TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

# PCIe monitoring for secure code execution in heterogeneous system architectures

*Author:*
Ioannis-Iason
GEORGAKAS

*Thesis Committee:*
Prof. Sotirios IOANNIDIS
Prof. Apostolos DOLLAS
Dr. Konstantinos
GEORGOPOULOS

*A thesis submitted in fulfillment of the requirements*
*for the diploma of Electrical and Computer Engineer*

*in the*

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

July 8, 2024

TECHNICAL UNIVERSITY OF CRETE

# *Abstract*

School of Electrical and Computer Engineering

Electrical and Computer Engineer

**PCIe monitoring for secure code execution in heterogeneous system architectures**

by Ioannis-Iason GEORGAKAS

Nowadays, heterogeneous systems architectures are becoming increasingly popular for running complex and heavy-load computational tasks. Many companies own vast data centers consisting of thousands of heterogeneous systems, allowing developers to run their applications in GPUs, FPGA, or TPUs. However, the rise in the use of such systems has led to a new vast attack vector with various possibilities to emerge. GPU vulnerabilities and their exploitations are realistic instances of such attacks. An instance of an attack on a GPU is to misuse a miner by continuously executing it and, as a result, overload the GPU and take it out for a long time period. This thesis presents a novel method of detecting attacks in heterogeneous systems by monitoring the PCIe traffic at a low level from the host to the corresponding PCIe endpoint. Monitoring the PCIe traffic makes it possible to check all the data and code being transferred to an accelerator in real-time and detect if any exhibit malicious behavior. All the PCIe traffic can be monitored based on rules that mark behaviors or patterns as malicious. This thesis presents a tool to monitor the PCIe traffic in an emulated heterogeneous system CPU-GPU using an MPSoC and an FPGA for any GPU miner running in the discrete GPU based on a set of rules composed of patterns presented in GPU instructions. The rules have been developed to detect miners' execution by checking specific patterns of GPU instructions that indicate their presence. The monitoring tool is fully scalable and does not add processing overhead to the execution of the application. Finally, the PCIe monitoring tool achieved better performance than the corresponding state-of-the-art implementation in a CPU by an overall speedup in execution time of 1.55, and it is more

energy efficient by 543 times the CPU's implementation with a minimum amount of resources needed.

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

# Περίληψη

Τμήμα Ηλετρκολόγων Μηχανικών και Μηχανικών Ηλεκτρονικών Υπολογιστών

Παρακολούθηση της **PCIe** θύρας για ασφαλή εκτέλεση κώδικα σε αρχιτεκτονικές ετερογενών συστημάτων

από Ιωάννη-Ιάσων Γεωργακά

Οι αρχιτεκτονικές ετερογενών συστημάτων γίνονται όλο και πιο δημοφιλείς για την εκτέλεση περίπλοκων και υπολογιστικά μεγάλου φόρτου εφαρμογών, πολλές εταιρείες έχουν τεράστια κέντρα δεδομένων αποτελούμενα από χιλιάδες τέτοια ετερογενή συστήματα ώστε να παρέχουν τη δυνατότητα σε ερευνητές και επαγγελματίες να εκτελέσουν τις εφαρμογές σε διάφορους επιταχυντές όπως GPU,FPGA και TPUs. Όμως αυτή η αύξηση στη χρήση τέτοιων συστημάτων ελλοχεύει το κίνδυνο να αποτελέσουν τα ετερογενή συστήματα, μια επιφάνεια επίθεσης με διάφορες δυνατότητες. Οι ευπάθειες που έχουν οι GPUs και οι τρόποι που μπορούν να τις εκμεταλλευθούν αποτελούν χαρακτηριστικό παράδειγμα τέτοιων επιθέσεων. Ένα τέτοιο παράδειγμα αποτελεί μία επίθεση σε μία GPU χρησιμοποιώντας κακόβουλα έναν **miner** εκτελώντας τον συνεχώς, υπερφορτώνοντας την έτσι και βγάζοντας την εκτός λειτουργία για αρκετά μεγάλο χρονικό διάστημα. Σε αυτή τη διπλωματική εργασία παρουσιάζεται μια καινούργια μέθοδος ανίχνευσης επιθέσεων σε ετερογενή συστήματα παρακολουθώντας και ελέγχοντας την κίνηση των δεδομένων του PCIe πρωτοκόλλου που κατευθύνονται από τον Host στο PCIe endpoint το οποίο λειτουργεί ως ο επιταχυντής. Παρακολουθώντας και ελέγχοντας την κίνηση των δεδομένων του PCIe πρωτοκόλλου, δίνεται η δυνατότητα να ελέγχονται όλα τα δεδομένα και ο κώδικας που μεταφέρονται στο PCIe endpoint σε αληθινό χρόνο και να ανιχνεύεται αν είναι κακόβουλα. Η κίνηση των δεδομένων μέσω του PCIe πρωτοκόλλου ελέγχεται με βάση ένα σετ από κανόνες το οποίο υποδηλώνει κακόβουλες προθέσεις. Στη παρούσα διπλωματική εργασία η περίπτωση που εξετάζεται είναι η παρακολούθηση και ο έλεγχος της κίνησης των δεδομένων σε ένα εξομοιωμένο σύστημα CPU-GPU χρησιμοποιώντας ένα MPSoC και μία FPGA για την εκτέλεση ενός **miner** σε μια GPU βάσει ενός σετ από κανόνες οι οποίοι αποτελούνται από ακολουθίες εντολών GPU. Οι κανόνες έχουν δημιουργηθεί για να ανιχνεύουν την εκτέλεση **miners** ελέγχοντας συγκεκριμένες

ακολουθίες εντολών των GPU οι οποίες υποδεικνύουν την εκτέλεση τους. Ο μηχανισμός παρακολούθησης είναι πλήρως κλιμακούμενος και δεν προσθέτει επιπλέον φόρτο επεξεργασίας κατά την εκτέλεση της εφαρμογής. Ο μηχανισμός παρακολούθησης επιτυγχάνει καλύτερη απόδοση από την αντίστοιχη υλοποίηση σε CPU με τελική επιτάχυνση 1.55 και ενεργειακά αποδοτικότερος 543 φορές από τη CPU χρησιμοποιώντας ελάχιστους πόρους.

