potential candidates for a valid triplet needs to be larger than a predefined threshold.

Charge: The W boson has a charge of ± 1 , which is why the charge sum of all particles within a triplet needs to be ± 1 as well to fulfill charge conservation.

Invariant Mass: At the end of the selection algorithm the invariant mass of the triplet is calculated, and only those within a reasonable range from the expected W boson mass are kept.

3 Related Work

Similar to this work, Brivio et al. from the University of Milan [7], address the implementation of the $W \rightarrow 3\pi$ analysis on FPGA for the future phase-2 L1DS system. To mitigate the combinatorial complexity of forming all possible pairs and triplets from the selected candidates in each event, they adopt a strategy based on sorting the candidates by p_T . Specifically, triplets are constructed by selecting the two candidates with the highest p_T and combining them with a third candidate chosen from among the ten highest- p_T candidates, yielding eight triplet combinations per event. Subsequently, computationally intensive selection criteria—such as isolation and angular separation—are applied to identify the best triplet. However, in order to bypass these non-trivial selections and to improve physics performance, which may be degraded due to the approximations in the combinatorial strategy, the authors primarily investigate a ML-based approach to this analysis using tools such as *hls4ml* and *conifer* [50, 51]. Instead of relying on a sequence of explicit cuts, the optimal triplet is selected using either a Deep Neural Network (DNN) or a Boosted Decision Tree (BDT). Currently, the implementation comprises solely the core firmware component while the input and output infrastructure remains to be implemented.

Fraticelli et al., from the University of Colorado, Boulder [6], are developing a FPGA implementation of the $W \rightarrow 3\pi$ selection module for integration into the GTT of the CMS L1T. The module processes 18 track links as input, selects the highest p_T tracks, and computes the invariant mass of the resulting triplets. For the computation of the trigonometric functions required to calculate the invariant mass of each triplet, Look-Up Tables (LUTs) containing pre-calculated results are employed. Overall, this implementation is significantly simpler than the analysis presented in this work. This simplification is due to the fact that it should be integrated into the firmware of the GTT and therefore operates under much stricter resource and timing constraints. It utilizes approximately 11% (~47K) of the LUT, 3% (~26K) of the Flip-Flop (FF) and 1% (~43) of the Digital Signal Processing (DSP) resources of a VU9P FPGA.

Additional work exists regarding implementations of the $W \rightarrow 3\pi$ online analysis for phase-2 L1DS on various platforms. As part of his Master's thesis, Pietro Cappelli from the University of Padua [52], developed a CPU implementation of the $W \rightarrow 3\pi$ analysis, which served as the baseline implementation for this work as well. This implementation was based on prior work by Catherine Miller from the University of Boston [53] and was further developed and optimized by Giovanni Petrucciani.

Lukasz Michalski, from the Wrocław University of Science and Technology [9], developed a GPU-based implementation of the $W \rightarrow 3\pi$ analysis. This represents a portable, platform-independent implementation using the alpaka library [54, 55]. Alpaka provides functionality to develop platform-independent code that can be compiled for GPUs from all vendors as well as for CPUs. The code has been integrated into the software library of the CMS experiment, known as CMSSW [48]. The computation of the analysis is fully performed on the device and can be executed asynchronously.

In his Master's thesis, Giovanni Zago from the University of Padua [8], implemented the $W \rightarrow 3\pi$ analysis on the AI Engine (AIE) of the AMD Versal VCK5000 Development Card. In this implementation, the data is read from the host CPU, unpacked on the programmable logic of the FPGA, and subsequently transferred to the AIE array, where the complete analysis computation takes place before the data is transmitted back to the host CPU via the FPGA. The algorithm is implemented on a single tile of the AIE. An AIE tile is a small VLIW-SIMD vector processor.

	Colorado [6]	Milan [7]	AnaAccel	Zago [8]	Michalski [9]
Target System	L1T (GTT)	L1DS	L1DS	L1DS	L1DS
Platform	FPGA	FPGA	FPGA	AIE	GPU
Input Data	Track Data	PUPPI	PUPPI	PUPPI	PUPPI
Selection Approach	simpler, cut-based, p_T sorting	ML-based, p_T sorting	cut-based, full analysis	cut-based, full analysis	cut-based, full analysis

Table 3.1: Comparison of the $W \rightarrow 3\pi$ Analysis project features of the implementations from the University of Colorado, the University of Milan and the AnaAccel implementation developed in this work, as well as the implementations on AIE and GPU.

4 Hardware-Software Design

This chapter presents the core development of this work: the design and implementation of FPGA-based acceleration for particle physics analysis algorithms, and primarily the $W \rightarrow 3\pi$ analysis. It introduces the AMD Alveo U55C compute card and gives an overview of the AMD Vitis Unified Software platform used to develop and deploy the hardware design. A description of the general architecture outlining the fundamental design principles and data flow patterns follows, before the implementation details of the $W \rightarrow 3\pi$ and $W \rightarrow \pi\gamma$ analyses are presented, and the host-side implementation is discussed. The chapter concludes with an exploration of the design space optimization process, demonstrating how various architectural choices and implementation strategies were evaluated to maximize computational performance.

4.1 Target Platform

This work was conducted using an AMD Alveo U55C high performance compute card, which is connected to a server equipped with an AMD EPYC 9654 [56] via PCIe. The AMD EPYC 9654 is an x86_64 architecture CPU featuring 96 cores and operating at a base frequency of 2400 MHz with a maximum boost frequency of 3707.812 MHz. The server's operating system is *Red Hat Enterprise Linux 8.10*. To build and run the implementation on the Alveo, AMD Vitis Unified Software Platform version 2023.2 was employed on the server.

4.1.1 AMD Alveo U55C

The Alveo U55C board [57] features a Xilinx Virtex XCU55 UltraScale+ FPGA with three Super Logic Regions (SLRs), two QSFP28 ports and 16 GB of HBM (second generation). Figure 4.1 illustrates the board architecture. The following table presents a summary of the relevant resource specifications of the Alveo U55C card:

LUT	1304K
Register	2607K
DSP Slices	9024
36 Kb Block RAM (BRAM)	70.9 Mb
288 Kb Ultra RAM (URAM)	270 Mb
HBM Capacity	2x8 GB
HBM Total Bandwidth	460 GB/s
PCIe	Gen3 x16, 2xGen4 x8
PCIe Total Bandwidth	32 GB/s (bidirectional)
Network Interface	2x QSFP28
Thermal Design Power (TDP)	115 W
Total Electrical Load	75 W (no AUX power cable connected in available setup)
Technology Node	16 nm TSMC low power FinFET+ [58]
Launch Date	November 2021

The 16 GB HBM is divided into two banks of 8 GB each. HBM consists of 3D-stacked DRAM dies, and with a stack of 8 DRAM dies, 8 physical channels provide parallel access to the memory. In the current version HBM2 there are two Pseudo-Channel (PC) per physical channel, which can be accessed independently to hide latency. This results in up to 16 parallel accessible PCs per stack.

4.1.2 AMD Vitis Unified Software Platform

The Vitis Software Platform [59] is the tool stack provided by AMD for building and deploying applications on AMD hardware, including AMD FPGAs. In addition to the core development kit, it includes several components and features. The components relevant to this work are briefly introduced below.

Vitis Core Development Kit

The kit offers all basic developer tools, including compilers, analyzers, and debuggers to build accelerators, analyze performance and debug code. Besides the Command Line Interface (CLI), it also provides graphical tools and Integrated Development Environments (IDEs).

Xilinx Runtime (XRT)

The XRT runs on the host CPU and manages communication between the host and the accelerator deployed on the FPGA. Key functions include loading the accelerator bitstream onto the device, managing all data transfers between host and accelerator, and providing general board utilities such as board recovery and power management.

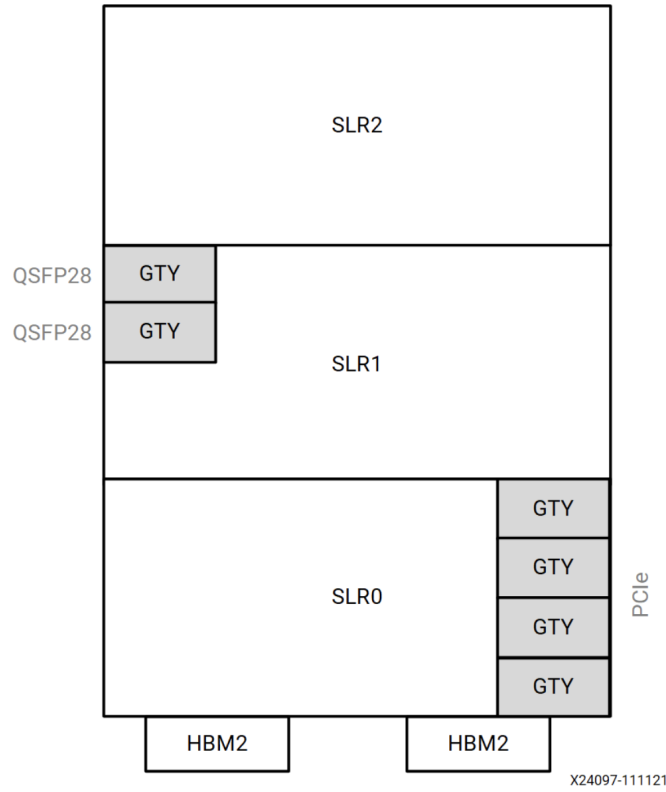


Figure 4.1: Floorplan of the Virtex XCU55 UltraScale+ FPGA [57].

Vitis Target Platforms

This component comprises a set of definitions describing the hardware and software architecture of AMD platforms. A Vitis Target Platform describes external memory interfaces, custom input/output interfaces, and the software runtime for the device. For instance, for embedded hardware it includes operating systems, boot loader, and drivers.

Vitis High Level Synthesis (HLS)

This tool performs the synthesis of C/C++ algorithms into Register Transfer Level (RTL) code. It offers functionality to leverage the advantages that HLS provides, including C simulation for early validation of the design and analysis and debugging capabilities to identify further optimizations.

4.2 General Architecture

The implementation presented in this work targets the CMS L1DS system. Its purpose is to implement the full analysis computation of the $W \rightarrow 3\pi$ decay on FPGA to investigate whether complete

physics analyses can be offloaded to an accelerator card to reduce the computational load on the CPU. Given this objective, it was evident that for the hardware-software co-design, the entire analysis calculation would be implemented on FPGA while only the providing of input data and fetching of output data would be handled by the CPU. As the Alveo U55C card offers 16 GB of HBM, the decision was made to utilize this memory and provide an implementation, that theoretically enables the CPU to offload a large amount of data to the accelerator card for asynchronous processing while the CPU remains available to execute other workloads. Thus, each Compute Unit (CU) performing the analysis is connected to at least one HBM channel for input data and one HBM channel to write back the analysis results.

On the hardware side, three major tasks required implementation: reading the input data, executing the analysis, and writing back the results. Ultimately, two different architectures were implemented and tested. Figure 4.2 illustrates the overall structure of both approaches. The first is, the one-kernel approach, in which the loading from HBM and storing back to HBM are integrated into the computation pipeline of the analysis computation within one comprehensive dataflow region. The second architecture consists of three separate kernels. In this approach, the analysis computation flow and the reading from and writing to memory is separated into kernels. While the memory accesses use the axi-memory-mapped interface (axi-mm), the analysis kernel is implemented as a free-running kernel connected to the others via axi-stream (axis).

To optimize the design, parallelism at different levels was implemented. The dataflow region enables task-level parallelism, by pipelining the different computation steps of the analysis and therefore allowing parallel execution of these functions when the pipeline is filled. Additionally, scaling up to multiple CUs on the FPGA introduces parallelism at the task level, since multiple analysis computation pipelines can run in parallel. In addition, the computation within the analysis functions is pipelined to achieve instruction-level parallelism in every computation step of the analysis. Parallelism at the data level in the form of vectorization can only be realized to a limited extent, since many computation steps of the analysis must be executed event-by-event without overlapping data between events. As the number of data elements per event can vary significantly, vectorization can only be applied to the data readout and the initial simple filters that do not depend on combinatorial tasks. Overall, the analysis implementation is not well-suited for intensive parallelization, since the calculation process contains numerous conditional branches. However, these computational characteristics motivated the investigation of this analysis as a challenging benchmark for evaluating accelerator architectures.

Since this work is a standalone implementation and is not integrated into the CMS software framework CMSSW, which provides implementations for receiving data from the upstream front-end devices and for further processing of the data after analysis, the host implementation only provides simplified functionality to supply input data and validate the results from the hardware. For this work, therefore, the role of the host-side is to read-in the input data from a dump file, pass

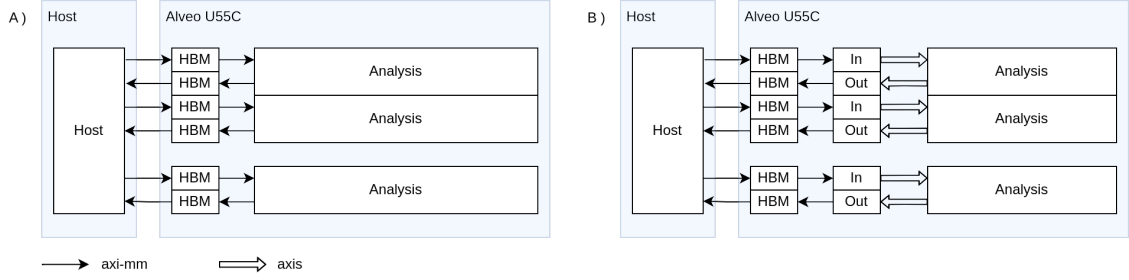


Figure 4.2: General layout of the implementation using either A) one kernel for memory reads/writes and the analysis computation, or B) three kernels separating memory reads, writes and analysis computation.

it to the device, retrieve the results from the device, and store them in a txt file. While reading the input data from file, end-of-event markers `0xffffffffffffffff` are placed between events, and the amount of input data for one HBM channel is padded to be a multiple of 512 bit to match the maximum read-out bit-width of one HBM channel and to allow vectorized memory read-outs.

4.3 Implementation of the $W \rightarrow 3\pi$ Analysis

For the online analysis of the $W \rightarrow 3\pi$ decay, the PUPPI input data is searched for a triplet of pions that matches all criteria to be a potential decay product of the W boson. The analysis consists of the following steps, which are also visualized in figure 4.3.

1. **pid-check** For the PUPPI input, each particle's ID is checked to be equal to 2, 3, 4 or 5. These IDs correspond to the PDGIDs ± 211 and ± 11 and indicate a positively/negatively charged hadron or an electron/positron, as they are often misidentified as charged hadrons. Pions, the particles this analysis searches for, are charged hadrons. An overview of the mapping of the particle's ID to PDGIDs can be seen in table 4.1.
2. **low p_T -cut** Next, the remaining particles are filtered based on their p_T values, and only candidates with $p_T \geq 7 \text{ GeV}$ are kept.
3. **isolation** For the low- p_T -cut particles, their isolation from all other PUPPI particles in the same event is determined. The isolation values must not exceed 2.0.
4. **mid p_T -cut** All pions passing the isolation check, undergo an additional p_T selection at 12 GeV .

5. **high p_T -cut** Additionally, a search is performed for high- p_T particles with $p_T \geq 15 \text{ GeV}$.
6. **angular-separation** In the next step, pairs of mid- p_T and high- p_T candidates are formed and their squared angular distance is calculated. This squared distance must be at least 0.25.
7. **charge-check** The particle pairs are then combined with the isolated low- p_T pions to triplets, and the total charge of the triplet is tested to determine whether it matches the charge of the W boson.
8. **angular-separation** Finally, the squared angular distance between the particle pair and the additional third pion is checked to also be ≥ 0.25 .
9. **triplet-mass** As a final step, the invariant mass of the triplet is calculated and required to lie within $[60, 100] \text{ GeV}$.

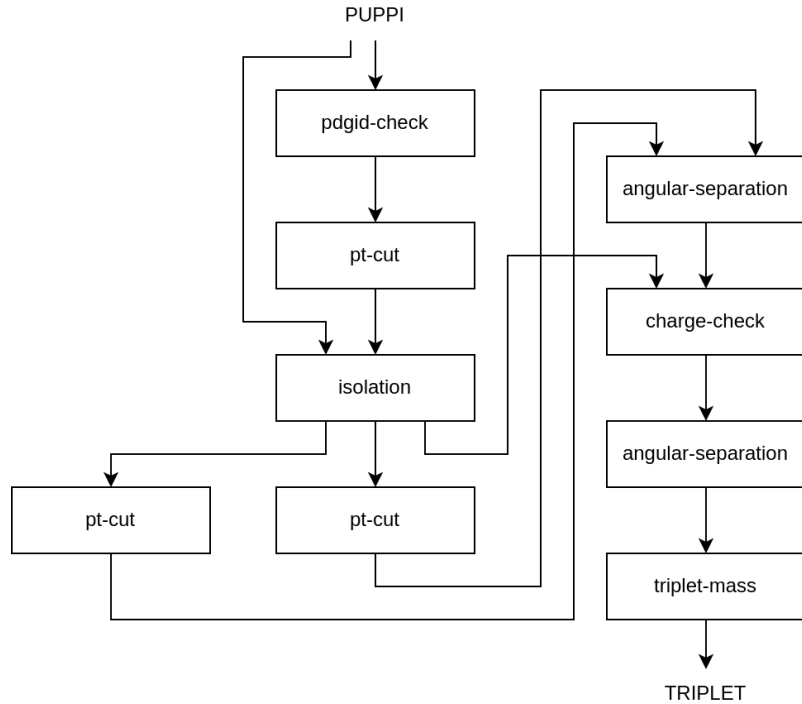


Figure 4.3: Steps of the $W \rightarrow 3\pi$ analysis.

4.3.1 Input Data

The input for this analysis implementation is PUPPI data. The data is structured event-wise. First, a 64 bit event header is read that contains information about the orbit number, the BX number

PID	PDGID	Particle
0 000	130	neutral hadron (h^0)
1 001	22	photon (γ)
2 010	-211	negatively charged hadron (h^-)
3 011	+211	positively charged hadron (h^+)
4 100	+11	electron (e^-)
5 101	-11	positron (e^+)
6 110	+13	muon (μ^-)
7 111	-13	anti-muon (μ^+)

Table 4.1: Mapping of the particle's ID (PID) to PDGID.