# *Acknowledgements*

Starting this thesis, I thank Prof. Sotirios Ioannidis (TUC) for trusting me to complete this assignment and being my supervisor. I would also like to thank Dr. Konstantinos Georgopoulos and Dr. Eva Papadogiannaki for providing guidance and support during the work and writing of this thesis. I would also like to thank Prof. Apostolos Dollas (TUC) for being the committee's third member and evaluating my thesis.

Additionally, I would like to express my gratitude to all the Microprocessors and Hardware Lab (MHL) members for covering all the necessary needs this thesis required.

Finally, I would like to thank my family and friends for their support over the years. This thesis is dedicated to them. ...

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **ALU** | **A**rithmetic **L**ogic **U**nit |
| **ASIC** | **A**pplication **S**pecific **I**ntegrated **C**ircuit |
| **BRAM** | **B**lock **R**andom **A**ccess **M**emory |
| **CPU** | **C**entral **P**rocessor **U**nit |
| **CS** | **C**omputer **S**cience |
| **DDR4** | **D**ouble **D**ata **R**ate type texbf4 memory |
| **DRAM** | **D**ynamic **R**andom **A**ccess **M**emory |
| **DSP** | **D**igital **S**ignal **P**rocessor |
| **FF** | **F**lip **F**lops |
| **FPGA** | **F**ield **P**rogrammable **G**ate **A**rray |
| **GDDR6** | **G**raphics **D**ouble **D**ata **R**ate type **6** memory |
| **GPU** | **G**raphic **P**rocessor **U**nit |
| **HBM** | **H**igh **B**andwidth **M**emory |
| **HDL** | **H**ardware **D**escription **L**anguage |
| **HLS** | **H**igh **L**evel **S**ynthesis |
| **HPC** | **H**ight **P**erformance **C**omputing |
| **LUT** | **L**ook **U**p **T**able |
| **MPSoC** | **M**ulti **P**rocessor **S**ystem **o**n **C**hip |
| **PL** | **P**rogrammable **L**ogic |
| **PS** | **P**rocessing **S**ystem |
| **RAM** | **R**andom **A**ccess **M**emory |
| **SDK** | **S**oftware **D**evelopment **K**it |
| **SIMD** | **S**ingle **I**nstruction **M**ultiple **D**ata |
| **SSE** | **S**treaming **S**IMD **E**xtensions |
| **SSD** | **S**olid **S**tate **D**rive |
| **TDP** | **T**hermal **D**esign **P**ower |
| **URAM** | **U**ltra **R**andom **A**ccess **M**emory |
| **USD** | **U**nited **S**tates **D**ollar |

*Dedicated to my family and friends…*

# Chapter 1

# Introduction

## 1.1 Motivation and Scientific Contributions

Heterogeneous systems architectures are becoming increasingly popular for running complex and heavy-load computational tasks, and many companies have vast data centers consisting of thousands of heterogeneous systems to allow developers to run their applications in GPUs, FPGA, or even TPUs. However, this rise in the use of such systems has led to the emergence of a new vast attack vector with various possibilities. GPU vulnerabilities and their exploitations are realistic instances of such attacks. Miners may be misused to overload the GPU and take them out for an essential period.

Companies, such as Microsoft and CISCO, report incidents [1] [2] regarding abusing their GPUs to execute crypto miners without the companies' knowledge and permission. The mitigation used is mainly focused on using software tools that can always be bypassed or disabled by a malicious user, letting their GPUs be exposed to such attacks.

This thesis introduces a novel method for detecting attacks in heterogeneous systems by monitoring the PCIe traffic at a low level from the host to the corresponding PCIe endpoint, which functions as the accelerator. The PCIe monitoring tool provides real-time monitoring of the code execution in the PCIe endpoint using a pattern-matching algorithm for optimal performance. It alerts the system when a malicious pattern is found in the data transferred to the PCIe endpoint in a similar way an Intrusion Detection System would do based on a set of rules. This thesis use case is to monitor the PCIe traffic in emulating a heterogeneous system CPU-GPU for any GPU miner running in the discrete GPU based on a set of rules composed of patterns presented in GPU instructions.

The contributions of this thesis are :

- We propose a real-time monitoring mechanism through PCIe protocol.
- We develop a custom set of rules composed of patterns of GPU instructions to detect GPU miners
- The use of the monitoring system does not add any extra processing overhead in the execution of GPU applications
- We achieve better performance in the execution time, by a factor of x1.55, when compared to the state-of-the-art software tool such as YARA tool
- The implementation of the monitoring tool requires a minimum amount of FPGA resources

## 1.2 Thesis Outline

- **Chapter 1 - Introduction:** This chapter provides an introduction to the motivation and scientific contributions of this th,esis as well as the outline of the thesis.
- **Chapter 2 - Brief Theoretical Background:** This chapter provides knowledge that is considered useful for understanding this thesis's points.
- **Chapter 3 - Related Work:** This chapter provides several papers that relate to the thesis work and presents how they are considered relevant to this work
- **Chapter 4 - Platforms and Tools:** This chapter analyzes the tools used for this thesis's implementation.
- **Chapter 5 - System Architecture:** This chapter describes the general flow of the architecture and how each of the components is implemented.
- **Chapter 6 - Evaluation:** This chapter presents the final results of the implementation that is described in the **Chapter 5**
- **Chapter 7 - Conclusions and Future Work:** This chapter proposes ideas about optimizations that can be done in the future and ideas to enhance functionality.

# Chapter 2

# Brief Theoretical Background

## 2.1 Field Programmable Gate Array (FPGA)

### 2.1.1 Overview

The widespread and adaptable nature of Field Programmable Gate Arrays (FPGAs) has led to their popularity in implementing digital circuits over the past few decades. Field programmable gate arrays (FPGAs) are pre-made silicon devices with programmable capabilities, allowing for the creation of nearly any digital circuit or system type. They offer advantages such as low risk, minimal incremental costs, and rapid prototyping, contributing to a competitive edge in time-to-market scenarios. Unlike the prolonged fabrication process required for ASIC designs, FPGAs enable testing ideas and validating concepts using various available tools in the market. Despite their advantages, programming and deploying FPGAs still pose challenges. Major cloud service providers like Amazon AWS, Alibaba, and Huawei now offer high-performance FPGA cloud services. This development makes high-end FPGAs, traditionally expensive resources, more accessible as hardware accelerators for real-world applications.

### 2.1.2 FPGA Architecture

The basic FPGA architecture consists of thousands of fundamental elements called configurable logic blocks (CLBs) surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. Input/output (I/O) blocks interface between the FPGA and external devices.

An individual CLB is made up of several logic blocks. An FPGA's characteristic feature is a lookup table (LUT). A LUT stores a predefined list of logic outputs for any combination of inputs. For instance, LUTs with four to six

input bits are widely used. Standard logic functions such as multiplexers (mux), full adders (FAs), and flip-flops are common.

### 2.1.3 FPGA PCIe IP cores

Xilinx offers various solutions for PCIe communication on an FPGA board from which a developer can choose. The most representative PCIe IP cores are DMA/Bridge Subsystem for PCI Express (PCIe), AXI Bridge for PCI Express Gen3 Subsystem, and FGPA Gen3 Integrated Block for PCI Express.

The most commonly used is the DMA/Bridge Subsystem for PCI Express (PCIe), also known as XDMA due to its ease of deployment and a ready-to-use PCIe driver offered by Xilinx along with test examples [3]. The XDMA IP core has two modes of operation to transfer data to the FPGA: memory-mapped mode and streaming mode using the AXI protocol. Both of the modes of operation leverage the use of DMA engines of scatter-gather type DMA. Figure 2.1 illustrates the block diagram of the XDMA IP core.

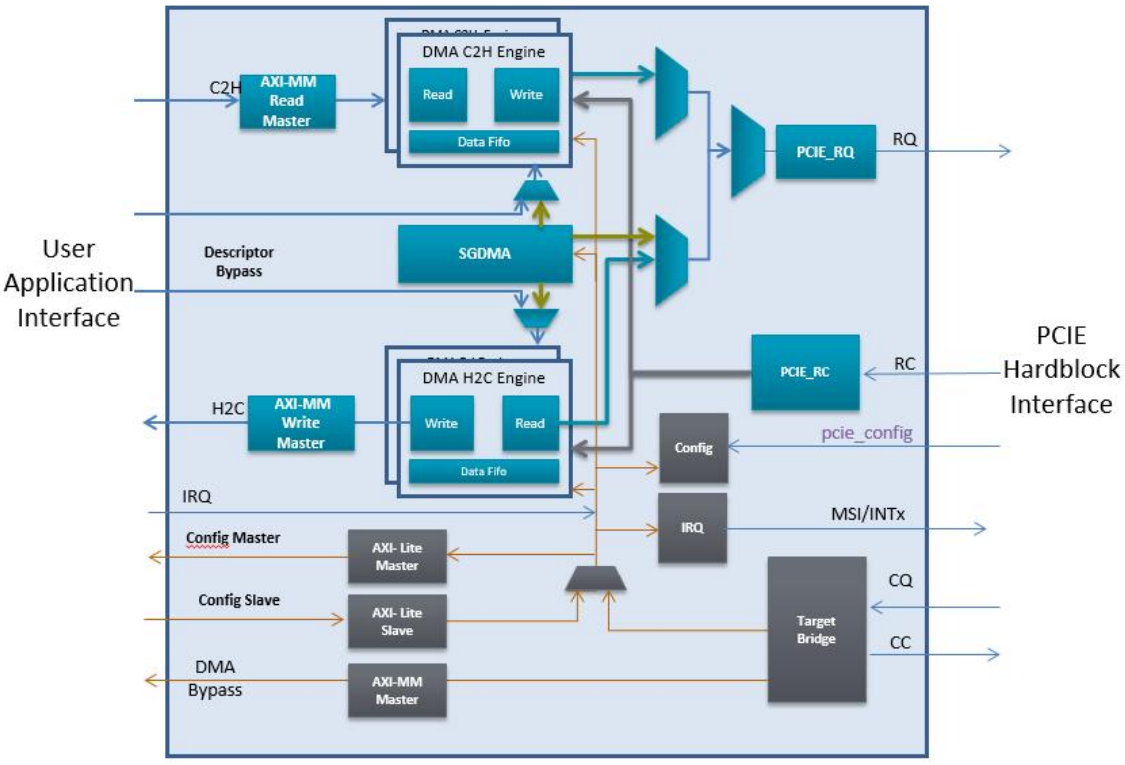


FIGURE 2.1: Block Diagram of XDMA IP Core

The XDMA IP core has several interfaces, offering developers greater design flexibility. Specifically, it provides the developer with two modes: one

memory-mapped mode and another stream mode. In addition to this, it offers a master axilite interface for debug purposes of the corresponding design.

## 2.2 Graphics Processor Unit (GPU)

In the last decade, GPUs have evolved from simple graphics cards to powerful accelerators, making them the central acceleration unit most industry and academia use for heavy computational tasks. Over the years, the hardware architecture of GPUs has changed and improved in terms of performance. The leading GPU vendors, NVIDIA and AMD, have expanded their original architectures to more complicated ones.