(one orbit consists of 3564 BXs, and one BX comprises one event) and the number of particles in the event. After the header, the particle data for that event follows. The particle information is packed into 64 bit words containing values such as pT, eta, phi, and the pid, which are necessary for the analysis, as well as quality information about the reconstruction resolution. In addition, charged particles come with track information, and neutral particles contain a value indicating the probability of originating from the primary vertex. After the header and all particle data of one event, the header and particles of the next event follow. The structure of the header and particle data is also illustrated in figure 4.4.

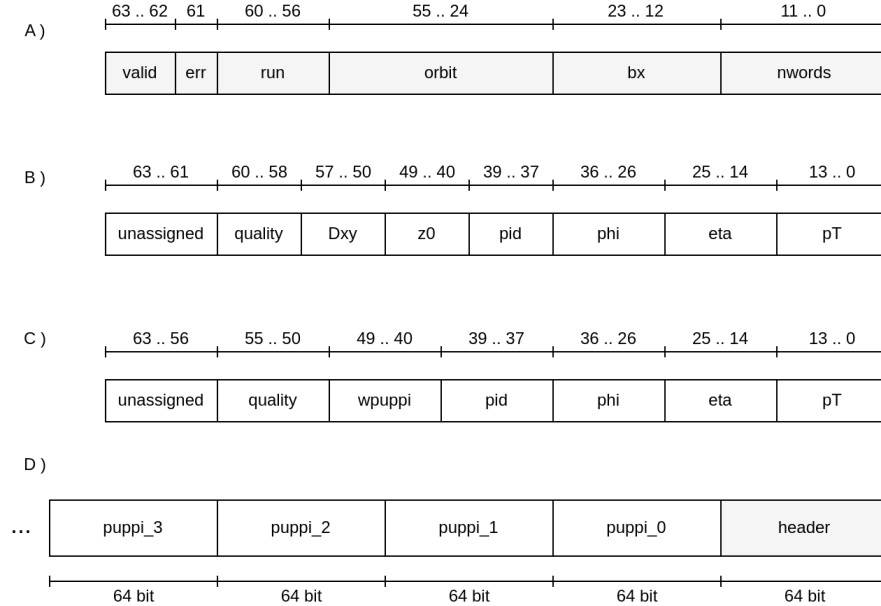


Figure 4.4: Format of the PUPPI input data with A) event header, B) charged particles and C) neutral particle; and how the data is streamed D).

4.3.2 Hardware Design

All computation steps of the $W \rightarrow 3\pi$ analysis are implemented separately as HLS functions and are connected through `hls::streams` in a dataflow region. This modular design was chosen to support reusability and simplify the implementation of further analyses. Figure 4.5 illustrates the arrangement of the computation modules and the data flowing through them. The `load_event` function reads the data from HBM and streams it to the first analysis steps. Each computation module reads the data from the stream, performs the corresponding calculations, and subsequently streams it to the next computation module. As the analysis is carried out event by event and the number of particles per event can vary from just a few to over a hundred, the data is streamed as individual particles, or, after combining, as pairs or triplets of particles. After the last analysis step, the remaining particle triplets, which have passed all selections, are written back to HBM by the `store_result` function.

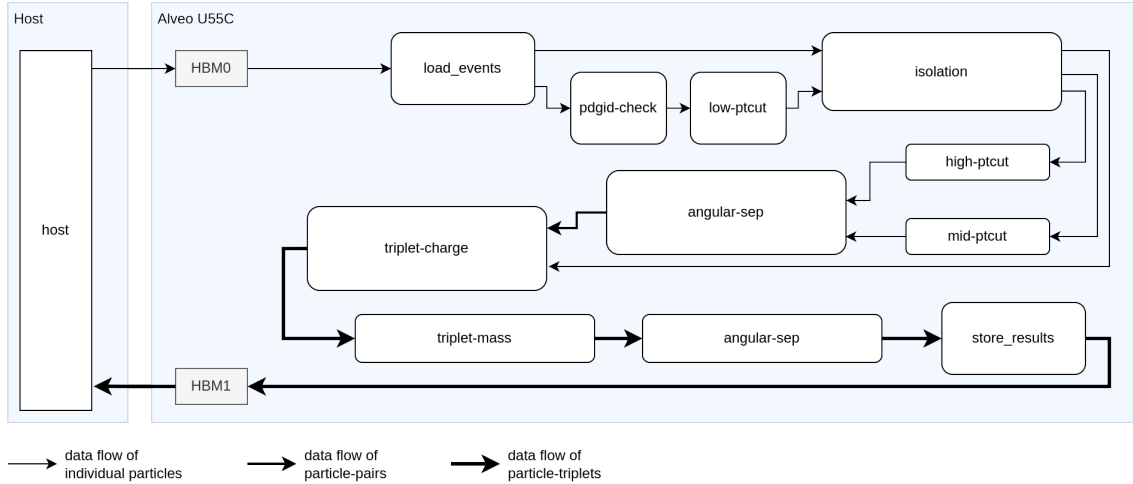


Figure 4.5: Hardware design of the $W \rightarrow 3\pi$ analysis implementation.

The order of the computation steps in this analysis can be slightly changed without violating the logic of the selection process. However, in general, it positively affects the performance of the entire design to place simple filters and cuts before the compute-heavy steps of the analysis. These are mainly computations that require the combination of particles within one event, either to form pairs or triplets of particles or to compare one particle to all others, as in the isolation computation. It is therefore advantageous to, for example, place the low- p_T cut before the isolation computation; even though the computation pipeline has one more stage, the kernel execution time is approximately 5x lower. This is also the reason for calculating the angular distance for the pairs of particles that are formed, thus further reducing the total number of candidates before they are combined to form triplets and the angular distance to the third particle is calculated again.

4.3.3 Combinatorics

The most time- and resource-intensive part of the analysis is the task of combining the particles of one event into pairs or triplets, here referred to as *combinatorics*. The complexity and data volume increase exponentially when performing this computation. The first simple design, which consisted of an array to temporarily store all particles of one event from one input to compare them with the streaming input of the other input, failed routing due to congestion. As a result, a different, more resource-saving design was implemented, which is illustrated in figure 4.6. Data is read from the two input streams, immediately combined, and written to the output stream. The data of one input stream is then written back to a FIFO queue, and the data from the other input stream is temporarily stored in an array. Next, the data is read again from the input streams, combined with each other and with all entries of the array, and again written to the queue or stored in the array. This process continues until all particle data of one event is consumed from the input streams. Then, the first entry of the array can be deleted, and one element is taken from the FIFO buffer and combined with all other data stored in the array. This implementation approach was inspired by the *Lemming Skyline* algorithm with parallel Block Nested Loops (BNL), described in “Parallel Computation of Skyline Queries” by Woods et al. [60].

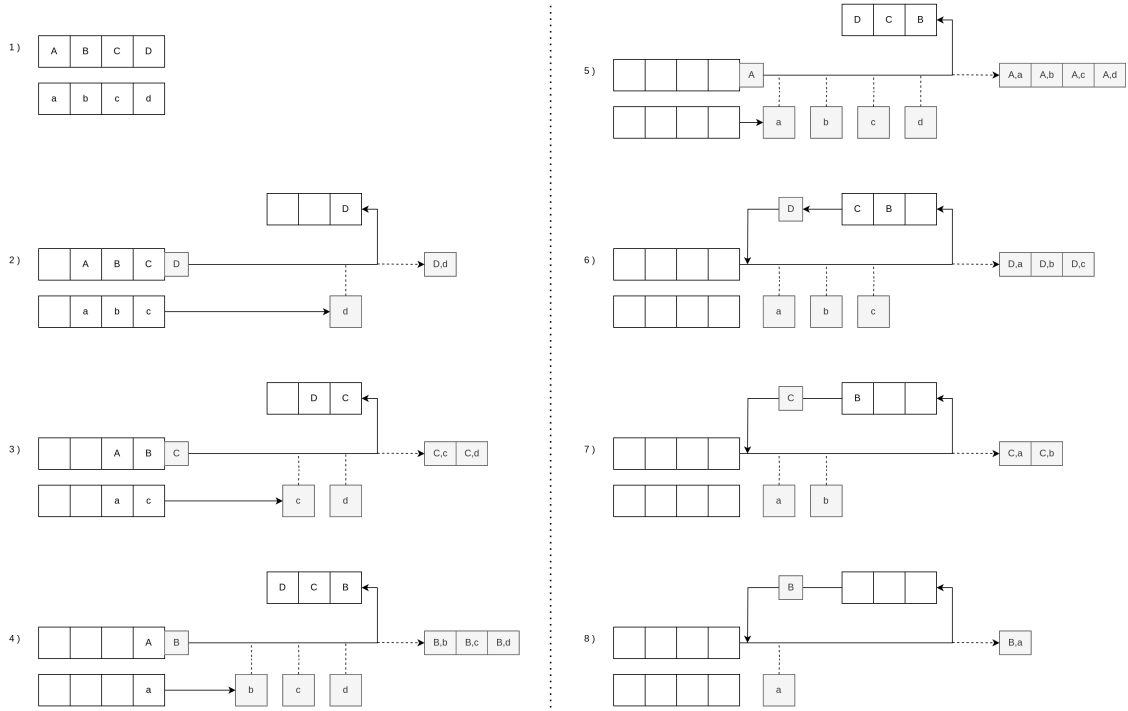


Figure 4.6: Process of combining data in the reworked combinatorics computation module.

With this implementation, the resource usage decreased significantly (FF: ↓87.5%, LUT: ↓74.8%) and the congestion problem could be solved. The Vivado device view in figure 4.7 shows the resource utilization before and after the optimization.

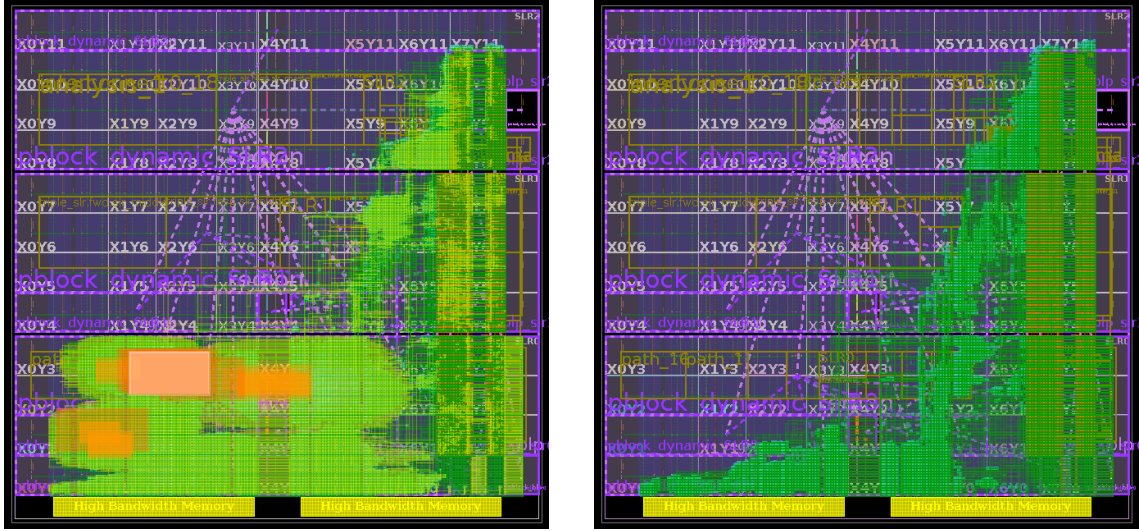


Figure 4.7: Vivado device view showing the congested areas of the old implementation (left) and the resource usage of the new (right) combinatorics module.

4.3.4 Trigonometric Functions

For the computation of the invariant mass, the trigonometric functions $\sinh()$, $\cosh()$, $\sin()$, $\cos()$ are necessary to calculate the energies as well as the x , y , z components of the momentum vectors of the particles using p_T , η , ϕ . For the implementation of the trigonometric functions, the LUT based approach from Fraticelli et al. [6] is employed. Each LUT stores 1024 pre-calculated values of the respective functions. This approach exploits the symmetry of the functions, ensuring that no absolute values are stored twice. Additionally it is considered, that in the PUPPI input data, charged objects are reconstructed with their tracks and the tracker's detector acceptance is within $|\eta| < 2.5$. Therefore, for the hyperbolic sine and cosine, only function values for $0 \leq x \leq 2.5$ are stored with the result negated as appropriate based on the sign of η . For sine and cosine, only one LUT is used, which provides data points to cover one quarter of the cosine's period. From this, the sine function results can also be obtained by shifting the corresponding index accordingly.

The implementation of Fraticelli et al. was adapted and slimmed down for this work to better suit the hardware. For example, instead of casting the eta values to float and using `std::abs()`, `hls::abs()` is now used on the fixed-point data types. Additionally, long chained computations were split into multiple steps.

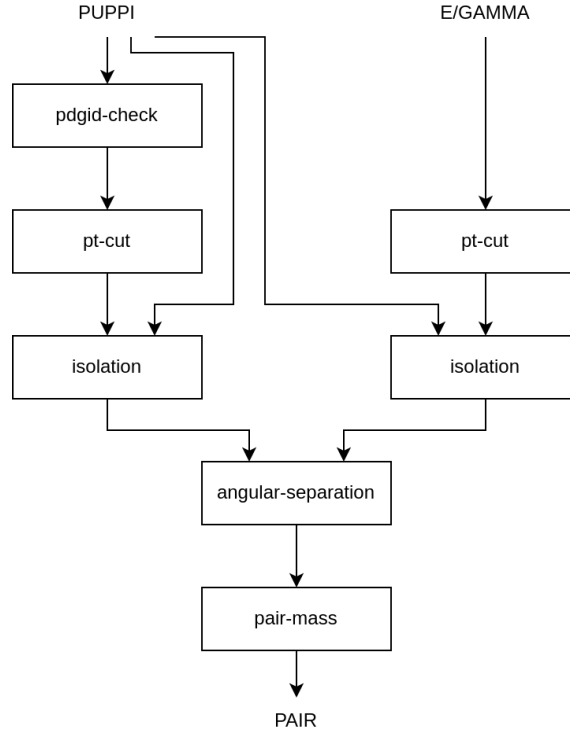
4.4 Implementation of the $W \rightarrow \pi\gamma$ Analysis

Building on the implementation of the $W \rightarrow 3\pi$ analysis, the $W \rightarrow \pi\gamma$ analysis was also ported to FPGA. In this analysis, a pair of a pion and a photon needs to be identified. The analysis consists of the following computational steps, which are illustrate in figure 4.8. For a more detailed view of the particle physics background, refer to chapter 2.

- | | |
|-----------------------|--|
| 1. pid-check | As in the $W \rightarrow 3\pi$ analysis, the particle ID of the PUPPI input data is checked to be equal to 2, 3, 4 or 5. |
| 2. gamma-ptcut | For the e/γ input, only candidates with $p_T \geq 20 \text{ GeV}$ are retained. |
| 3. pi-ptcut | The PID-checked PUPPI candidates must meet the condition $p_T \geq 25 \text{ GeV}$ to not be sorted out. |
| 4. gamma-isolation | Subsequently, the isolation of the e/γ particles from the pions is calculated. The isolation values must not exceed 0.3. |
| 5. pi-isolation | Similarly, the isolation of the pions relative to all other PUPPI particles in the same event is determined and must be smaller than or equal to 0.3. |
| 6. angular-separation | All PUPPI and e/γ particles passing the isolation requirements are combined to pairs. Their squared angular distance is computed and must be greater than or equal to 0.25. |
| 7. pair-mass | Finally, the invariant mass of the formed pairs is calculated and only pairs with a mass within $[60, 100] \text{ GeV}$ pass this selection. |

4.4.1 Input Data

A major difference compared to the $W \rightarrow 3\pi$ analysis lies in the input data. For the $W \rightarrow \pi\gamma$ analysis, PUPPI candidates serve as input along with e/γ data (electrons and photons). Compared to PUPPI data, the e/γ input is structured differently as shown in figure 4.9. Each particle consists of a 96 bit word containing p_T , η , and φ . Each event begins with a 64 bit event header containing information about the orbit and BX index and the number of particles in the event. However, the particle data section of each event always consists of 12 photons followed by (number of particles – 12) electrons. The 96 bit words are divided into a 64 bit lower part and a 32 bit upper part and are arranged in a 64 bit pattern as depicted in figure 4.9. To handle this input and enable reuse of the implemented computation modules, an adapter was developed to extract all necessary particle information and reshape it to match the PUPPI particle structure while pre-processing the data for the device.

Figure 4.8: Steps of the $W \rightarrow \pi\gamma$ analysis.

4.4.2 Hardware Design

The hardware layout of the $W \rightarrow \pi\gamma$ analysis implementation, illustrated in figure 4.10, follows the same scheme as the $W \rightarrow 3\pi$ analysis. Leveraging the modular implementation of the $W \rightarrow 3\pi$ analysis steps, computation modules could be reused, with only the thresholds of the filters and cuts requiring updates. However, the computation of the invariant mass had to be adapted to handle a system of two particles, rather than three. Unlike the $W \rightarrow 3\pi$, the $W \rightarrow \pi\gamma$ requires input from two different sources. Therefore, the hardware architecture is modified and two separate HBM channels are used to provide the data. After the the `load_events` function has read the data from memory, it is streamed to the computation modules. In the initial analysis steps, PUPPI and e/γ input undergo different selections before they are combined to pairs. Finally, the particle pairs that have passed all filters are written back to HBM. Since two input sources are required for the $W \rightarrow \pi\gamma$ analysis, three HBM channels are needed for each CU. This constraint limits the implementation to only 10 parallel CUs on the FPGA.