### 2.2.1 GPU Hardware Architecture overview

From a high-level hardware perspective, the GPU comprises several streaming multiprocessors (SMs). Each SM is assigned several thread blocks, which contain many wraps. Each warp consists of several threads that execute the same instruction. Figure 2.2 illustrates the GPU architecture in contrast with a conventional CPU's architecture.

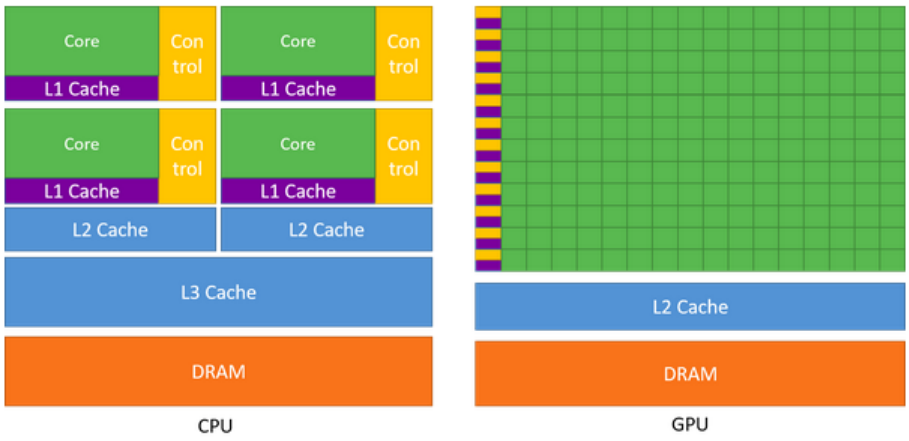


FIGURE 2.2: CPU Vs GPU Architectures

Streaming multiprocessors (SMs) are general-purpose processors with a low clock rate target and a small cache. Their primary task is to execute several thread blocks in parallel. SMs generally support instruction-level parallelism, excluding branch prediction. An SM contains execution cores (e.g., single-precision floating-point units, double-precision floating-point units),

caches (L1 cache, shared memory, constant cache, texture cache), schedulers for wraps, and several registers. Figure 2.3 shows the architecture of a single SM.


FIGURE 2.3: GPU Streaming multiprocessor's architecture

A thread block is composed of warps. A warp is a set of 32 threads within a thread block, and all the threads in a warp execute the same instruction. The SM selects these threads serially. Each warp is executed in a SIMD fashion and is assigned to a warp scheduler. Figure 2.4 illustrates a warp scheduler.

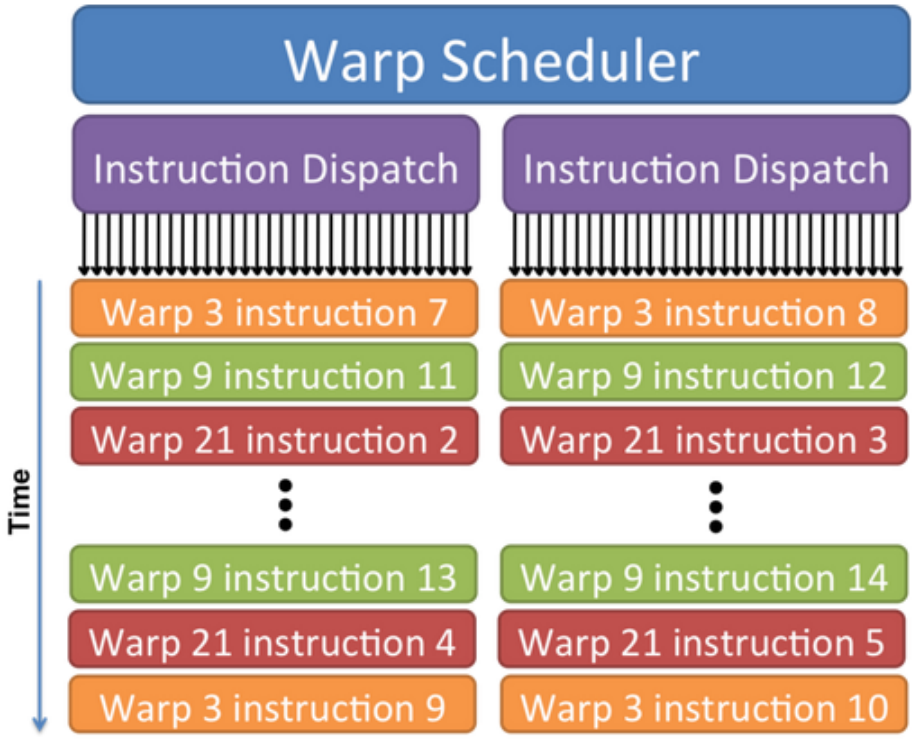


FIGURE 2.4: GPU wrap scheduler

A system containing a CPU and a GPU is well known as a heterogeneous system, other examples are systems containing a CPU and an FPGA, an FPGA and an FPGA, or a GPU and an FPGA. These systems are considered heterogeneous systems because they are composed of more than one acceleration unit. The communication between the CPU and GPU is through the PCIe interface and its corresponding protocol. Figure 2.5 illustrates a typical heterogeneous system architecture.

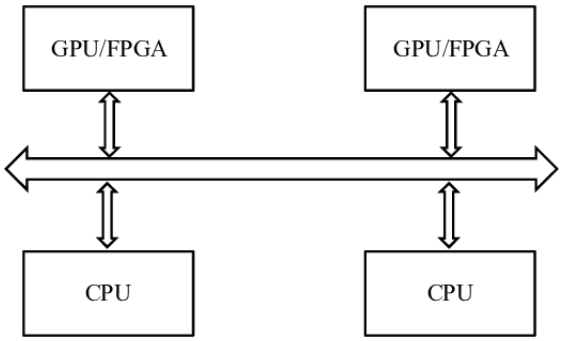


FIGURE 2.5: A typical Heterogeneous System Architectures

### 2.2.2 CPU-GPU Communication through PCIe

The CPU and GPU communication is achieved using a GPU driver, which handles all the low-level tasks and resource allocation. In an abstract overview, the CPU and GPU share a unified virtual address space for communication, constructed by several mechanisms like DMA, PCIe BARS, and MMIO. Initially, the CPU communicates with the GPU using the PCIe BARs dedicated to it, which function as windows in the GPU's device memory. More specifically, PCIe BARs declare dedicated regions in the physical memory shared by the CPU and GPU for communication. CPU writes to these memory regions as MMIO in order not to waste cycles transferring the data to GPU. Still, in contrast, the DMA engines are responsible for transmitting the data to GPU device memory.