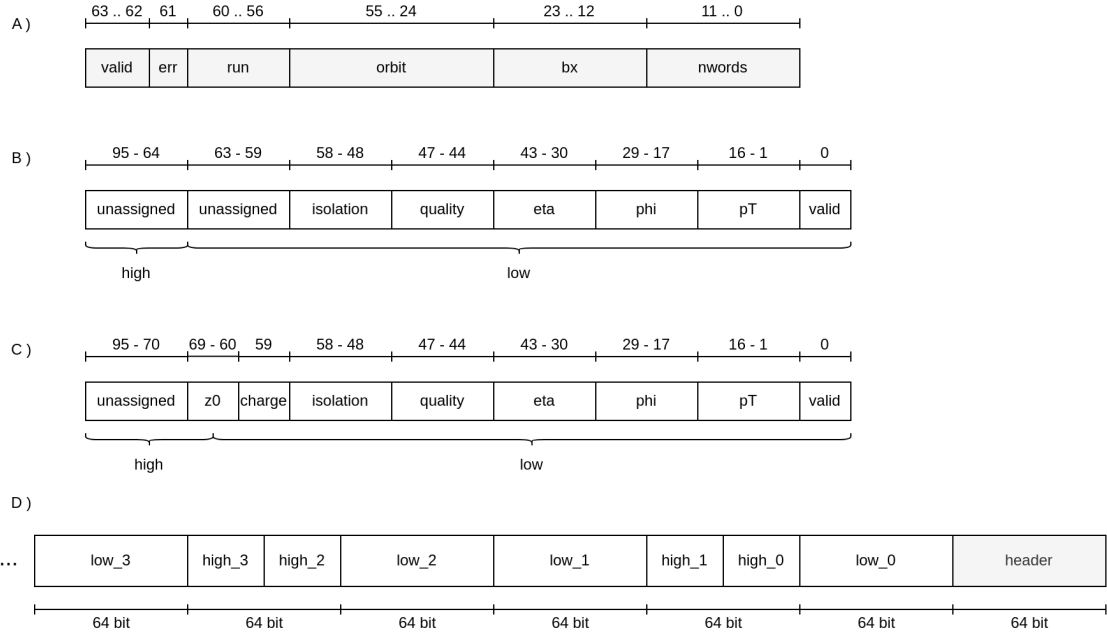


Figure 4.9: Format of the e/γ input data with A) header, B) photons (γ) and C) electrons (e); and how the data is streamed D).

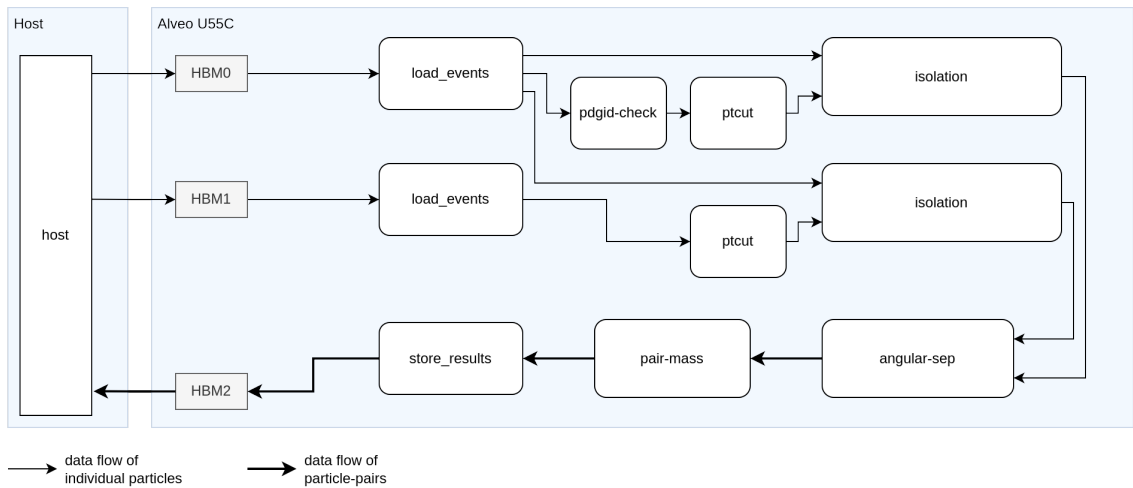


Figure 4.10: Hardware design of the $W \rightarrow \pi\gamma$ analysis implementation.

4.5 Host Implementation

The host executable acts as the main program of the entire implementation. It takes the compiled `xc1bin` bitfile of the hardware design as a command line argument and controls all interactions with the FPGA device, including loading the bitfile to the device, allocating memory on the device, transferring the data, and starting the kernel execution. Several variants of the host code were created to test different execution modes and adapt to the various versions of the hardware design that were implemented to explore the design space and optimize performance. These include the attempt to split the input data across multiple HBM channels per analysis and the scaling up to multiple parallel CU on the FPGA. Two different APIs were used to communicate to the FPGA: OpenCL and the XRT Native API.

4.5.1 OpenCL API

AMD offers the Xilinx OpenCL extension to communicate with AMD devices via XRT using the OpenCL API. It provides a specific memory extension to create buffers on the device using the `cl_mem_ext_ptr_xilinx` struct. For the majority of functionalities, standard OpenCL API objects can be used, like:

<code>cl::Device</code>	Finds and represents an accelerator device.
<code>cl::Program</code>	Loads the program from the <code>xc1bin</code> file to later extract the kernel.
<code>cl::Kernel</code>	A kernel object from the <code>xc1bin</code> file from which multiple hardware instances (CUs) can be created.
<code>cl::Buffer</code>	Represents a memory object through which all data transfers are performed.
<code>cl::Context</code>	Manages kernel execution and memory management.
<code>cl::CommandQueue</code>	A queue for submitting commands such as kernel execution or memory transfers.

4.5.2 XRT Native API

Introduced in XRT version 2020.2, AMD Xilinx offers a XRT Native API set for C/C++ and Python. It provides all functionality to find the device, load the `xc1bin` file, transfer data, and launch the kernel. The C++ class objects used in this work are listed below:

<code>xrt::device</code>	Finds a Xilinx device and loads the <code>xc1bin</code> file.
--------------------------	---

xrt::bo	Handles buffer allocation on host and device and transfers data from host to device and back using read/write, copy, or mapped memory.
xrt::kernel	A kernel object from the xclbin file from which multiple hardware instances (CUs) can be created.
xrt::run	Handles execution of a kernel object.

The range of API functionality is still limited, for instance, connecting four or more HBM channels to the kernel is not supported with the standard function structure. However, it offers a lightweight and straightforward implementation of the most important host functionalities with significantly fewer lines of code compared to OpenCL. Additionally, slightly better runtimes for the memory transfers were achieved using the XRT API functions directly compared to OpenCL, likely due to the direct XRT calls without the overhead of the OpenCL runtime. Measurements are shown in the following section 4.6.

4.6 Further Design Space Exploration and Optimization

To further improve the kernel execution time as well as the time spent on input/output operations and to achieve comparable runtimes to the GPU and AIE implementations, different optimizations of the $W \rightarrow 3\pi$ analysis implementation were explored. The whole design space exploration conducted in this work and the evolution of the kernel and i/o times can be seen in figure 4.11. The entire development can be divided into 7 phases, which are described in more detail below.

First Attempts

Versions 1 to 4 represent measurements of the first technically and functionally correct implementations running on the Alveo card. No major optimizations were applied beyond basic pipelining and unrolling. For the memory transfers, a simple ring buffer implementation was tested to allow the host to continuously write data to the device.

Memory Transfer Optimizations

The ring buffer implementation introduced a significant overhead, not only in the i/o times but also introduced more complexity to the code and increased susceptibility to errors. Due to these issues and the fact, that both reference projects on GPU and AIE used the approach of transferring data to the device, executing the analysis computation, and writing back the results, the memory transfers

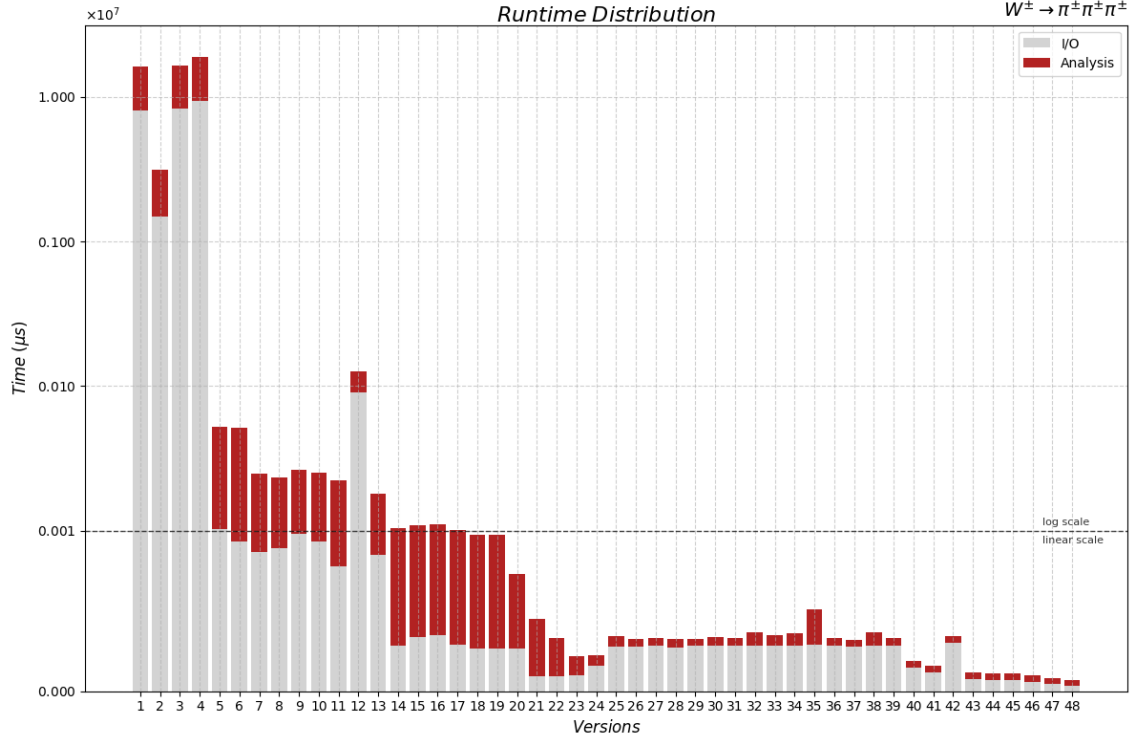


Figure 4.11: Development of the per-obit runtimes during design space exploration. The runtimes below $10^4 \mu s$ are plotted on a linear scale and above $10^4 \mu s$ on a logarithmic scale.

in this work were simplified and implemented to be able to copy a bulk of data and then start the kernel execution before fetching the results. To achieve this, the XRT Native API was employed from version 5 onwards, which led to a substantial decrease in both, kernel runtime and i/o time after version 4.

To further improve the memory accesses from the kernel side, in version 7 to 11, the HBM readout was not longer done particle-by-particle (64 bit) but in chunks to exploit the HBM bit width and enable burst transfers. This resulted in a significant improvement in the kernel runtime, reducing it by half compared to versions 5 and 6 (note the logarithmic scale for these measurements in figure 4.11).

Increasing the Data Size

In version 12, the hardware architecture was slightly modified to no longer use one HBM channel as input for the analysis computation and one as output, but instead to parallelize memory readouts and use 4 HBM channels to provide the input data for the analysis. This hardware design is

illustrated in figure 4.12. Consequently, a new merge function was added to the analysis flow to combine the input data into one stream after readout. This has led to significantly higher overhead in memory transfers, and the OpenCL API was required to create a kernel with 5 HBM channels. With this design, the data size was increased step-by-step from 32 events in version 12 to approximately 50k events in version 19 attempting to mitigate the larger i/o overhead.

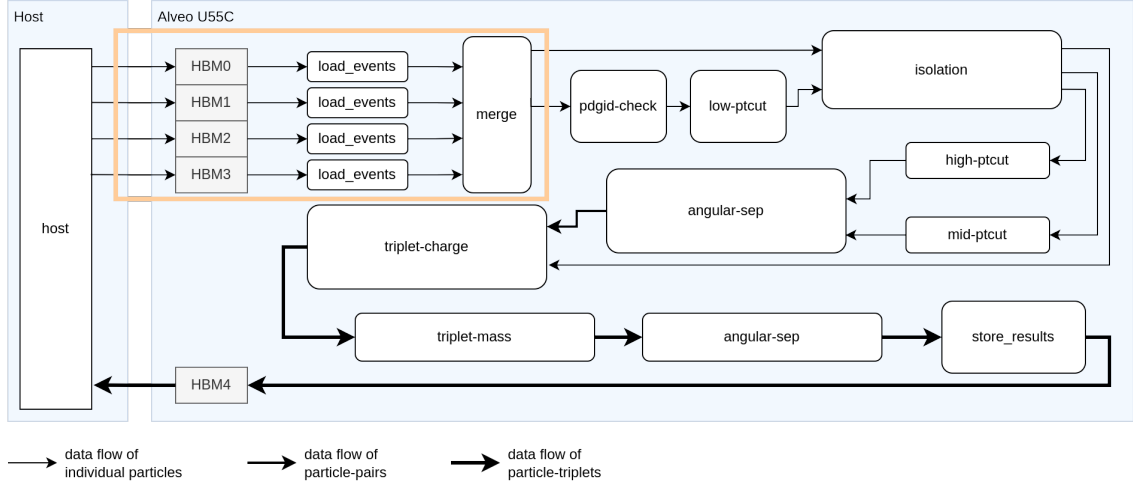


Figure 4.12: Hardware design of the $W \rightarrow 3\pi$ analysis using 5 HBM channels.

So far, all tests were performed on the `puppi_WTo3Pion_PU200.dump` signal dataset, which was generated to contain a high number of $W \rightarrow 3\pi$ decay data to test and validate the analysis implementation. However, in a real-world detector scenario, this very rare decay happens almost never. To test under more real-world conditions, the background dataset `puppi_SingleNeutrino_PU200.125X_v1.0.dump` was used from version 20 onwards. The measurements clearly show that the kernel runtime of version 20 compared to version 19 is significantly reduced, as less particles pass the filters of the analysis, and thus, the latency, especially for the combinatorial analysis steps, drops.

Upscaling to multiple CUs

The most significant speedup in the kernel execution time was achieved by scaling up the entire design to place multiple CUs in parallel on the FPGA. In versions 21 and 22, two CUs were created, each using 5 HBM channels but with version 21 operating on the signal dataset and version 22 on background data. As the next step, version 23 consists of 4 CUs with 5 HBM channels each. Doubling the number of CUs halved the kernel runtime, from 4.7 ms per orbit using one CU in version 20 to 2.4 ms and 1.2 ms per orbit placing two and 4 CUs in versions 22 and 23, respectively.

To scale up further, in version 24, only three HBM channels were used per CU to feed the analysis computation, and 8 CUs were placed on the FPGA. The kernel runtime decreased to around 0.64 ms per orbit but it is evident that the i/o time became worse again compared to versions 21 to 23. However, reducing the number of HBM channels even further, back to two channels per CU (one for input, one for output) in version 25, led to almost no change in the kernel execution time and only further increased the i/o time. However, now only 16 out of 32 HBM channels were used, and even more CUs could be placed on the FPGA. Version 26 featured 12 CUs and a kernel execution time of 0.46 ms per orbit, but when attempting to scale up to 16 CUs and utilizing all 32 HBM channels in version 27, the design exhibited significant timing violations (TNS=-9992.737), and the kernel execution time did not improve further, even when forcing Vivado to place the design exclusively within SLR 1 and 2, which are nearest to the HBM banks in version 28. Tests with 14 CUs and 15 CUs in versions 29 and 30 showed that placing 14 CUs on the FPGA might be the optimum for this hardware design.

Optimization Plateau

From versions 31 to 39, different optimizations and design variations were explored, including splitting up loops and unrolling partial loops or forcing to inline sub-functions in the mass computation. Additionally, the order of the analysis steps was changed, leading to a significantly increased kernel execution time in version 35, when placing the low- p_T cut after the isolation computation to achieve a shorter computation pipeline, as all three p_T cuts of the $W \rightarrow 3\pi$ analysis could have been executed in parallel on the output of the isolation function. However, as the low- p_T cut filters out a large number of particles, it is more beneficial to place it before the computationally expensive isolation calculation, even though the analysis computation pipeline has one additional stage, as illustrated in figure 4.13.

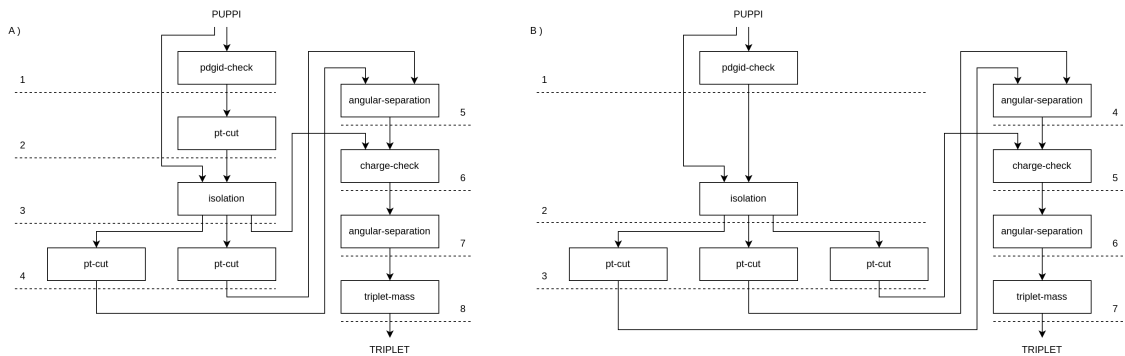


Figure 4.13: $W \rightarrow 3\pi$ analysis computation pipeline stages with A) low p_T -cut before isolation and B) low p_T -cut after isolation.

XRT API, Loop and Mass Computation Optimization

The last attempts to optimize the implementation were mainly focused on reducing the kernel execution to the point that it only makes up a small fraction of the total runtime, which is now heavily bound by i/o. As the current best design, using 14 CUs with two HBM channels each, no longer relies on the OpenCL API, the host implementation was switched to use the XRT Native API in version 40, leading to a substantial improvement in i/o time. In version 41, all `while(. . .)` loops were changed to `while(true)` loops together with a break condition inside the loop body; this can be seen in listing 4.1 in lines 9 and 27. Additionally long cascaded calculations in the mass function were split into multiple computation steps. With these optimizations, the best kernel execution time of 0.41 ms per orbit could be achieved.

```

1 void lowptcut(
2     hls::stream<PUPPI_T> &in,
3     hls::stream<PUPPI_T> &out
4 )
5 {
6     #pragma HLS pipeline II = 1
7
8     unsigned int event_count = 0;
9     while (true)
10    {
11        PUPPI_T particle_packed = in.read();
12
13        if (particle_packed == 0xffffffffffffffff) // Check end of event
14        {
15            out.write(0xffffffffffffffff);
16            event_count++;
17        }
18        else // Low pt cut
19        {
20            ap_uint<14> pt = ap_uint<14>(particle_packed & 0x3FFF);
21            if (pt >= MINPT1_HW)
22            {
23                out.write(particle_packed);
24            }
25        }
26
27        if (event_count >= (NUM_EVENTS / NUM_CU))
28            break;
29    }
30 }

```

Listing 4.1: HLS C++ function to perform the low pt-cut with a `while(true)` loop and break condition.

From version 42 onwards, the time spent on allocating the buffers is also measured and included in the i/o time. This was not done before in order to maintain comparability with the implementation on AIE, but as the optimized runtimes also reached the GPU performance, the allocation time was also considered to be comparable to the measurements on GPU. To further improve the i/o times in version 44 to 45, the allocation was optimized by no longer allocating the maximum needed

space (for each event, the maximum number of particles was assumed) but only the amount needed to store the actual size of the input data.

Increasing Data Size

Finally, in versions 46 to 48, the data size was further increased, and measurements were made on 100k events in version 46 and 400k events in version 47 which represents the maximum number of events in the available input file. By replicating the same input data multiple times, the data size was finally scaled up to 10 million events (approximately 2.6 GB) to take advantage of the available HBM capacity. The exact numbers of these measurements will be discussed in chapter 6.

5 Library Implementation

As part of this work, a small HLS library of some common physics algorithms was created based on the $W \rightarrow 3\pi$ analysis. It features an easy-to-use software abstraction layer to generate a ROOT-based software analysis and a hardware implementation in parallel. This enables the Vitis software emulation results to be compared with the results of the ROOT analysis, allowing adjustments to the analysis to be made easily. This effort demonstrates the feasibility of a library approach and could represent the first step towards a larger HLS library of common physics analysis computations. Such a library would simplify and reduce the development time for accelerating physics analysis on FPGA platforms.

5.1 Concept

The modular functions of the $W \rightarrow 3\pi$ analysis implementation were further abstracted and grouped together to form a small library of physics algorithms for FPGA. The library should especially act as an easy starting point to create and debug a physics analysis implementation for FPGA. To achieve this, two main concepts were attempted:

1. **Easy Interface:** Provision of a simple software interface to define the analysis
2. **Software Twin:** Availability of software versions of the HLS functions to create a software twin of the analysis

When it becomes possible to effortlessly define an analysis, execute it in software, and generate a first hardware implementation, the prototyping of a combined software and hardware implementation is simplified and accelerated.

5.1.1 ROOT Framework

The ROOT framework [61, 62] is an open-source data analysis framework that is widely used in high-energy physics. It allows efficient storage, processing and analysis of large amounts of data in root files. It has a C++ interpreter and a python interface and provides, together with a library of physics computations, an ideal and commonly used basis for fast prototyping of physics analyses. One typical way to elaborate a physics analysis using ROOT is through the `RDataFrame` class. In this approach, a dataframe object is constructed using, for example, a root file with physics data as input. The data is structured into columns containing different features such as a vector of p_T values of all particles within one event, or even a vector containing the 64 bit particle information of all particles within one event. The constructed dataframe object can then be transformed by applying filters to specified columns or by creating custom data columns with the results of computations that are executed on the dataset. To define a new column, the `Define()` function is called on the dataframe object. It typically takes the name of the output column that will be newly created to hold the results, a lambda expression or function that modifies the input data and produces the results, and the names of the columns that should serve as input data. Thus, a minimal example to define a physics analysis could be a sequence of `Define()` calls on a `RDataFrame` object, as illustrated in listing 5.1.

```

1 // Open ROOT file
2 ROOT::RDataFrame df(tree, rootfile);
3
4 // 1. pdgid check
5 df = df.Define("AfterPdgidcheck", sw_pdgidcheck, {"ParticleObjects"});
6
7 // 2. low pt cut
8 df = df.Define("AfterLowptcut", sw_lowptcut, {"AfterPdgidcheck"});
9
10 // 3. isolation
11 df = df.Define("AfterIsolation", sw_isolation, {"AfterLowptcut", "ParticleObjects"});
12
13 // 4. mid pt cut
14 df = df.Define("AfterMidptcut", sw_midptcut, {"AfterIsolation"});
15
16 // 5. high pt cut
17 df = df.Define("AfterHighptcut", sw_highptcut, {"AfterMidptcut"});
18
19 // 6. make pairs and check angular separation on pairs
20 df = df.Define("AfterMakepairs", sw_combine_pairs, {"AfterHighptcut", "AfterMidptcut"});
21
22 // 7. make triplets and check charge on triplets
23 df = df.Define("AfterMaketriplets", sw_combine_triplets, {"AfterLowptcut", "AfterMakepairs"});
24
25 // 8. angular separation on triplets
26 df = df.Define("AfterAngularsep", sw_angularsep_triplets, {"AfterMass"});
27
28 // 9. invariant mass test
29 df = df.Define("AfterMass", sw_mass, {"AfterMaketriplets"});

```

Listing 5.1: Minimal non-functional code example of the $W \rightarrow 3\pi$ analysis steps in ROOT.

5.1.2 Project Structure

Given that ROOT is well-established among physicists, the user interface of the library implementation developed in this work follows the ROOT syntax for defining analyses. Based on this interface, both HLS and a ROOT implementations of the analysis are generated to create not only create a hardware version but also a software twin of the analysis in ROOT. Figure 5.1 illustrates the library structure. The library provides hardware and software versions of the analysis computation steps. Using these functions, analyses can be defined step-by-step in ROOT-like syntax. From this specification, a graph representing the entire analysis flow is generated. The software analysis is then executed based on this graph using the ROOT framework, while the HLS hardware analysis entry function is generated to invoke the functions that compute the analysis steps within a dataflow region. The structure of a functional ROOT analysis and the corresponding HLS entry function for the $W \rightarrow 3\pi$ analysis are presented in appendix A.1.

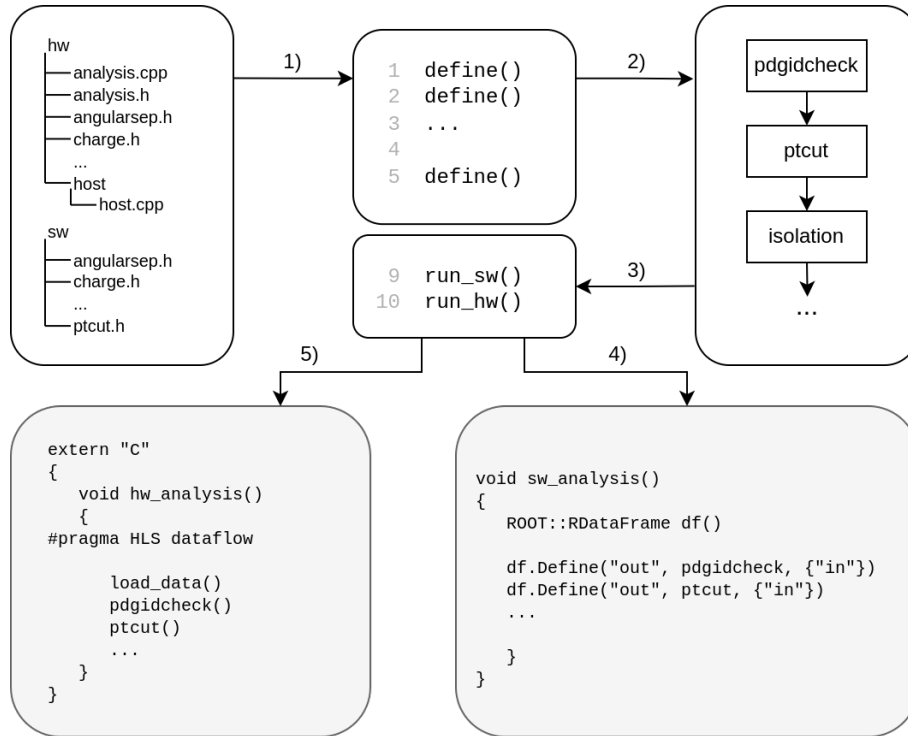


Figure 5.1: Concept of the library implementation.

5.2 Implementation

The implementation of the concepts presented previously can be divided into the realization of the user interface to provide the ROOT-like abstraction layer and the implementation of the mechanisms required to realize the execution of a ROOT-based software analysis and to generate a HLS version targeting the FPGA.

5.2.1 Interface

Based on the implementation structure, the software abstraction layer enabling the ROOT-like syntax of defining the analysis was implemented. This is realized with an `Analysis` class that provides the three functions needed to define and run the analysis: `define()`, `run_on_sw()` and `run_on_hw`. The main function demonstrating this usage can be seen in listing 5.2. An object of the `Analysis` class is created, and the analysis steps are defined on this object, similar to ROOT, by providing the name of the output data, the name of the computation module and the names of the inputs as parameters. The `define()` function creates two lists, one for the hardware and one for the software analysis. These lists represent the sequence of the analysis computation by linking to the provided functions that implement the analysis computation steps. The lists are stored within the analysis object.

```

1  int main(int argc, char *argv[])
2  {
3      Analysis a;
4
5      a.define("pdgidcheck_done", "pdgidcheck", {"input"});
6      a.define("lowptcut_done", "lowptcut", {"pdgidcheck_done"});
7      a.define("isolation_done", "isolation", {"input", "lowptcut_done"});
8      a.define("highptcut_done", "highptcut", {"isolation_done"});
9      a.define("midptcut_done", "midptcut", {"isolation_done"});
10     a.define("angularsep_done", "angularsep_makepairs", {"highptcut_done", "midptcut_done"});
11     a.define("charge_done", "charge_maketriplets", {"isolation_done", "angularsep_done"});
12     a.define("angularseptriplets_done", "angularsep_triplets", {"charge_done"});
13     a.define("mass_done", "mass_triplets", {"angularseptriplets_done"});
14
15     a.run_on_sw(ANALYSIS::W3PI);
16     a.run_on_hw(ANALYSIS::W3PI);
17 }

```

Listing 5.2: Main function to define the analysis steps.

Additional parameters of the analysis are specified in the header file `analysis_config.h`. This file defines a set of preprocessor macros that set general parameters including the number of events the analysis should process, the cuts and filter thresholds used in the analysis, information about the input dump and root files, and device parameters such as the number of HBM channels the device has.

5.2.2 Software Analysis

The function `run_on_sw()` processes the root file and first defines vectors of particle data, to match the input requirements of the software analysis functions. Subsequently, the function iterates through the generated analysis list and calls the ROOT `Define()` method for each analysis step passing the provided analysis computation functions that were added to the list when defining the analysis. Finally, it outputs the analysis results to a txt file. These software twins of the hardware analysis functions are identical to the corresponding hardware versions. They are as similar as possible in their input and output data structures. However, since the software analysis works on a root file, the calculations within these functions are more similar to ROOT-style analysis approaches, also using ROOT library functions. A comparison of the angular-separation computation in software and on hardware is shown in listing 5.3 and 5.4.

```

1  std::vector<SW_TRIPLET_T> sw_angularsep_triplets(
2      std::vector<SW_TRIPLET_T> &in)
3  {
4      std::vector<SW_TRIPLET_T> triplets;
5      for (const auto &triplet : in)
6      {
7          float deta_0 = std::get<0>(std::get<0>(triplet)) - std::get<0>(std::get<2>(triplet));
8          float dphi_0 = ROOT::VecOps::DeltaPhi<float>(std::get<1>(std::get<0>(triplet)),
9              ↪ std::get<1>(std::get<2>(triplet)));
10         float dr2_0 = deta_0 * deta_0 + dphi_0 * dphi_0;
11
12         float deta_1 = std::get<0>(std::get<1>(triplet)) - std::get<0>(std::get<2>(triplet));
13         float dphi_0 = ROOT::VecOps::DeltaPhi<float>(std::get<1>(std::get<1>(triplet)),
14             ↪ std::get<1>(std::get<2>(triplet)));
15         float dr2_1 = deta_1 * deta_1 + dphi_1 * dphi_1;
16
17         if ((dr2_0 >= MINDELTA2_SW) && (dr2_1 >= MINDELTA2_SW))
18             triplets.emplace_back(triplet);
19     }
20     return triplets;
21 }

```

Listing 5.3: Function computing the angular-separation computation in software.

5.2.3 Hardware Analysis

Using the analysis list, the function `run_on_hw()` generates an HLS entry function by textually placing the code into an empty cpp file. The generated file for the $W \rightarrow 3\pi$ analysis is shown in listing 5.5. The function iterates over the analysis list multiple times. It first extracts information about the input and output data of the individual analysis steps to create the corresponding `hls::streams` that connect the analysis computation modules. One challenge was that the `Define()` function in ROOT only accepts one output parameter and so does the `define()` function in this implementation. However, in hardware implementations, to reuse the same output data for different analysis

```

1 void hw_angularsep_triplets(
2     hls::stream<HW_TRIPLET_T> &in0,
3     hls::stream<HW_TRIPLET_T> &out
4 )
5 {
6     #pragma HLS pipeline II = 1
7
8     unsigned int event_count = 0;
9     while (true)
10    {
11        HW_TRIPLET_T triplet_packed = in0.read();
12
13        if (triplet_packed.first == 0xffffffffffffffff) // check end of event
14        {
15            out.write(triplet_packed);
16            event_count++;
17        }
18        else // angularsep
19        {
20            ap_int<12> eta1 = (triplet_packed.first >> 14) & 0xfff;
21            ap_int<12> eta2 = (triplet_packed.second >> 14) & 0xfff;
22            ap_int<12> eta3 = (triplet_packed.third >> 14) & 0xfff;
23            ap_int<11> phi1 = (triplet_packed.first >> 26) & 0x7ff;
24            ap_int<11> phi2 = (triplet_packed.second >> 26) & 0x7ff;
25            ap_int<11> phi3 = (triplet_packed.third >> 26) & 0x7ff;
26
27            ap_int<12> deta_0 = eta3 - eta1;
28            ap_int<11> dphi_0 = deltaPhi(phi3, phi1);
29            ap_int<24> dr2_0 = deta_0 * deta_0 + dphi_0 * dphi_0;
30            ap_int<12> deta_1 = eta3 - eta2;
31            ap_int<11> dphi_1 = deltaPhi(phi3, phi2);
32            ap_int<24> dr2_1 = deta_1 * deta_1 + dphi_1 * dphi_1;
33
34            if ((dr2_0 >= MINDELTA2) && (dr2_1 >= MINDELTA2))
35                out.write(triplet_packed);
36        }
37
38        if (event_count >= (NUM_EVENTS / NUM_CU))
39            break;
40    }
41 }

```

Listing 5.4: Function computing the angular-separation computation on hardware.

steps, the data stream needs to be replicated. Therefore, output data that serves as input data for several other analysis steps, must first pass through a `replicate()` function on hardware. This function is inserted into the list of analysis steps. After this preprocessing step to adapt the analysis list so that it matches the hardware computation flow, calls to the provided hardware analysis functions are inserted into the HLS entry function file. The compilation and execution of the software emulation and hardware emulation, as well as the synthesis and execution on the target hardware can be started using the provided Makefile.

```

1  extern "C"
2  {
3      void analysis(const HW_OBJ_T *in, HW_MASS_TRIPLET_T *out)
4      {
5          #pragma HLS interface m_axi port=in offset=slave bundle=buf_in depth=DEPTH_HBM
6          #pragma HLS interface m_axi port=out offset=slave bundle=buf_out depth=DEPTH_HBM
7          #pragma HLS interface s_axilite port=in
8          #pragma HLS interface s_axilite port=out
9          #pragma HLS interface s_axilite port=return
10
11          hls::stream<HW_OBJ_T> obj_stream[11];
12          hls::stream<HW_PAIR_T> pair_stream[1];
13          hls::stream<HW_TRIPLET_T> triplet_stream[2];
14          hls::stream<HW_MASS_TRIPLET_T> mass_triplet_stream[1];
15
16          #pragma HLS dataflow
17
18          load_data(in, obj_stream[0]);
19
20          replicate_2(obj_stream[0], obj_stream[1], obj_stream[2]);
21
22          pdgidcheck(obj_stream[1], obj_stream[3]);
23
24          lowptcut(obj_stream[3], obj_stream[4]);
25
26          isolation(obj_stream[2], obj_stream[4], obj_stream[5]);
27
28          replicate_3(obj_stream[5], obj_stream[6], obj_stream[7], obj_stream[8]);
29
30          highptcut(obj_stream[6], obj_stream[9]);
31
32          midptcut(obj_stream[7], obj_stream[10]);
33
34          angularsep_makepairs(obj_stream[9], obj_stream[10], pair_stream[0]);
35
36          charge_maketriplets(obj_stream[8], pair_stream[0], triplet_stream[0]);
37
38          angularsep_triplets(triplet_stream[0], triplet_stream[1]);
39
40          mass_triplets(triplet_stream[1], mass_triplet_stream[0]);
41
42          store_data(mass_triplet_stream[0], out);
43      }
44 }

```

Listing 5.5: Generated HLS C++ entry function of the analysis computation.

6 Results and Evaluation

The $W \rightarrow 3\pi$ analysis was implemented and evaluated across three different hardware platforms to assess the relative performance characteristics and suitability for high-energy physics data processing applications. Accordingly, both the physics performance - demonstrating that the algorithm produces meaningful physics results - and the computational performance were evaluated and compared among each other. In particular, the execution time, the resource utilization and the power consumption were considered. The CPU and GPU measurements were provided by Michalski [9], while the AIE implementation results were obtained from measurements conducted by Zago [8]. Further measurements and discussion of the results related to the $W \rightarrow \pi\gamma$ analysis are presented at the end of each section.

6.1 Testing Environment

The time measurements on FPGA were done using `std::chrono::high_resolution_clock::now()` in the host code as shown in listing 6.1. The same approach was also implemented for the measurements on the other hardware platforms.

```
1 // Run kernel
2 std::cout << "Run kernel" << std::endl;
3 std::array<xrt::run, NUM_CU> run;
4 auto kernel_start = std::chrono::high_resolution_clock::now();
5 for (unsigned int i = 0; i < NUM_CU; i++)
6 {
7     run[i] = analysis_kernel[i](boIn[i], boOut[i]);
8 }
9 for (unsigned int i = 0; i < NUM_CU; i++)
10 {
11     run[i].wait();
12 }
13 auto kernel_end = std::chrono::high_resolution_clock::now();
```

Listing 6.1: Kernel runtime measurement of the single kernel implementation.