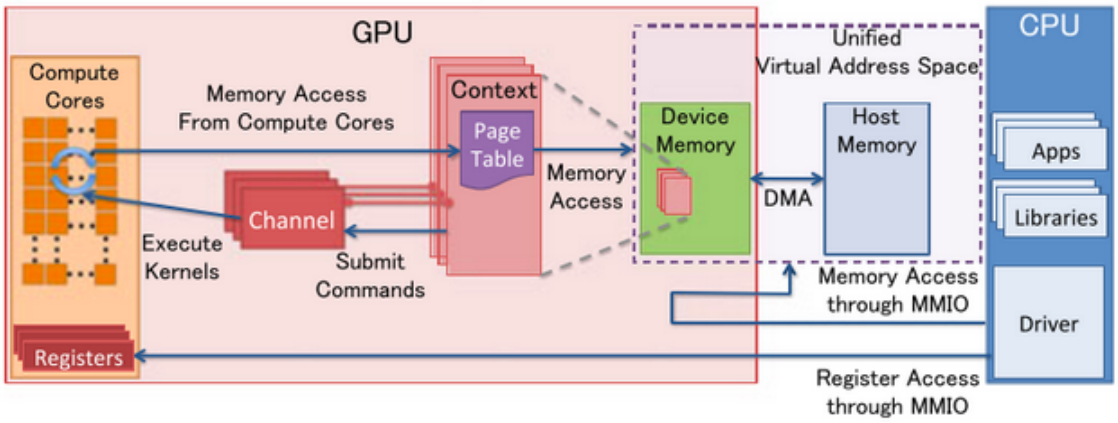


FIGURE 2.6: CPU-GPU communication

### 2.2.3 GPU ISA Architecture

Hardware Instruction Set Architecture (ISA) of GPUs is considered vendor intellectual property and is being disclosed. As a result, there is no official document from the GPU's vendors explaining and analyzing their architectures, so many researchers have tried to reverse engineer GPU's hardware architecture. To understand the underlying GPU's architecture, researchers have been attempting to reverse engineer the corresponding driver of GPU and, more specifically, NVIDIA's official driver along with the communication with the CPU.

Consequently, getting an abstract overview of the hardware ISA that describes how GPU functions at a low level has been achieved. Researching

efforts [4],[5],[6],[7],[8] achieved to decode a significant amount of ISA of NVIDIA GPUs.

Figure 2.7,2.8,2.9 illustrates the results of the efforts to reverse engineer the ISA architecture of NVIDIA GPUs.

Figure 2.7 shows standard instructions used in GPU and their effects. The operand regX refers to 32-bit general registers, operand pX refers to 1-bit predicate registers, and the operand composite refers to operands with multiple possible types.

| Instruction | Effect |
|---|---|
| MOV reg1, comp | reg1 ⇐ comp |
| S2R reg1, special_reg | reg1 ⇐ special_reg |
| IADD reg1, reg2, comp | reg1 ⇐ reg2+comp |
| IMUL reg1, reg2, comp | reg1 ⇐ reg2×comp |
| IMAD reg1, reg2, comp, reg4<br>IMAD reg1, reg2, reg4, comp | reg1 ⇐ reg2×comp+reg4<br>reg1 ⇐ reg2×reg4+comp |
| PSETP p2, p1, p3, p4, p5 | p2 ⇐ p3 LOP p4 LOP p5;<br>p1 ⇐ !p2 |
| BRA const/lit comp | PC ⇐ const/lit comp |
| CAL const/lit comp | callstack.push(PC);<br>PC ⇐ const/lit comp |
| RET | PC ⇐ callstack.pop() |
| LD reg1, [reg2 + 32-bit lit] | reg1 ⇐ [reg2 + 32-bit lit] |
| ST [reg2 + 32-bit lit], reg1 | [reg2 + 32-bit lit] ⇐ reg1 |

FIGURE 2.7: GPU Instructions effects

Figure 2.8 explains the mapping between the bit-sequence and the operand/opcode/modifier in the assembly instruction for different compute capabilities on CUDA architecture.

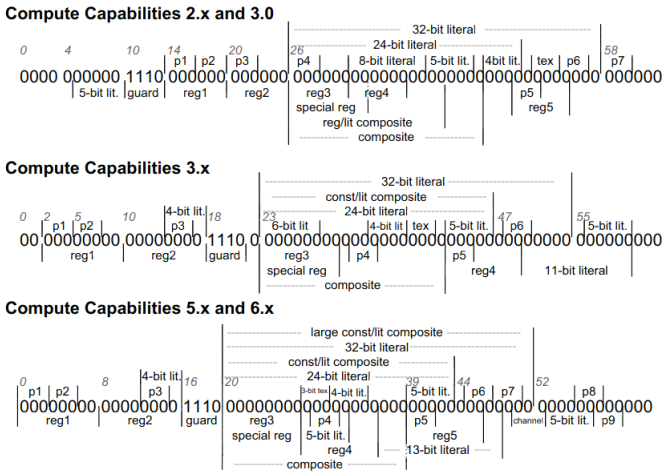


FIGURE 2.8: GPU common operands on different CUDA Architectures

Figure 2.9 illustrates the scheduling information of each group for several instructions on Kepler's GPU architecture. The scheduling information varies from architecture to architecture.

| 0x08a088808c8c10bc | |
|---|---|
| MOV R1, c[0x0][0x44]; | 0x2f |
| S2R R0, SR_CTAID.X; | 0x04 |
| ISUB R1, R1, 0x30; | 0x23 |
| S2R R3, SR_CTAID.Y; | 0x23 |
| S2R R4, SR_TID.Y; | 0x20 |
| IMUL R2, R0, c[0x0][0x28]; | 0x22 |
| S2R R5, SR_TID.X; | 0x28 |

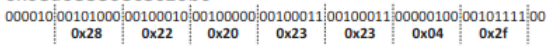


FIGURE 2.9: GPU scheduling information of instructions

### 2.2.4 GPU Assembly

In general, GPU assembly has two types in frameworks like CUDA or OpenCL: one intermediate, also known as PTX (Parallel Thread Execution), which various developers can use to optimize their code, and another named SASS (), which symbolizes the actual assembly that will convert to machine code and be sent to the GPU to be executed. Two figures of the same CUDA code translated to PTX and SASS assembly can be shown.

```
code for sm_30
        Function : _Z6vecAddPdS_S_i
.headerflags    @"EF_CUDA_SM30 EF_CUDA_PTX_SM(EF_CUDA_SM30)"
                                                                        /* 0x2202e2c2828232b7 */
/*0008*/                   MOV R1, c[0x0][0x44];                      /* 0x2800400110005de4 */
/*0010*/                   S2R R0, SR_CTAID.X;                        /* 0x2c00000094001c04 */
/*0018*/                   S2R R3, SR_TID.X;                          /* 0x2c0000008400dc04 */
/*0020*/                   IMAD R0, R0, c[0x0][0x28], R3;             /* 0x20064000a0001ca3 */
/*0028*/                   ISETP.GE.AND P0, PT, R0, c[0x0][0x158], PT; /* 0x1b0e40056001dc23 */
/*0030*/               @P0 EXIT;                                      /* 0x80000000000001e7 */
/*0038*/                   ISCADD R2.CC, R0, c[0x0][0x140], 0x3;      /* 0x4001400500009c63 */
                                                                        /* 0x22c04282c04282b7 */
/*0048*/                   MOV32I R9, 0x8;                            /* 0x1800000020025de2 */
/*0050*/                   IMAD.HI.X R3, R0, R9, c[0x0][0x144];       /* 0x209280051000dce3 */
/*0058*/                   ISCADD R4.CC, R0, c[0x0][0x148], 0x3;      /* 0x4001400520011c63 */
/*0060*/                   LD.E.64 R2, [R2];                          /* 0x8400000000209ca5 */
/*0068*/                   IMAD.HI.X R5, R0, R9, c[0x0][0x14c];       /* 0x2092800530015ce3 */
/*0070*/                   LD.E.64 R4, [R4];                          /* 0x8400000000411ca5 */
/*0078*/                   ISCADD R8.CC, R0, c[0x0][0x150], 0x3;      /* 0x4001400540021c63 */
                                                                        /* 0x20000002e04293f7 */
/*0088*/                   IMAD.HI.X R9, R0, R9, c[0x0][0x154];       /* 0x2092800550025ce3 */
/*0090*/                   DADD R6, R4, R2;                           /* 0x4800000008419c01 */
/*0098*/                   ST.E.64 [R8], R6;                          /* 0x9400000000819ca5 */
/*00a0*/                   EXIT;                                      /* 0x80000000000001de7 */
/*00a8*/                   BRA 0xa8;                                  /* 0x4003ffffe0001de7 */
/*00b0*/                   NOP;                                       /* 0x4000000000001de4 */
/*00b8*/                   NOP;                                       /* 0x4000000000001de4 */
        ..........................
```

FIGURE 2.10: GPU SASS Assembly

Figure 2.10 shows a snippet code of GPU SASS assembly, which at the right, except for the hex value of the corresponding GPU instruction, there is a hex value, a control code between every several instructions. This control code is used for control policy, which decides which instruction in which thread and block to be executed.

```
.reg .pred      %p<2>;
.reg .b32       %r<9>;
.reg .b64       %rd<4>;


ld.param.u64    %rd1, [getlaneid_param_0];
ld.param.u32    %r2, [getlaneid_param_1];
mov.b32 %r3, %envreg3;
mov.u32         %r4, %ctaid.x;
mov.u32         %r5, %ntid.x;
mad.lo.s32      %r6, %r4, %r5, %r3;
mov.u32         %r7, %tid.x;
add.s32         %r1, %r6, %r7;
setp.ge.s32     %p1, %r1, %r2;
@%p1 bra        BB0_2;

// inline asm
mov.u32 %r8, %laneid;
// inline asm
mul.wide.s32    %rd2, %r1, 4;
add.s64         %rd3, %rd1, %rd2;
st.global.u32   [%rd3], %r8;
```

FIGURE 2.11: GPU PTX Assembly

Various groups have made many research efforts to decode the GPU assembly, most of which have focused on NVIDIA GPU and decoding the executable produced by the CUDA compiler, known as nvcc. As a result, it has successfully decoded a significant amount of GPU assembly versions. The papers [4] are the most characteristic instances of such efforts, which decode the GPU assembly produced by various versions of the CUDA compiler.