6.1.1 Datasets

All tests were conducted using the datasets listed in table 6.1. These datasets contain simulated event data from the phase-2 CMS detector. The signal dataset for the $W \rightarrow 3\pi$ analysis consists of data explicitly emulating the $W \rightarrow 3\pi$ decay and therefore contains significantly more suitable pion triplets than usual, given that the $W \rightarrow 3\pi$ decay is extremely rare. The background dataset comprises simulated data from a *neutrino gun* experiment designed to generate single neutrinos. Using background data without artificially generated $W \rightarrow 3\pi$ events as input provides a more realistic way of running the analysis. The $W \rightarrow \pi\gamma$ analysis requires two distinct input datasets for both, signal and background data: one for the PUPPI candidates and one providing the e/γ input. For the PUPPI input for the $W \rightarrow \pi\gamma$ analysis, the same background dataset was used as for the $W \rightarrow 3\pi$ analysis.

Dataset	Details
Signal W3Pi	Binary: puppi_WTo3Pion_PU200.dump Size: 12585480 B Events: ~50k (approx. 14 orbits)
Signal WPiGamma	Binary: puppi_WPiGamma_PU200.dump Size: 5777224 B Events: ~20k (approx. 6 orbits)
Signal WPiGamma	Binary: egamma_WPiGamma_PU200.dump Size: 3518112 B Events: ~20k (approx. 6 orbits)
Background	Binary: puppi_SingleNeutrino_PU200.125X_v1.0.dump Size: 109739592 B Events: ~400k (approx. 112 orbits)
Background	Binary: egamma_SingleNeutrino_PU200.125X_v1.0.dump Size: 61708344 B Events: ~400k (approx. 112 orbits)

Table 6.1: Details of the datasets used.

6.1.2 Hardware Devices

All measurements on FPGA that will be discussed in the following sections were conducted on the Alveo U55C accelerator card introduced in chapter 4.1. The performance of the FPGA implementation will also be compared with CPU, GPU and AIE implementations. For these comparative evaluations, the following devices were employed.

CPU

The server CPU **AMD EPYC 9654** [56, 63] was used for the measurements of the CPU implementations tested. This processor features the following specifications:

Cores	96
L1 cache	64 KB per core
L2 cache	1 MB
L3 cache	384 MB
DDR5 Capacity	12x 64 GB
DDR5 Total Bandwidth	460.8 GB/s
Frequency	2.4 GHz (up to 3.7 GHz)
TDP	360 W
Technology Node	5 nm TSMC
Launch Date	November 2022

GPU

Two Nvidia GPUs were used to evaluate the performance of the GPU implementation of the $W \rightarrow 3\pi$ analysis: the **Nvidia L4** GPU with Ada Lovelace architecture [64, 65] and the **Nvidia T4** with the Tesla Turing architecture [66, 67]. These devices provide the following specifications:

	Nvidia L4	Nvidia T4
Streaming Multiprocessors (SMs)	60	40
Cuda Cores	7424	2560
Tensor Cores	240	320
L1 cache	128 KB per SM	64 KB per SM
L2 cache	48 MB	4 MB
GDDR6 Capacity	24 GB	16 GB
GDDR6 Total Bandwidth	300 GB/s	320 GB/s
PCIe	Gen4 x16	Gen3 x16
PCIe Total Bandwidth	64 GB/s (bidirectional)	32 GB/s (bidirectional)
Frequency	795 MHz (up to 2040 MHz)	585 MHz (up to 1590 MHz)
TDP	72 W	70 W
Technology Node	5 nm TSMC	12 nm TSMC
Launch Date	March 2023	October 2018

AI Engine

The **AMD VCK5000** Versal Development Card [68, 69] features a VC1902 SoC that includes 400 AIE tiles, 1968 DSP engines and an FPGA fabric all connected via a network-on-chip. It is equipped with 16 GB DDR memory and connected to the host computer via PCIe. Detailed specification are listed below:

AIE Tiles	400
AIE Memory Blocks	100 Mb
LUT	899k
Register	1799k
DSP Slices	1968
36 Kb BRAM	34 Mb
288 Kb URAM	130.2 Mb
DDR Capacity	16 GB
DDR Bandwidth	102.4 GB/s
PCIe	Gen3 x16, 2xGen4 x8
PCIe Total Bandwidth	32 GB/s (bidirectional)
Network Interface	2x QSFP28
TDP	225 W
Technology Node	7 nm TSMC
Launch Date	March 2022

6.2 Physics Performance

To assess the physics performance of the implemented FPGA version of the $W \rightarrow 3\pi$ analysis, the computed invariant mass of the triplets selected by the algorithm is plotted in a mass histogram, as shown in figure 6.1. The figure illustrates the distribution of the invariant mass of the particle triplets selected by the analysis on FPGA and compares it with the reference software implementation on CPU and the results of the Vitis software emulation of the FPGA design. As expected, the plot shows identical results on the Alveo card and in the software emulation. In comparison with the software implementation of the algorithm running on CPU, minor deviations are observed, resulting from differences in the implementation between CPU and FPGA platforms. These differences also lead to a small efficiency decrease, which is discussed in greater detail below. However, despite these minor deviations, the mass peak around 80 GeV, which corresponds to the mass of a W boson, is clearly visible in the histogram. This demonstrates sufficient correctness of the algorithm, especially for an online selection.

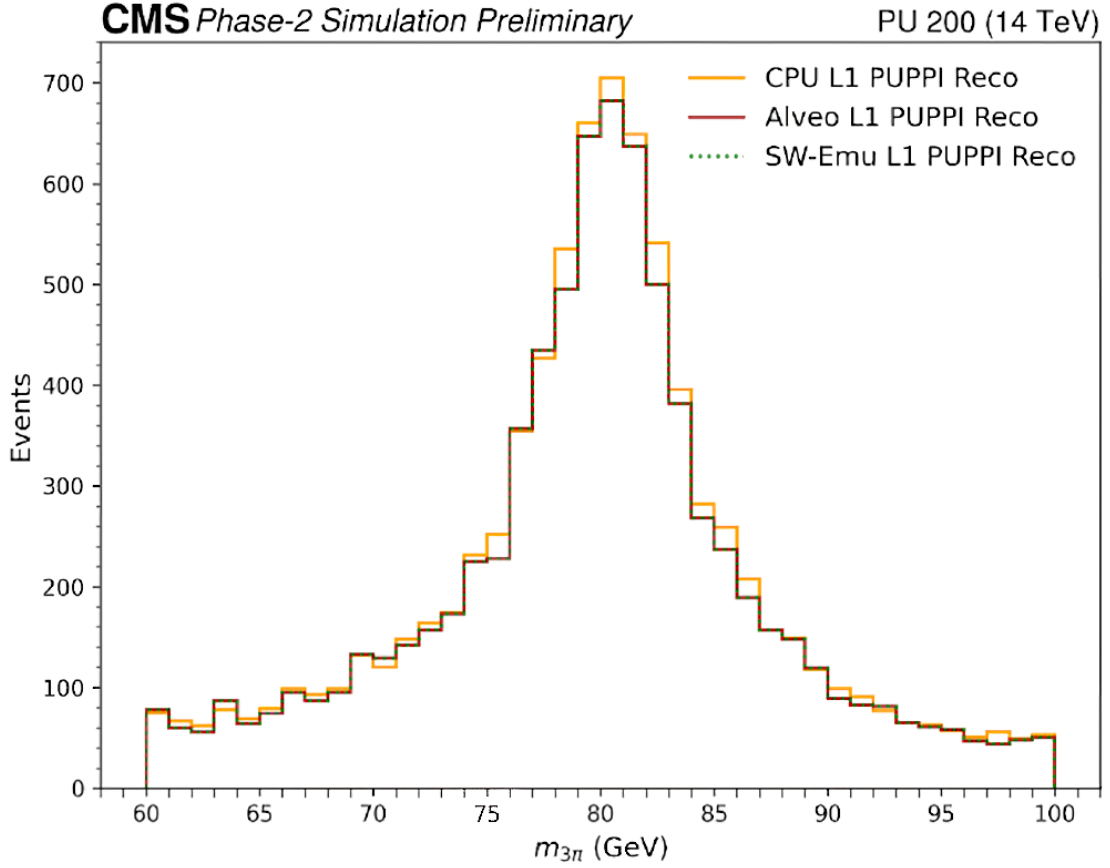


Figure 6.1: Invariant mass distribution of the $W \rightarrow 3\pi$ analysis compared to the reference implementation on CPU and with the Vitis software emulation results.

Another parameter for evaluating the physics performance is the aforementioned efficiency of an analysis implementation, which indicates the percentage of events selected by the analysis algorithm. Table 6.2 shows the efficiencies of the different implementations of the algorithm targeting CPU, GPU, AIE and FPGA. The CPU implementation serves as a reference value in the comparison. The GPU implementation achieves the same efficiency as the CPU implementation. Both hardware implementations, on AIE and on FPGA, show slight efficiency decreases on the signal dataset: 0.5% on the AIE and 0.69% on the FPGA. These deviations are likely caused by approximations in the computation of the analysis steps that were introduced to better suit the hardware. For example, all calculations on FPGA are performed using fixed-point representation instead of floating-point arithmetic used on CPU and GPU, and the results of the trigonometric functions are estimated using LUTs. For background data however, a low selection rate is preferred. This is because the $W \rightarrow 3\pi$ is an extremely rare decay so that a high selection rate mostly chooses false-positive events that will be discarded after a finer offline analysis. The FPGA implementation

developed in this work performs well in this regard, selecting 24 events out of 14 orbits (~50k events) of test data and thus achieving a selection rate of 15 kHz.

Dataset	CPU	GPU	AIE	FPGA
Signal	12.61%	12.61%	12.01%	11.92%
Background	18 kHz	18 kHz	48 kHz (76 events out of 14 orbits)	15 kHz (24 events out of 14 orbits)

Table 6.2: Efficiency of the $W \rightarrow 3\pi$ analysis on different hardware architectures.

$W \rightarrow \pi\gamma$ Analysis

For the FPGA implementation of the $W \rightarrow \pi\gamma$ analysis, the histogram of the invariant mass of the selected pion and photon pair is shown in figure 6.2. Compared to the results of the reference computation on CPU, slightly larger deviations are observed. However, the peak at the mass of the W boson around 80 GeV remains clearly visible, making the implementation sufficiently good for an online analysis serving as a preselection. The efficiency of 13.61% shows only a 0.55% difference from the software implementation on signal data. However, the selection rate on background data of 27 kHz and 42 selected events out of 14 orbits is relatively high and leaves room for improvement. The standalone software implementation of the $W \rightarrow \pi\gamma$ analysis to run on CPU was kindly provided by Cécile Caillol (CERN). Table 6.3 provides an additional overview of the efficiencies discussed above.

Dataset	CPU	FPGA
Signal	12.61%	13.16%
Background	0.2 kHz	27 kHz (42 events out of 14 orbits)

Table 6.3: Efficiency of the $W \rightarrow \pi\gamma$ analysis on CPU and FPGA.

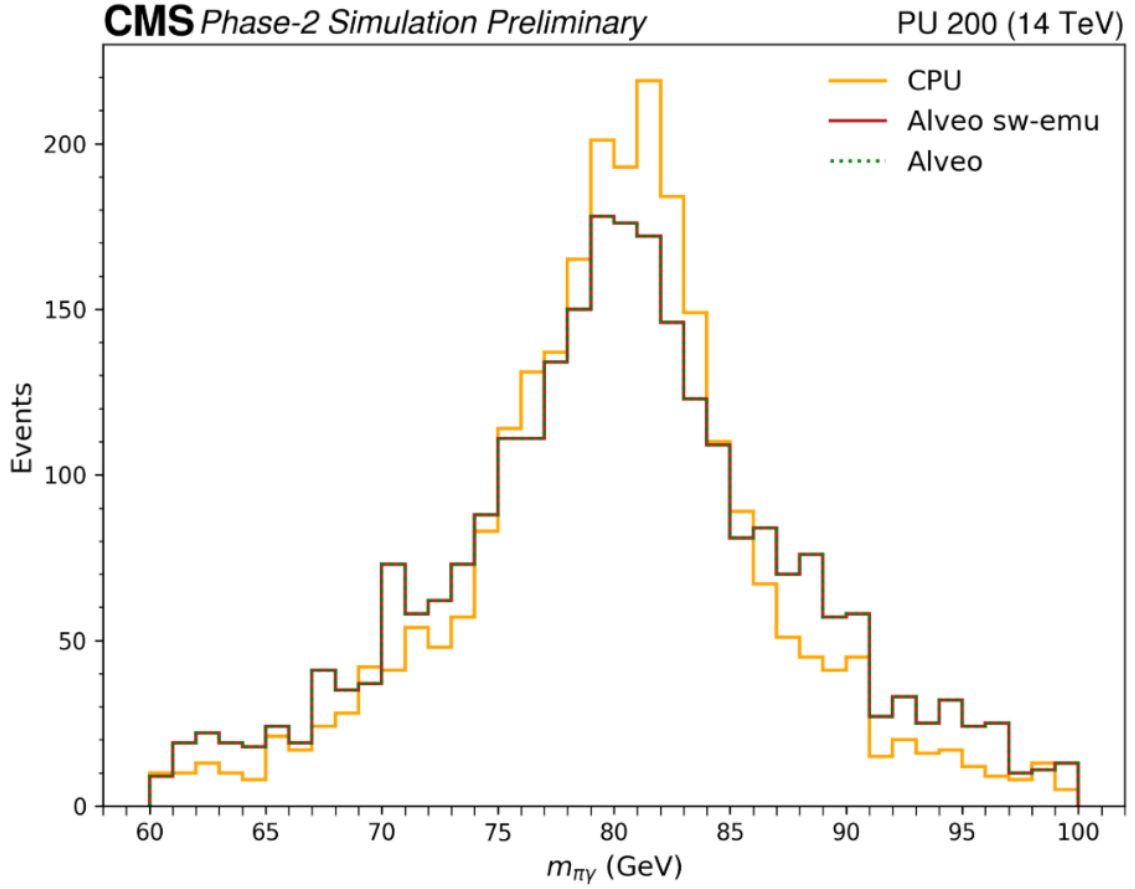


Figure 6.2: Invariant mass distribution of the $W \rightarrow \pi\gamma$ compared to the reference implementation on CPU and with the Vitis software emulation results.

6.3 Computational Benchmarking

The computational performance of the $W \rightarrow 3\pi$ analysis implementation is evaluated based on measurements of the runtime, which is decomposed into analysis kernel execution time and i/o times, as well as resource utilization, and a power consumption comparison. These tests are run at a frequency of 300 MHz for the FPGA design on the Alveo U55C, and at a frequency of 360 MHz for the programmable logic of the VCK5000 and approximately 1.2 GHz for the AIE. The frequencies of the CPU and GPUs utilized for the tests are listed in section 6.1.2, which provides a detailed overview of the hardware employed in the tests.

6.3.1 Runtime Comparison

Table 6.4 presents the execution times for each platform. It shows the total runtime and the proportion of this time spent for i/o operations and the core analysis computations. Since the CPU and GPU implementations are integrated into the CMS software library CMSSW and not standalone implementation like those for FPGA and AIE, the measurements in the table were performed orbit-wise. For the accelerator cards this means that memory buffers for one orbit (3564 events, approximately 940 KB) of data are allocated, the data is transferred to the device, then the analysis computation runs on that data on the device, and finally the results are transferred back. All operations are executed synchronized. The measurements on FPGA and AIE are averaged over 14 orbits and on GPU and CPU over 1000 orbits.

Although the FPGA implementation also supports orbit-wise execution, it was designed as a standalone system and is therefore not optimized to process orbit after orbit. Consequently, an additional measurement was conducted in *bulk processing* mode in which 10 million events (approximately 2.64 GB) are offloaded to the device and computed in a single batch before the results are transferred back. For this scenario table 6.4 shows the measured runtimes divided by the number of orbits to enable comparison with the other per-orbit runtimes. Additionally, due to limitation introduced by the CMSSW integration the CPU implementation, while optimized, is executed serially.

Hardware	Total Runtime (ms)	I/O ¹ (ms)	Analysis (ms)
CPU baseline (AMD EPYC 9654) [52]	3.1	2.5	0.59
CPU alpaka (AMD EPYC 9654) [9]	1.4	0.74	0.65
GPU (Nvidia L4) [9]	0.75	0.66	0.089
GPU (Nvidia T4) [9]	1.0	0.86	0.14
AIE (AMD VCK5000) ² [8]	8.1	1.0	7.1
FPGA (AMD Alveo U55C)	1.5	1.0	0.54
FPGA bulk processing (AMD Alveo U55C)	0.72	0.34	0.38

¹incl. I/O, memory buffer allocations and preprocessing

²memory buffer allocation not measured and preprocessing time included in analysis kernel runtime

Table 6.4: Execution times (per orbit) of the $W \rightarrow 3\pi$ analysis on different hardware devices.

Considering the total runtime, which includes both, the analysis execution and the time spent on i/o operations, the FPGA bulk processing version achieved the best overall timing result with 0.72 ms per orbit. The bulk transfer of 10 million events significantly improves the i/o efficiency, as it achieves approximately 12.2 GB/s for the input data transfer, which best saturates the available PCIe bandwidth of 16 GB/s in one direction, and the API overhead for memory transfers can be

reduced to a minimum compared to multiple small memory transfers. Moreover, the runtime of the FPGA analysis kernel benefits significantly from the bulk processing (0.38 ms vs. 0.54 ms) because the host-device communication overhead (e.g. to launch the kernel) is amortized as the data size increases.

However, the best analysis runtime is achieved by the GPU implementation on the Nvidia L4, closely followed by the measurements on the Nvidia T4. These measurements are also visualized in figure 6.3. The measurements also clearly indicate, that the Nvidia L4 is the most advanced hardware platform, featuring the newer PCIe Gen4 and 60 SMs compared to 40 on the Nvidia T4. The Nvidia L4 was launched in 2023 using a 5 nm manufacturing process and is therefore the most recent and advanced device, also in comparison to the Alveo U55C (16 nm) or the VCK5000 (7 nm). The analysis runtime of the orbit-wise FPGA implementation is slightly better than both CPU implementations - the baseline implementation and the alpaka version - and significantly faster than the implementation on AIE, which suffers from a lack of parallelization and utilized only a small fraction of the available resources of the VCK5000 card.