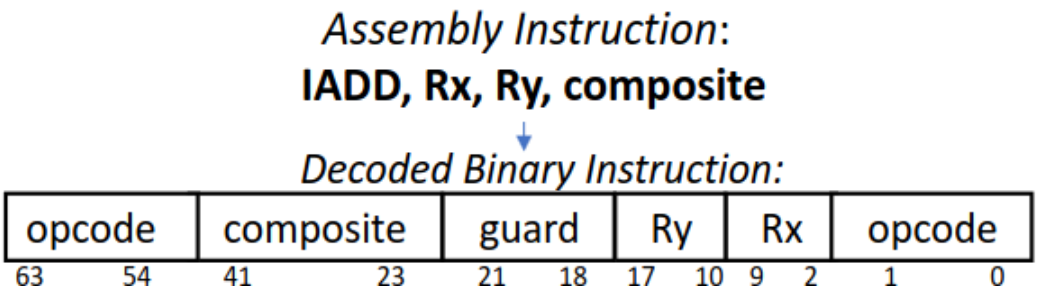


FIGURE 2.12: Decoded IADD instruction for SM35

Figure 2.12 illustrates how an IADD instruction' format in 64-bit is being organized for a specific version of CUDA compute capability. The first 10 bits from 63 to 54 are the opcode of the instruction, the bits 41 to 23 are being

used to store the composite, and the guard is represented by the bits 21 to 18, Rx and Ry, with the corresponding bits 17 to 10 and 9 to 2 and the last 2 bits are also used for the opcode of the instruction.

## 2.3 Peripheral Component Interconnect Express(PCIe) Protocol

PCI Express is a high-performance, general-purpose I/O interconnect defined for various computing and communication platforms. PCI Express takes advantage of recent advances in point-to-point interconnects, Switch-based technology, and packetized protocol to deliver new levels of performance and features. Power Management, Quality Of Service (QoS), Hot-Plug/Hot-Swap support, Data Integrity, and Error Handling are among some of the advanced features supported by PCI Express.

### 2.3.1 PCIe Fabric Topology

A fabric is composed of point-to-point Links that interconnect a set of components. Figure 2.13 shows an example of such topology.

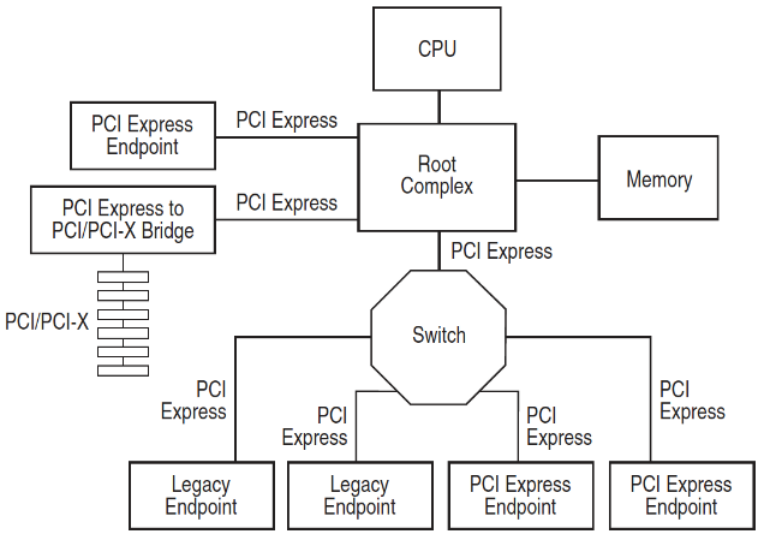


FIGURE 2.13: Example of PCIe fabric Topology

This figure illustrates a single fabric instance called a hierarchy, composed of a Root Complex, multiple Endpoints (I/O devices), a Switch, and a PCI Express to PCI/PCI-X Bridge, all interconnected via PCI Express links.

A Root Complex denotes the root of the I/O hierarchy that connects the CPU/memory subsystem to the I/O. As illustrated in Figure 1.2, a Root Complex may support one or more PCI Express ports. Each interface defines a separate hierarchy domain. Each hierarchy domain may comprise a single Endpoint or a sub-hierarchy containing one or more Switch components and Endpoints. Endpoint refers to a type of Function that can be the Requester or the Completer of a PCI Express transaction either on its behalf or on behalf of a distinct non-PCI Express device (other than a PCI device or Host CPU), e.g., a PCI Express attached graphics controller or a PCI Express-USB host controller. Endpoints are either legacy, PCI Express, or Root Complex Integrated Endpoints.

### 2.3.2 PCIe Layers Overview

PCIe consists of three layers: the Transaction Layer, the Data Link Layer, and the Physical Layer. Each of these layers is divided into two sections: one that processes outbound (to be transmitted) information and one that processes inbound (received) information, as shown in Figure 2.14

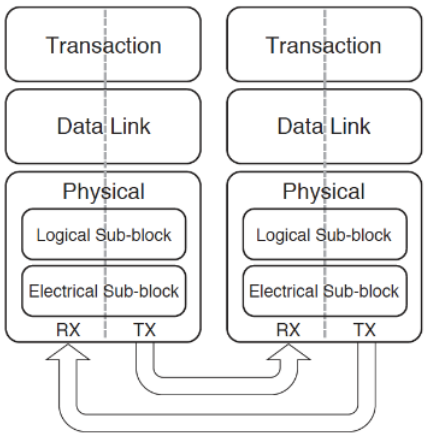


FIGURE 2.14: PCIe Layers

PCIe uses packets to communicate information between components. Packets are formed in the Transaction and Data Link Layers to carry the information from the transmitting component to the receiving component. As the transmitted packets flow through the other layers, they are extended with additional information necessary to handle packets at those layers. On the receiving side, the reverse process occurs, and packets get transformed from their Physical Layer representation to the Data Link Layer representation and

finally (for Transaction Layer Packets) to the form that the Transaction Layer of the receiving device can process.

### 2.3.3 PCIe Communication

The PCIe communication from the operating system's (OS) perspective is achieved through dedicated drivers. During the system's boot process, the OS enumerates all the PCIe devices connected to it and dedicates specific memory regions in its physical memory for the system's communication with the PCIe devices. These memory regions are exposed from the OS with special registers, better known as Base Address Registers(BARs), which indicate each memory region's start and corresponding offsets. The appropriate device driver of each PCIe device is responsible for providing an API library for the user to communicate with the corresponding PCIe device. The drivers use special routines to write to these memory regions as memory-mapped input-output (MMIO), and then the data is transferred to the PCIe devices using DMA engines. This happens to optimize the performance of the CPU instead of letting the CPU transfer the data directly to the PCIe device, which would lead to several cycles of waste.