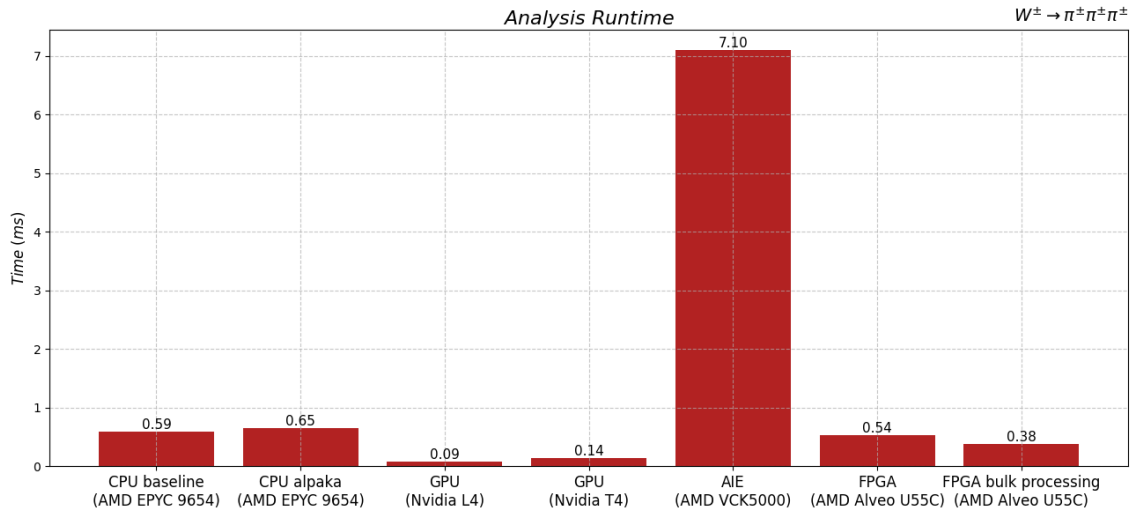


Figure 6.3: Analysis runtime measurements on CPU, GPU, AIE and FPGA.

The i/o times are relatively similar across all devices. This can be seen in figure 6.4. However, a direct comparison cannot be made because of different measuring scopes. The CPU and GPU implementations restructure the incoming event data for optimized processing on the hardware, such as converting to Structure of Arrays (SoAs). The time for this preprocessing is part of the measured i/o time. Inefficient preprocessing is also the reason for the high i/o times in the CPU baseline implementation. The preprocessing is significantly improved with the optimized alpaka version. In contrast, the FPGA implementation performs no complex preprocessing as the event data is streamed through the entire analysis computation which also provides the possibility for

a standalone implementation earlier in the L1DS pipeline. Furthermore, the AIE version doesn't include the time for memory buffer allocation to the measurements and also the preprocessing is performed on the device and is thus part of the analysis runtime.

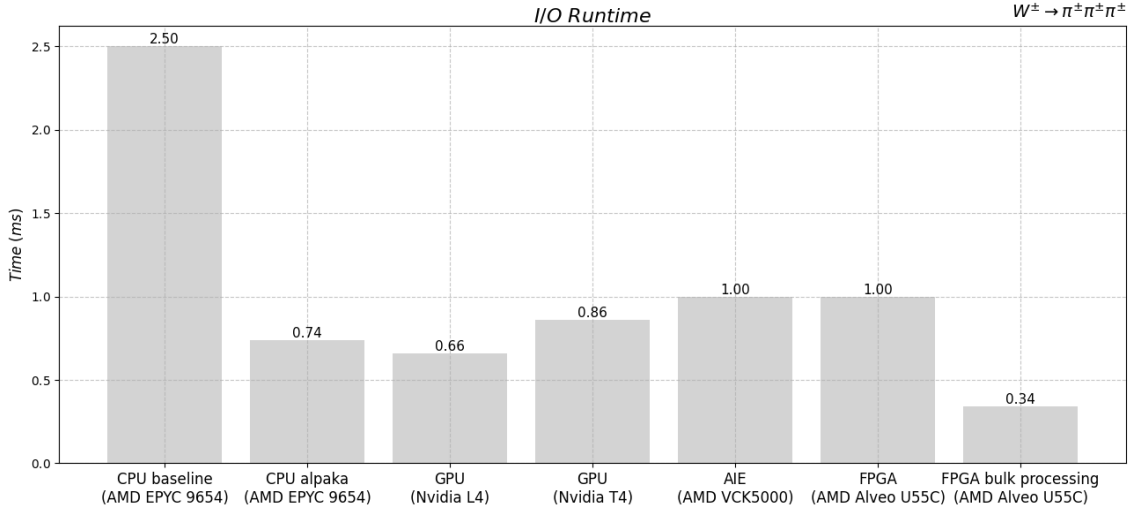


Figure 6.4: I/O time measurements on CPU, GPU, AIE and FPGA.

Overall, the measurements show that bulk processing is the most efficient, especially offloading large amounts of data to the device for an optimal utilization of the PCIe interface to achieve good i/o times. The raw analysis computation performs best on GPU. Figure 6.5 illustrates the total runtime across all devices, and shows the fraction of the i/o times relative to the total runtime. This plot also shows that almost all implementations are bound by the i/o times and memory transfers, especially the highly optimized GPU implementation with more than 80% of the time spent on i/o. An exception is the AIE implementation, which is clearly compute bound with only 12% spent on i/o and would benefit a lot from further kernel optimization. Furthermore, the FPGA bulk implementation indicates further optimization potential of the FPGA kernel implementation.

$W \rightarrow \pi\gamma$ Analysis

The implementation of the $W \rightarrow \pi\gamma$ analysis leveraged several optimizations from the previously developed $W \rightarrow 3\pi$ analysis, though it underwent less extensive optimization and testing. Nonetheless, this implementation served as a testbed for evaluating the three-kernel architecture approach. The runtime performance comparison between the monolithic and three-kernel architecture was conducted using a dataset of 50k events, with results presented in table 6.5. The measurements demonstrate remarkable consistency between both implementation strategies. Despite the fact, that the three-kernel approach includes one additional function in the dataflow region of the

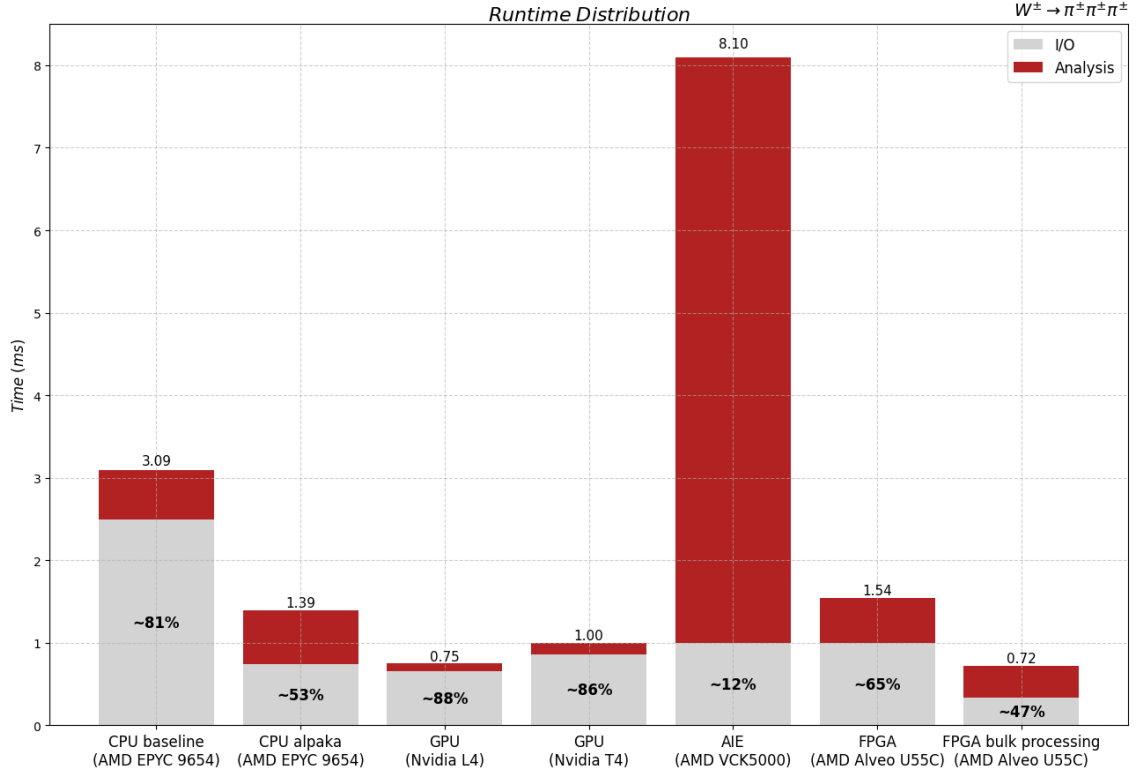


Figure 6.5: Runtime distribution showing the ratio of i/o time on the total execution time across CPU, GPU, AIE and FPGA platforms.

analysis to replicate the streamed data from the input kernel, the measurements indicate that architectural partitioning does not introduce measurable performance penalties.

Design	Total Runtime (ms)	I/O ¹ (ms)	Analysis (ms)
Monolithic Kernel	1.42	0.9	0.52
3 Kernels	1.45	0.91	0.54

Table 6.5: Execution times (per orbit) of the $W \rightarrow \pi\gamma$ analysis comparing the monolithic kernel design with the three-kernel (input, analysis, output) approach..

6.3.2 Resource Utilization

The resource utilization of the GPU, AIE, and FPGA implementations is listed below in table 6.6. For the GPU implementation up to 43% of the available SMs are utilized although the implementation is highly optimized and achieves the best analysis runtime among the tested implementations. This

demonstrates the limitations in exploiting parallel architectures with this algorithm due to its high control flow complexity. The implementation on AIE only uses 1 out of 400 available AIE tiles. This underlines the significant optimization and parallelization potential of this implementation that could benefit from upscaling.

Platform	Resources
GPU	43% of all SMs (Nvidia L4) 30% of all SMs (Nvidia T4)
AIE	1 AIE Tile out of 400
FPGA	Registers: 17.77% (463331 out of 2607360) LUT: 32.84% (428076 out of 1303680) DSP: 11.52% (1040 out of 9024) 36 Kb BRAM: 30.06% (606 out of 2016)

Table 6.6: Resource usage on different hardware platforms.

The FPGA design resource utilization was obtained from the Vivado resource report file after the design was fully placed and routed. The implementation uses up to 33% of the available resources, with LUTs being the mostly used, followed by BRAM with around 30%. This indicates high requirement for combinatorial logic due to the control flow complexity of the algorithm. It also reflects the high on-chip memory requirements for data buffering to realize the combinatorics. The relatively low DSP consumption is likely due to the use of fixed-point and integer data types and the extensive simplification of the calculations.

Overall, it clearly stands out, that the data path complexity limits further optimization and parallelization of the algorithm on almost all platforms, so that only a fraction of the available resources can be exploited. For FPGA, routing problems and unmet timings occur when trying to further scale up the design to use more of the resources. Since the algorithm has large memory dependencies, a majority of the logic must be placed near the HBM banks that are accessible from SLR0. This can also be seen when examining the percentage of logic utilized in each of the three SLRs on the FPGA in table 6.7. Most of the logic is placed in SLR0 (with 54% LUT usage) and SLR1 (with almost 38% LUT usage), which are the closest to the HBM banks. The utilization in both of them is above the total percentage for the entire FPGA as shown in table 6.6. SLR2, on the other hand, is quite underutilized compared to the overall usage.

$W \rightarrow \pi\gamma$ Analysis

Table 6.8 presents the resource utilization of the $W \rightarrow \pi\gamma$ analysis implementation, comparing two design approaches: the monolithic kernel and the three-kernel design, which comprises

Resources	SLR0	SLR1	SLR2
Registers	30.29%	18.67%	4.13%
LUT	54.35%	37.81%	5.97%
DSP	23.13%	12.04%	0.13%
36 Kb BRAM	47.77%	34.97%	7.44%

Table 6.7: Resource usage on FPGA by SLR.

dedicated kernels to read from memory, to perform the analysis computations, and to write the results back to the HBM. Register, LUT, and DSP utilization exhibit negligible variation between the architectures, indicating that the fundamental computational requirements remain unchanged regardless of the kernel partitioning strategy. The most significant difference appears in the BRAM utilization, where the three-kernel approach shows a notable reduction of 2.99% compared to the monolithic kernel. Given the slightly higher analysis kernel execution time of the three-kernel design, this performance-resource trade-off suggests that the reduced buffering in the three-kernel architecture may introduce inter-kernel communication overhead that manifests as increased computational latency.

However, due to the very small differences between the designs it appears worth using separate kernels for computation and memory operations. This approach provides a robust foundation for optimizing each domain independently, thereby facilitating more precise and targeted optimization.

Resources	Monolithic Kernel	3 Kernels
Registers	17.43% (454448 out of 2607360)	17.30% (451080 out of 2607360)
LUT	30.11% (392539 out of 1303680)	29.34% (382451 out of 1303680)
DSP	6.91% (624 out of 9024)	6.91% (624 out of 9024)
36 Kb BRAM	24.80% (500 out of 2016)	21.83% (440 out of 2016)

Table 6.8: Resource usage of the $W \rightarrow \pi\gamma$ analysis, comparing the monolithic kernel design with the three-kernel (input, analysis, output) approach.

6.3.3 Power Consumption

The power consumption data presented here was obtained through the XRT ecosystem's built-in power monitoring capabilities. Specifically, the measurements were captured using the `xbutil` command-line utility, which interfaces with the FPGA device's on-board power monitoring infrastructure and were collected by the *Vitis Analyzer* tool. The `xbutil` utility accesses hardware-level power sensors integrated within the FPGA platform through a power monitoring architecture. The Alveo U55C employs a Texas Instruments MSP432 microcontroller as part of its Satellite Controller firmware, which orchestrates the power monitoring system. The MSP432 interfaces with dedicated power monitoring integrated circuits via I2C/PMBus protocols to collect real-time measurements from three voltage rails (12 V PEX, VCCINT, and 3.3 V PEX). The total power consumption is calculated by summing up the PCIe and AUX power rails, however, the AUX power is 0 because the setup used has no AUX power supply connected to the board [70].

Figure 6.6 shows the power consumption measured for the entire application runtime of the FPGA bulk processing version with a sample rate of approximately 20 ms. The plot demonstrates a stable baseline power draw of approximately 24-25 W during idle and data preparation phases. This baseline consumption is distributed across the primary power rails, with the 12 V PEX rail contributing approximately 20 W and representing the largest component of static power consumption, primarily attributed to the core FPGA fabric, the HBM, and other i/o board components. The 3.3 V PEX rail supports board management controllers, board sensors, and minor peripherals with a consumption of approximately 3.6 W. The VCCINT rail corresponds to the internal core voltage domain powering the programmable logic and on-chip memory systems. With a baseline power of around 9 W it represents a large fraction of the PCIe input power supply. With a baseline power of around 9 W it represents a large fraction of the PCIe input power supply.

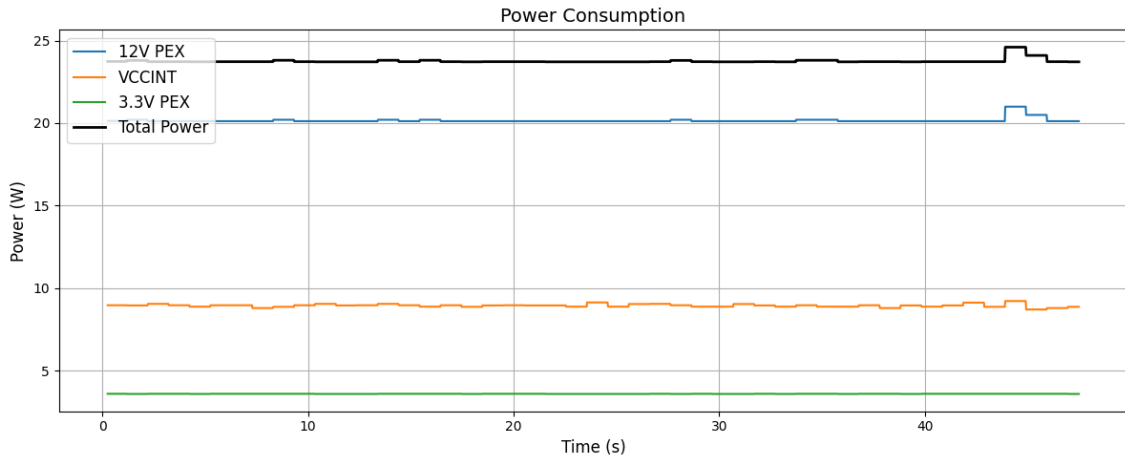


Figure 6.6: Power consumption over the entire application runtime of the FPGA bulk processing version, taken with a sample rate of ~20 ms.

The power profile shows a distinct increase hinting to the actual kernel execution phase with a peak consumption reaching up to 25 W, which is approximately 1 W above the baseline, corresponding to an increase of 4.2%. Additionally, very small spikes during the application runtime before the kernel execution, likely correspond to preparatory operations such as memory allocations and data transfers. The modest increase in power consumption can be attributed to the fact that the baseline idle power of the FPGA encompasses the power consumed by the FPGA logic even during periods when the kernel is not executing. This baseline includes power drawn by for example operating clock signals and powered BRAM modules. Consequently, the observed 1 W increase represents only the additional power consumed by active kernel execution on the programmed logic. Appendix A.2 presents the outputs of the `xbutil examine -device 0000:01:00.1 -report electrical` command during idle time and kernel execution.

The power statistics derived from the measured power consumption data are summarized in table 6.9, which lists the exact maximum and median values of the measurements. The table also includes the maximum total power consumption of the Nvidia L4 GPU, which is approximately 35 W - a value that is energy efficient in absolute terms but still notably higher than that of the FPGA. The GPU power consumption is measured using the power monitoring functionality of `nvidia-smi`. In summary, the power consumption measurement results validate the energy efficiency of the FPGA-based acceleration approach, demonstrating that high computational performance can be achieved with minimal increases in power consumption relative to baseline system operation. Moreover, in direct comparison with the GPU measurements, the FPGA exhibits significantly lower power consumption.

Rail	Average (W)	Median (W)	Maximum (W)
Total (Nvidia L4) ¹	-	-	35.00
Total (Alveo U55C)	23.76	23.72	24.6
12 V PEX	20.16	20.12	21.00
VCCINT	8.95	8.96	9.22
3.3 V PEX	3.60	3.60	3.60

¹power consumption obtained from `nvidia-smi`

Table 6.9: Power statistics over the entire application runtime of the FPGA bulk processing implementation compared to the GPU implementation on the Nvidia L4.

7 Conclusion and Future Work

This work has successfully demonstrated the feasibility and potential of FPGA-based acceleration for particle physics analysis algorithms in the context of the CMS Level-1 Data Scouting system. The implementation of the $W \rightarrow 3\pi$ and $W \rightarrow \pi\gamma$ analyses on the AMD Alveo U55C platform showcases how specialized hardware can enhance computational performance for high-energy physics applications.

The developed system represents a standalone implementation that offers considerable flexibility in deployment scenarios. It could either be extended to be integrated directly into the CMSSW framework to support orbit-wise computation within the existing CMS software infrastructure, or to operate as an independent acceleration unit earlier in the scouting pipeline running on dedicated hardware boards.

Through iterative optimization cycles, including pipeline restructuring, resource allocation refinement, and algorithmic adaptations specifically tailored for FPGA architectures, the implementation performance was progressively enhanced. This optimization effort proved effective, as the final implementation achieves the best total per-orbit runtime for bulk processing and the second-best analysis kernel execution time after the GPU implementation, nearly matching GPU performance levels. The development and design space exploration timelines on FPGA differ significantly from those of traditional GPUs or CPUs, primarily due to the substantially longer iteration cycles required for FPGA development. Consequently, achieving a similar degree of optimization to that seen on CPU and GPU platforms demands more extensive design space exploration on FPGAs. The extensive design space exploration conducted in this work involved the investigation of multiple architectural configurations, memory access patterns, parallelization strategies, and numerical precision trade-offs. In addition to the competitive performance metrics, the analysis FPGA consumes significantly less power than the GPU version while utilizing approximately the same amount of computational resources ($\sim 30\%$), highlighting the energy efficiency benefits of specialized hardware acceleration.

Accompanying this hardware implementation, a small library with ROOT like user interface was developed that generates code for both software and hardware-accelerated analysis execution, enabling seamless comparison and validation between different computational approaches.

Future Work

The current implementation serves as a proof-of-concept that demonstrates the feasibility of FPGA-equipped accelerator cards within the L1DS system and establishes a foundation for further FPGA-based particle physics analysis implementations. For future development on the basis of this work, two main directions can be identified:

Library Expansion and Usability Enhancement: The existing small library implementation could be expanded into a more comprehensive framework with improved usability. A python-based interface would significantly lower the entry barrier for physicists, encouraging broader adoption within the particle physics community. This could be achieved through template-based code generation using systems like mustache templating [71], enabling dynamic generation of hardware description code from user-defined analysis specifications. Alternatively, deeper integration into the ROOT framework would leverage existing workflows and provide seamless access to FPGA acceleration capabilities.

The expansion could also encompass the implementation of additional hardware modules to support a broader spectrum of particle physics analyses and input data formats towards a versatile acceleration platform capable of handling diverse physics use cases. A valuable enhancement would for example be the functionality to process multiple analyses on the same FPGA in parallel, maximizing resource utilization and enabling simultaneous execution of different physics channels on the same hardware platform.

Performance Optimization: Continued design space exploration could focus on pushing the performance boundaries further. This includes investigating further hardware architectures, exploring novel algorithmic implementations, and optimizing data movement patterns to minimize latency and maximize throughput. Future optimization efforts could explore whether a single FPGA implementation might achieve the target 40 MHz processing rate through further architectural refinements. Alternatively, performance gains could be pursued through heterogeneous computing approaches that leverage the co-design of CPU, GPU, and FPGA resources, or by utilizing advanced hybrid architectures such as AIE that combine FPGA-like programmability with GPU-like parallel

processing capabilities.

With continued development along these outlined directions, FPGA based systems could become integral components of the existing computing infrastructures, enabling more sophisticated analyses and expanding the scientific reach of particle physics experiments. The foundation established by this work demonstrates the significant potential of FPGA acceleration in particle physics analysis.

Bibliography

- [1] Aberle, O. and others. *High-Luminosity Large Hadron Collider (HL-LHC): Technical design report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2020. DOI: [10.23731/CYRM-2020-0010](https://cds.cern.ch/record/2749422). URL: <https://cds.cern.ch/record/2749422>.
- [2] G L Bayatyan et al. *CMS TriDAS project: Technical Design Report, Volume 1: The Trigger Systems*. Technical design report. CMS. URL: <https://cds.cern.ch/record/706847>.
- [3] Christopher Edward Brown. "Fast Machine Learning in the CMS Level-1 Trigger for the High-Luminosity LHC". PhD thesis. Imperial Coll., London, 2023. URL: <https://cds.cern.ch/record/2875830>.
- [4] Abhijith Gandrakota. *Real-time Anomaly Detection at the L1 Trigger of CMS Experiment*. 2024. arXiv: [2411.19506](https://arxiv.org/abs/2411.19506) [hep-ex]. URL: <https://arxiv.org/abs/2411.19506>.
- [5] A. M. Sirunyan et al. "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>.
- [6] Jared Fraticelli et al. "GTT: $W \rightarrow 3\pi$ Module". 2024.
- [7] Francesco Brivio et al. "W decay to 3 pions on Alveo: a case study". 2024.
- [8] Giovanni Zago. "Accelerating Online Selection Algorithms for Level-1 Trigger Scouting in the CMS Phase-2 Upgrade". MA thesis. University of Padua, Italy, 2025.
- [9] Lukasz Michalski. "Real-time and Quasi-Real-Time Data Acquisition and Physics Analysis Algorithms for the Level 1 Trigger System". MA thesis. Wrocław University of Science and Technology, Poland, 2025.
- [10] Robert Mann. *An Introduction to Particle Physics and the Standard Model*. Taylor & Francis, 2010. ISBN: 978-1-4200-8300-2. DOI: [10.1201/9781420083002](https://doi.org/10.1201/9781420083002).
- [11] CERN. URL: <https://home.cern> (visited on 06/12/2025).

- [12] MissM] et al. *Standard Model of Elementary Particles*. URL: https://commons.wikimedia.org/wiki/File:Standard_Model_of_Elementary_Particles.svg (visited on 06/13/2025).
- [13] Oliver Sim Brüning et al. *LHC Design Report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2004. DOI: [10.5170/CERN-2004-003-V-1](https://cds.cern.ch/record/782076). URL: <https://cds.cern.ch/record/782076>.
- [14] S Chatrchyan et al. "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://cds.cern.ch/record/1129810). URL: <https://cds.cern.ch/record/1129810>.
- [15] G Aad et al. "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://cds.cern.ch/record/1129811). URL: <https://cds.cern.ch/record/1129811>.
- [16] A Augusto Alves et al. "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://cds.cern.ch/record/1129809). URL: <https://cds.cern.ch/record/1129809>.
- [17] K Aamodt et al. "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://cds.cern.ch/record/1129812). URL: <https://cds.cern.ch/record/1129812>.
- [18] 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>.
- [19] C Wiesner et al. "Abort Gap Keeper and Abort Gap Protection". In: (2019), pp. 141–145. URL: <https://cds.cern.ch/record/2813530>.
- [20] CMS Collaboration. *Full summary of proton-proton collision luminosity measurements*. CERN. 2024. URL: https://twiki.cern.ch/twiki/bin/view/CMSPublic/LumiPublicResults#Full_summary_proton_proton_colli (visited on 06/13/2025).
- [21] Matteo Migliorini. *40MHz scouting at the CMS experiment*. 2024.
- [22] TikZ.net contributors. *3D Axis CMS Style Example (TikZ)*. TikZ example for CMS-style 3D axis. 2021. URL: https://tikz.net/axis3d_cms/ (visited on 06/11/2025).
- [23] CMS Collaboration. *The CMS Detector at CERN*. Overview of the Compact Muon Solenoid (CMS) experiment and its components. CERN. 2024. URL: <https://cms.cern/index.php/detector> (visited on 06/11/2025).
- [24] V Karimäki et al. *The CMS tracker system project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: <https://cds.cern.ch/record/368412>.

- [25] *The CMS electromagnetic calorimeter project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: <https://cds.cern.ch/record/349375>.
- [26] *The CMS hadron calorimeter project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: <https://cds.cern.ch/record/357153>.
- [27] J. G. Layter. *The CMS muon project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: <https://cds.cern.ch/record/343814>.
- [28] 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>.
- [29] L Agostino et al. "Commissioning of the CMS High Level Trigger". In: *Journal of Instrumentation* 4.10 (2009), P10005. DOI: [10.1088/1748-0221/4/10/P10005](https://doi.org/10.1088/1748-0221/4/10/P10005). URL: <https://dx.doi.org/10.1088/1748-0221/4/10/P10005>.
- [30] A Tapper and Darin Acosta. *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. 2013. URL: <https://cds.cern.ch/record/1556311>.
- [31] Marcin Andrzej Konecki. *The CMS Level-1 muon triggers for the LHC Run II*. Tech. rep. Geneva: CERN, 2019. DOI: [10.22323/1.340.0918](https://cds.cern.ch/record/2646770). URL: <https://cds.cern.ch/record/2646770>.
- [32] B. Kreis et al. "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>.
- [33] J. Wittmann et al. "The upgrade of the CMS Global Trigger". In: *Journal of Instrumentation* 11.02 (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>.
- [34] D Contardo et al. *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, 2015. DOI: [10.17181/CERN.VU8I.D59J](https://cds.cern.ch/record/2020886). URL: <https://cds.cern.ch/record/2020886>.
- [35] *The Phase-2 Upgrade of the CMS Tracker*. Tech. rep. Geneva: CERN, 2017. DOI: [10.17181/CERN.QZ28.FLHW](https://cds.cern.ch/record/2272264). URL: <https://cds.cern.ch/record/2272264>.
- [36] Collaboration CMS. *A MIP Timing Detector for the CMS Phase-2 Upgrade*. Tech. rep. Geneva: CERN, 2019. URL: <https://cds.cern.ch/record/2667167>.

- [37] *The Phase-2 Upgrade of the CMS Endcap Calorimeter*. Tech. rep. Geneva: CERN, 2017. DOI: [10.17181/CERN.IV8M.1JY2](https://cds.cern.ch/record/2293646). URL: <https://cds.cern.ch/record/2293646>.
- [38] *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>.
- [39] *The Phase-2 Upgrade of the CMS Muon Detectors*. Tech. rep. This is the final version, approved by the LHCC. Geneva: CERN, 2017. DOI: [10.17181/CERN.5T9S.VPMI](https://cds.cern.ch/record/2283189). URL: <https://cds.cern.ch/record/2283189>.
- [40] *The Phase-2 Upgrade of the CMS Level-1 Trigger*. Tech. rep. Final version. Geneva: CERN, 2020. URL: <https://cds.cern.ch/record/2714892>.
- [41] A.M. Sirunyan et al. "Particle-flow reconstruction and global event description with the CMS detector". In: *Journal of Instrumentation* 12.10 (2017), P10003. DOI: [10.1088/1748-0221/12/10/P10003](https://dx.doi.org/10.1088/1748-0221/12/10/P10003). URL: <https://dx.doi.org/10.1088/1748-0221/12/10/P10003>.
- [42] Daniele Bertolini et al. "Pileup per particle identification". In: *Journal of High Energy Physics* 2014.10 (Oct. 2014). ISSN: 1029-8479. DOI: [10.1007/JHEP10\(2014\)059](https://dx.doi.org/10.1007/JHEP10(2014)059). URL: [http://dx.doi.org/10.1007/JHEP10\(2014\)059](http://dx.doi.org/10.1007/JHEP10(2014)059).
- [43] Emilio Meschi. *The CMS Level-1 Trigger Data Scouting system for the HL-LHC upgrade*. Tech. rep. Geneva: CERN, 2024. DOI: [10.22323/1.476.0864](https://cds.cern.ch/record/2913371). URL: <https://cds.cern.ch/record/2913371>.
- [44] Leah-louisa Sieder. *CMS Level-1 Data Scouting for High-Luminosity LHC*. Tech. rep. Geneva: CERN, 2025. URL: <https://cds.cern.ch/record/2931373>.
- [45] "Level-1 trigger scouting in Phase-2". In: (2024). URL: <https://cds.cern.ch/record/2916191>.
- [46] 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>.
- [47] M. Migliorini. "An online data processing system for the CMS Level-1 Trigger data scouting demonstrator". In: *Proceedings of the 27th Conference on Computing in High Energy and Nuclear Physics (CHEP 2024)*. CHEP. Krakow, Poland, 2024.
- [48] CMS Collaboration. *CMSSW – CMS Offline Software*. GitHub repository. 2024. URL: <https://github.com/cms-sw/cmssw> (visited on 06/13/2025).
- [49] Rocco Ardino et al. "Design and perspectives of the CMS Level-1 trigger Data Scouting system". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 1067 (2024), p. 169719. ISSN: 0168-9002. DOI: <https://doi.org/10.1016/j.nima.2024.169719>. URL: <https://www.sciencedirect.com/science/article/pii/S0168900224006454>.

- [50] FastML Team. *fastmachinelearning/hls4ml*. URL: <https://github.com/fastmachinelearning/hls4ml> (visited on 06/13/2025).
- [51] FastML Team. *conifer: Fast inference of Boosted Decision Trees in FPGAs*. URL: <https://github.com/thesps/conifer> (visited on 06/13/2025).
- [52] Pietro Cappelli. "On the measurement of W to 3 pi with the Phase 2 L1 scouting system at CMS". MA thesis. University of Padua, Italy, 2023.
- [53] Catherine Miller. 2022.
- [54] Erik Zenker et al. "Alpaka - An Abstraction Library for Parallel Kernel Acceleration". In: IEEE Computer Society, 2016. arXiv: 1602.08477. URL: <http://arxiv.org/abs/1602.08477>.
- [55] Benjamin Worpitz et al. *alpaka: Abstraction Library for Parallel Kernel Acceleration*. URL: <https://zenodo.org/records/4452613> (visited on 06/13/2025).
- [56] AMD EPYC™ 9654. Advanced Micro Devices, Inc. URL: <https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9654.html> (visited on 06/13/2025).
- [57] *Alveo U55C Data Center Accelerator Cards Data Sheet*. Version 2023-06-23. Advanced Micro Devices, Inc. URL: <https://docs.amd.com/r/en-US/ds978-u55c/Summary> (visited on 06/13/2025).
- [58] *Virtex UltraScale+ FPGA Product Brief*. Advanced Micro Devices, Inc. URL: <https://www.amd.com/content/dam/amd/en/documents/products/adaptive-socs-and-fpgas/fpga/ultrascale-plus/virtex-ultrascale-product-brief.pdf> (visited on 06/13/2025).
- [59] *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)*. Advanced Micro Devices, Inc. URL: <https://docs.amd.com/r/2023.2-English/ug1393-vitis-application-acceleration/Getting-Started-with-Vitis> (visited on 06/13/2025).
- [60] Louis Woods, Gustavo Alonso, and Jens Teubner. "Parallel Computation of Skyline Queries". In: *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. 2013, pp. 1–8. DOI: [10.1109/FCCM.2013.18](https://doi.org/10.1109/FCCM.2013.18).
- [61] René Brun and Fons Rademakers. "ROOT – An Object Oriented Data Analysis Framework". In: *AIHENP'96 Workshop, Lausanne, September 1996*. Vol. 389. Nucl. Instrum. Methods Phys. Res. A. 1997, pp. 81–86. DOI: [10.1016/S0168-9002\(97\)00048-X](https://doi.org/10.1016/S0168-9002(97)00048-X).
- [62] René Brun, Fons Rademakers, and the ROOT Team. *ROOT: An Object Oriented Data Analysis Framework*. URL: <https://root.cern/> (visited on 06/13/2025).
- [63] TechPowerUp. *AMD EPYC 9654*. URL: <https://www.techpowerup.com/cpu-specs/epyc-9654.c2933> (visited on 06/13/2025).

- [64] Nvidia. *NVIDIA L4 Tensor Core GPU*. Nvidia. URL: <https://nvdam.widen.net/s/rvq98gbwsw/14-datasheet-2595652> (visited on 06/13/2025).
- [65] TechPowerUp. *NVIDIA L4*. URL: <https://www.techpowerup.com/gpu-specs/l4.c4091> (visited on 06/13/2025).
- [66] *NVIDIA T4 TENSOR CORE GPU*. Nvidia. URL: <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf> (visited on 06/13/2025).
- [67] TechPowerUp. *NVIDIA Tesla T4*. URL: <https://www.techpowerup.com/gpu-specs/tesla-t4.c3316> (visited on 06/13/2025).
- [68] *VCK5000 Data Center Acceleration Development Kit Hardware Installation Guide (UG1531)*. Version 2023-07-19. Advanced Micro Devices, Inc. URL: <https://docs.amd.com/r/en-US/ug1531-vck5000-install/Introduction> (visited on 06/13/2025).
- [69] *Versal Architecture and Product Data Sheet: Overview*. Advanced Micro Devices, Inc. URL: <https://docs.amd.com/v/u/en-US/ds950-versal-overview> (visited on 06/13/2025).
- [70] *Alveo Card Management Solution Subsystem Product Guide (PG348)*. Version 2024-11-13. Advanced Micro Devices, Inc. URL: <https://docs.amd.com/r/en-US/pg348-cms-subsystem/Sensor-Monitoring> (visited on 06/13/2025).
- [71] Chris Wanstrath et al. *Mustache: Logic-less Templates*. URL: <https://mustache.github.io/> (visited on 06/13/2025).

A Appendix

A.1 HLS and ROOT Analysis Implementation

Listing A.1: HLS C++ entry function of the analysis computation.

```
1  extern "C"
2  {
3      void analysis(
4          const PUPPI_T *event_hbm_0,
5          PUPPI_TRIO_T *event_hbm_1
6      )
7      {
8          #pragma HLS interface m_axi port = event_hbm_0 \
9              offset = slave \
10             bundle = analysis_buf_in0 \
11             depth = NUM_EVENTS/NUM_CU * MAX_NUM_PARTICLES
12         #pragma HLS interface m_axi port = event_hbm_1 \
13             offset = slave \
14             bundle = analysis_buf_out \
15             depth = MAX_NUM_PARTICLES
16
17         // AXI Lite interfaces for control
18         #pragma HLS interface s_axilite port = event_hbm_0
19         #pragma HLS interface s_axilite port = event_hbm_1
20         #pragma HLS interface s_axilite port = return
21
22         hls::stream<PUPPI_T> event_stream_0;
23         hls::stream<PUPPI_T> event_stream_1;
24         hls::stream<PUPPI_T> event_stream_2;
25         hls::stream<PUPPI_T> event_stream_3;
26         hls::stream<PUPPI_T> event_stream_4;
27         hls::stream<PUPPI_T> event_stream_5;
28         hls::stream<PUPPI_T> event_stream_6;
29         hls::stream<PUPPI_T> event_stream_7;
30         hls::stream<PUPPI_PAIR_T> event_stream_8;
31         hls::stream<PUPPI_T> event_stream_9;
32         hls::stream<PUPPI_TRIO_T> event_stream_10;
33         hls::stream<PUPPI_TRIO_T> event_stream_11;
34         hls::stream<PUPPI_TRIO_T> event_stream_12;
35         hls::stream<PUPPI_TRIO_T> event_stream_13;
36
37         #pragma HLS STREAM variable = event_stream_0 depth = MAX_NUM_PARTICLES
38         #pragma HLS STREAM variable = event_stream_1 depth = MAX_NUM_PARTICLES
39         #pragma HLS STREAM variable = event_stream_2 depth = MAX_NUM_PARTICLES
```

```

40 #pragma HLS STREAM variable = event_stream_3 depth = MAX_NUM_PARTICLES
41 #pragma HLS STREAM variable = event_stream_4 depth = MAX_NUM_PARTICLES
42 #pragma HLS STREAM variable = event_stream_5 depth = MAX_NUM_PARTICLES
43 #pragma HLS STREAM variable = event_stream_6 depth = MAX_NUM_PARTICLES
44 #pragma HLS STREAM variable = event_stream_7 depth = MAX_NUM_PARTICLES
45 #pragma HLS STREAM variable = event_stream_8 depth = MAX_NUM_PARTICLES
46 #pragma HLS STREAM variable = event_stream_9 depth = MAX_NUM_PARTICLES
47 #pragma HLS STREAM variable = event_stream_10 depth = MAX_NUM_PARTICLES
48 #pragma HLS STREAM variable = event_stream_11 depth = MAX_NUM_PARTICLES
49 #pragma HLS STREAM variable = event_stream_12 depth = MAX_NUM_PARTICLES
50 #pragma HLS STREAM variable = event_stream_13 depth = MAX_NUM_PARTICLES
51
52 #pragma HLS dataflow
53
54 #ifndef __SYNTHESIS__
55     std::cout << std::endl;
56     std::cout << "\033[36mDEBUG [HW]:\033[0m Entering load-events 0" << std::endl;
57 #endif
58     load_events_hbm(
59         event_hbm_0,
60         event_stream_0,
61         event_stream_1);
62 #ifndef __SYNTHESIS__
63     std::cout << "\033[36mDEBUG [HW]:\033[0m Leaving load-events" << std::endl;
64     std::cout << std::endl;
65 #endif
66
67 #ifndef __SYNTHESIS__
68     std::cout << std::endl;
69     std::cout << "\033[36mDEBUG [HW]:\033[0m Entering pdgidcheck" << std::endl;
70 #endif
71     pdgidcheck(
72         event_stream_1,
73         event_stream_2);
74 #ifndef __SYNTHESIS__
75     std::cout << "\033[36mDEBUG [HW]:\033[0m Leaving pdgidcheck" << std::endl;
76     std::cout << std::endl;
77 #endif
78
79 #ifndef __SYNTHESIS__
80     std::cout << std::endl;
81     std::cout << "\033[36mDEBUG [HW]:\033[0m Entering isolation" << std::endl;
82 #endif
83     isolation_simple(
84         event_stream_0,
85         event_stream_2,
86         event_stream_3,
87         event_stream_4,
88         event_stream_7);
89 #ifndef __SYNTHESIS__
90     std::cout
91         << "\033[36mDEBUG [HW]:\033[0m Leaving isolation" << std::endl;
92     std::cout << std::endl;
93 #endif
94
95 #ifndef __SYNTHESIS__
96     std::cout << std::endl;
97     std::cout << "\033[36mDEBUG [HW]:\033[0m Entering high-pt cut" << std::endl;
98 #endif
99     highptcut(
100         event_stream_3,
101         event_stream_5);
102 #ifndef __SYNTHESIS__
103     std::cout

```



```

104         << "\033[36mDEBUG [HW]:\033[0m Leaving high-pt cut" << std::endl;
105         std::cout << std::endl;
106     #endif
107
108     #ifndef __SYNTHESIS__
109         std::cout << std::endl;
110         std::cout << "\033[36mDEBUG [HW]:\033[0m Entering mid-pt cut" << std::endl;
111     #endif
112     midptcut(
113         event_stream_4,
114         event_stream_6);
115     #ifndef __SYNTHESIS__
116         std::cout
117         << "\033[36mDEBUG [HW]:\033[0m Leaving mid-pt cut" << std::endl;
118         std::cout << std::endl;
119     #endif
120
121     #ifndef __SYNTHESIS__
122         std::cout << std::endl;
123         std::cout << "\033[36mDEBUG [HW]:\033[0m Entering make pairs" << std::endl;
124     #endif
125     makepairs(
126         event_stream_5,
127         event_stream_6,
128         event_stream_8);
129     #ifndef __SYNTHESIS__
130         std::cout << "\033[36mDEBUG [HW]:\033[0m Leaving make pairs" << std::endl;
131     #endif
132
133     #ifndef __SYNTHESIS__
134         std::cout << std::endl;
135         std::cout << "\033[36mDEBUG [HW]:\033[0m Entering make triplets" << std::endl;
136     #endif
137     maketriplets(
138         event_stream_7,
139         event_stream_8,
140         event_stream_10);
141
142     #ifndef __SYNTHESIS__
143         std::cout << std::endl;
144         std::cout << "\033[36mDEBUG [HW]:\033[0m Leaving make triplets" << std::endl;
145     #endif
146
147     #ifndef __SYNTHESIS__
148         std::cout << std::endl;
149         std::cout << "\033[36mDEBUG [HW]:\033[0m Entering angular sep 2" << std::endl;
150     #endif
151     deltar_triplets(
152         event_stream_10,
153         event_stream_12);
154     #ifndef __SYNTHESIS__
155         std::cout
156         << "\033[36mDEBUG [HW]:\033[0m Leaving angular sep 2" << std::endl;
157         std::cout << std::endl;
158     #endif
159
160     #ifndef __SYNTHESIS__
161         std::cout << std::endl;
162         std::cout << "\033[36mDEBUG [HW]:\033[0m Entering triplet mass" << std::endl;
163     #endif
164     tripletmass(
165         event_stream_12,
166         event_stream_13);

```

```

168 #ifndef __SYNTHESIS__
169     std::cout
170     << "\033[36mDEBUG [HW]:\033[0m Leaving triplet mass" << std::endl;
171     std::cout << std::endl;
172 #endif
173
174 #ifndef __SYNTHESIS__
175     std::cout << std::endl;
176     std::cout << "\033[36mDEBUG [HW]:\033[0m Entering store-result" << std::endl;
177 #endif
178     store_result(
179         event_stream_13,
180         event_hbm_1);
181 #ifndef __SYNTHESIS__
182     std::cout
183     << "\033[36mDEBUG [HW]:\033[0m Leaving store-result" << std::endl;
184     std::cout << std::endl;
185 #endif
186 }
187 }

```

Listing A.2: ROOT version of the analysis computation.

```

1  std::vector<SW_TRIPLET_T> sw_analysis(
2      const char *rootfile,
3      const char *tree,
4      const char **leafs)
5  {
6      // Open ROOT file
7      ROOT::RDataFrame df(tree, rootfile);
8      ROOT::RDF::RNode rn = df;
9      rn = rn.Range(NUM_EVENTS);
10
11     // 0. form particle objects
12     auto rn0 = rn.Define("ParticleObjects",
13         [](const ROOT::RVec<float> &etas, const ROOT::RVec<float> &phis, const
14             ↪ ROOT::RVec<float> &pts, const ROOT::RVec<int> &pdgids)
15         {
16             std::vector<SW_OBJ_T> objects;
17             for (size_t i; i < pdgids.size(); i++)
18             {
19                 objects.emplace_back(std::make_tuple(etas[i], phis[i], pts[i],
20                 ↪ pdgids[i]));
21             }
22             return objects;
23         },
24         {leafs[1], leafs[2], leafs[3], leafs[4]});
25     auto c0 = rn0.Filter([](const std::vector<SW_OBJ_T> &objects)
26         { return !objects.empty(); }, {"ParticleObjects"})
27         .Count();
28     std::cout << "#particles after 0: " << std::dec << *c0 << std::endl;
29
30     // 1. pdgid check
31     auto rn1 = rn0.Define("AfterPdgidcheck", sw_pdgidcheck, {"ParticleObjects"});
32     auto c1 = rn1.Filter([](const std::vector<SW_OBJ_T> &objects)
33         { return !objects.empty(); }, {"AfterPdgidcheck"})
34         .Count();
35     std::cout << "#particles after 1: " << std::dec << *c1 << std::endl;
36
37     // 2. low pt cut
38     auto rn2 = rn1.Define("AfterLowptcut", sw_lowptcut, {"AfterPdgidcheck"});
39     auto c2 = rn2.Filter([](const std::vector<SW_OBJ_T> &objects)
40         { return !objects.empty(); }, {"AfterLowptcut"})

```

```

39         .Count();
40     std::cout << "#particles after 2: " << std::dec << *c2 << std::endl;
41
42     // 3. isolation
43     auto rn3 = rn2.Define("AfterIsolation", sw_isolation, {"AfterLowptcut", "ParticleObjects"});
44     auto c3 = rn3.Filter([](const std::vector<SW_OBJ_T> &objects)
45         { return !objects.empty(); }, {"AfterIsolation"})
46         .Count();
47     std::cout << "#particles after 3: " << std::dec << *c3 << std::endl;
48
49     // 4. mid pt cut
50     auto rn4 = rn3.Define("AfterMidptcut", sw_midptcut, {"AfterIsolation"});
51     auto c4 = rn4.Filter([](const std::vector<SW_OBJ_T> &objects)
52         { return !objects.empty(); }, {"AfterMidptcut"})
53         .Count();
54     std::cout << "#particles after 4: " << std::dec << *c4 << std::endl;
55
56     // 5. high pt cut
57     auto rn5 = rn4.Define("AfterHighptcut", sw_highptcut, {"AfterMidptcut"});
58     auto c5 = rn5.Filter([](const std::vector<SW_OBJ_T> &objects)
59         { return !objects.empty(); }, {"AfterHighptcut"})
60         .Count();
61     std::cout << "#particles after 5: " << std::dec << *c5 << std::endl;
62
63     // 6. make pairs and check angular separation on pairs
64     auto rn6 = rn5.Define("AfterMakepairs", sw_combine_pairs, {"AfterHighptcut", "AfterMidptcut"});
65     auto c6 = rn6.Filter([](const std::vector<SW_PAIR_T> &pairs)
66         { return !pairs.empty(); }, {"AfterMakepairs"})
67         .Count();
68     std::cout << "#particles after 6: " << std::dec << *c6 << std::endl;
69
70     // 7. make triplets and check charge on triplets
71     auto rn7 = rn6.Define("AfterMaketriplets", sw_combine_triplets, {"AfterLowptcut",
72         ↪ "AfterMakepairs"});
73     auto c7 = rn7.Filter([](const std::vector<SW_TRIPLET_T> &triplets)
74         { return !triplets.empty(); }, {"AfterMaketriplets"})
75         .Count();
76     std::cout << "#particles after 7: " << std::dec << *c7 << std::endl;
77
78     // 8. invariant mass test
79     auto rn8 = rn7.Define("AfterMass", sw_mass, {"AfterMaketriplets"});
80     auto c8 = rn8.Filter([](const std::vector<SW_TRIPLET_T> &triplets)
81         { return !triplets.empty(); }, {"AfterMass"})
82         .Count();
83     std::cout << "#particles after 8: " << std::dec << *c8 << std::endl;
84
85     // 9. angular separation on triplets
86     auto rn9 = rn8.Define("AfterAngularsep", sw_angularsep_triplets, {"AfterMass"});
87     auto c9 = rn9.Filter([](const std::vector<SW_TRIPLET_T> &triplets)
88         { return !triplets.empty(); }, {"AfterAngularsep"})
89         .Count();
90     std::cout << "#particles after 9: " << std::dec << *c9 << std::endl;
91
92     // 10. get indices of final triplets
93     auto rn10 = rn9.Define("TripletEventIndex",
94         [](const std::vector<SW_TRIPLET_T> &triplets)
95         {
96             return !triplets.empty();
97         }, {"AfterAngularsep"});
98
99     auto triplet_event_index_vector = rn10.Take<bool>("TripletEventIndex");
100
101     // 11. get final triplets

```

```

102     auto filtered_particle_triplets = rn9.Filter([](const std::vector<SW_TRIPLET_T> &triplets)
103         { return !triplets.empty(); }, {"AfterAngularsep"})
104         .Take<std::vector<SW_TRIPLET_T>>("AfterAngularsep");
105
106     const auto &nested_vector = *filtered_particle_triplets;
107     std::vector<SW_TRIPLET_T> final_triplets;
108     int event_count = 0;
109
110     for (size_t i = 0; i < triplet_event_index_vector->size(); ++i)
111     {
112         if ((*triplet_event_index_vector)[i])
113         {
114             auto inner_vector = nested_vector[event_count];
115             std::cout << "Event " << i << " has " << inner_vector.size() << " triplets." << std::endl;
116             final_triplets.insert(final_triplets.end(), inner_vector.begin(), inner_vector.end());
117             event_count++;
118         }
119     }
120
121     return final_triplets;
122 }

```

A.2 Power Measurements

Listing A.3: Alveo U55C idle power measurement obtained from xbutil.

```

1 -----
2 [0000:01:00.1] : xilinx_u55c_gen3x16_xdma_base_3
3 -----
4 Electrical
5   Max Power          : 75 Watts
6   Power              : 23.721256 Watts
7   Power Warning      : false
8
9   Power Rails        : Voltage Current
10  -----
11  12 Volts Auxillary  : 0.792 V 0.000 A
12  12 Volts PCI Express : 12.200 V 1.649 A
13  3.3 Volts PCI Express : 3.312 V 1.088 A
14  3.3 Volts Auxillary  : 0.000 V 0.000 A
15  Internal FPGA Vcc    : 0.854 V 10.500 A
16  Internal FPGA Vcc IO : 0.855 V 3.100 A
17  DDR Vpp Bottom      : 0.000 V 0.000 A
18  DDR Vpp Top         : 0.000 V 0.000 A
19  5.5 Volts System     : 5.044 V 0.000 A
20  Vcc 1.2 Volts Top    : 0.000 V 0.000 A
21  Vcc 1.2 Volts Bottom : 0.000 V 0.000 A
22  1.8 Volts Top        : 1.797 V 0.000 A
23  0.9 Volts Vcc        : 0.894 V 0.000 A
24  12 Volts SW          : 0.000 V 0.000 A
25  Mgt Vtt              : 1.197 V 0.000 A
26  3.3 Volts Vcc        : 3.359 V 0.000 A
27  1.2 Volts HBM        : 1.201 V 0.000 A
28  Vpp 2.5 Volts        : 2.473 V 0.000 A
29  12 Volts Aux1        : 0.000 V 0.000 A
30  Vcc 1.2 Volts i      : 0.000 V 0.000 A
31  V12 in i             : 0.000 V 0.000 A
32  V12 in Aux0 i        : 0.000 V 0.000 A
33  V12 in Aux1 i        : 0.000 V 0.000 A

```

```

34 Vcc Auxillary      : 0.000 V 0.000 A
35 Vcc Auxillary Pmc  : 0.000 V 0.000 A
36 Vcc Ram            : 0.000 V 0.000 A
37 0.9 Volts Vcc Vcu  : 0.000 V 0.000 A

```

Listing A.4: Alveo U55C power measurement during kernel execution obtained from xbutil.

```

1 -----
2 [0000:01:00.1] : xilinx_u55c_gen3x16_xdma_base_3
3 -----
4 Electrical
5   Max Power      : 75 Watts
6   Power          : 24.599656 Watts
7   Power Warning   : false
8
9   Power Rails    : Voltage Current
10  -----
11  12 Volts Auxillary : 0.792 V 0.000 A
12  12 Volts PCI Express : 12.200 V 1.721 A
13  3.3 Volts PCI Express : 3.312 V 1.088 A
14  3.3 Volts Auxillary : 0.000 V 0.000 A
15  Internal FPGA Vcc : 0.854 V 10.800 A
16  Internal FPGA Vcc IO : 0.855 V 3.500 A
17  DDR Vpp Bottom : 0.000 V 0.000 A
18  DDR Vpp Top : 0.000 V 0.000 A
19  5.5 Volts System : 5.044 V 0.000 A
20  Vcc 1.2 Volts Top : 0.000 V 0.000 A
21  Vcc 1.2 Volts Bottom : 0.000 V 0.000 A
22  1.8 Volts Top : 1.797 V 0.000 A
23  0.9 Volts Vcc : 0.894 V 0.000 A
24  12 Volts SW : 0.000 V 0.000 A
25  Mgt Vtt : 1.196 V 0.000 A
26  3.3 Volts Vcc : 3.360 V 0.000 A
27  1.2 Volts HBM : 1.202 V 0.000 A
28  Vpp 2.5 Volts : 2.473 V 0.000 A
29  12 Volts Aux1 : 0.000 V 0.000 A
30  Vcc 1.2 Volts i : 0.000 V 0.000 A
31  V12 in i : 0.000 V 0.000 A
32  V12 in Aux0 i : 0.000 V 0.000 A
33  V12 in Aux1 i : 0.000 V 0.000 A
34  Vcc Auxillary : 0.000 V 0.000 A
35  Vcc Auxillary Pmc : 0.000 V 0.000 A
36  Vcc Ram : 0.000 V 0.000 A
37  0.9 Volts Vcc Vcu : 0.000 V 0.000 A

```