## 2.4 Monitoring Algorithms

Various tools and algorithms are available for detecting malware or malicious code based on a set of predefined rules. The Yara and Snort tools are two characteristic examples widely used in academia and industry. These tools are based on Aho-Corasick's algorithm, which is considered to be state-of-the-art due to its fundamental property, which is that the time of execution is according only to the input and not from the number of rules to detect. However, other algorithms, such as hashing and cross-correlation methods, may be used depending on the case.

### 2.4.1 Aho-Corasick

In general, Aho-Corasick is the most efficient and suitable mechanism to detect malware signatures in the industry(e.g., YARA and SNORT tools) and academia since the algorithm is considered to be state-of-the-art. It leverages the power of regular expressions and can detect signatures in many diverse file formats. For example, it can be used to detect if any sequence

of GPU instructions is malicious, given a set of rules that indicate such malicious patterns. Aho–Corasick is one of the most widely used algorithms that perform simple string pattern matching. It is considered the best option for multiple pattern searching since it matches all signatures simultaneously in a single pass. This simultaneous matching can be achieved when the set of patterns is being preprocessed. In the preprocessing phase, an automaton is built, which will be used eventually in the matching phase. Additionally, each text character will be processed only once during the matching phase. The Aho–Corasick algorithm has the property that, theoretically, the processing time does not explicitly depend on the number of patterns. Letting $P = p_1p_2p_3...p_n$ be the patterns to be searched inside a text $T = t_1t_2t_3...t_m$ (with lengths n and m, accordingly), both sequences of characters form a finite character set $\Sigma$. The algorithm's complexity is linear in the pattern length n, the length of the given text m, and the number of output matches.

Given a set of patterns, the algorithm constructs a pattern-matching machine that matches all patterns in the text at once, one byte at a time. Each processing action of the automaton accepts an input event. The very first action starts with the initial state, represented by zero. Each action that accepts an input event moves the current state to the next state based exclusively on that input. There are three distinct functions: (a) a goto function, (b) a failure function, and (c) an output function. According to the input event, one function is being triggered. Figure 2.15 presents an example of these functions for the set of patterns, i.e., he, she, his, hers.

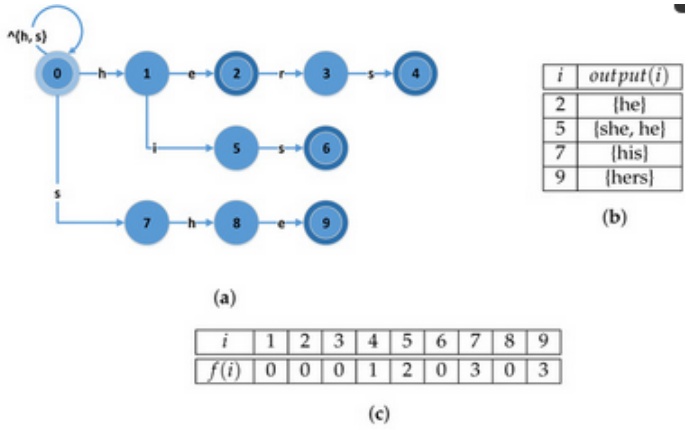


FIGURE 2.15: An example of multi-pattern matching with Aho–Corasick. (a) The Aho–Corasick goto function is presented as a state machine for the patterns she, he, hers, and his. (b) The output function indicates the final state of each pattern in the given set. (c) The Aho–Corasick failure function for the given pattern set.

The goto function (Figure 2a) determines if a state transition can be performed based on the current state and the ASCII value of the input character. If the input character matches one of the transitions starting from the current state, the state pointed to by this transition becomes the next state. Otherwise, the next state is resolved by the failure function $f(i = currentstate)$. For example (based on Figure 2a), the edge labeled h from 0 to 1 indicates that $goto(0, h)$=1, while the absence of an arrow for a indicates failure. The failure function either drives a transition to one or more intermediate states or to the initial state (which is represented with 0 in the goto graph). After each state transition, the algorithm checks the output function $output(i = currentstate)$ to determine if the pattern matches a sub-string of the text T. This procedure continues and terminates at the end of the input text T. Since failed transitions may not consume any input—the so-called $\epsilon$-transitions—the produced automaton is non-deterministic (NFA). The failure function can also result in numerous state transitions for a single input character. In this way, the matching operation might require the exploration of multiple paths before the actual match of a pattern. A revised version of the traditional Aho–Corasick exists and replaces all failure transitions to avoid performance loss when the patterns' sizes are large. The new automaton that is produced is called "deterministic finite automaton" (DFA) and provides one transition per state and input character. Despite this approach requiring

more memory than the previous one, it is more efficient in terms of processing throughput. The achieved complexity of this approach is $O_n$.

### 2.4.2 Cross Correlation

The cross-correlation technique is used in many cases when it is desired to compare two signatures or two signals to decide how similar they are. It is also used in some cases to search for malware signatures or inconsistencies between the accepted signature and the provided one. For instance, it can be used to detect the similarity of two patterns composed of GPU instructions given an input and a specific sequence of GPU instructions.

Cross-correlation measures the similarity of two series as a function of the displacement of one relative to the other. It is also known as a sliding dot product or sliding inner product. It is commonly used to search a long signal for a shorter, known feature. The mathematical equation for two series of signals $x[n], y[m]$ that describes cross-correlation is : $\sum_{n=-\infty}^{\infty} x[n] * y[n-m]$

Figure 2.16 illustrates an example of cross correlation between two series of signals x,y.

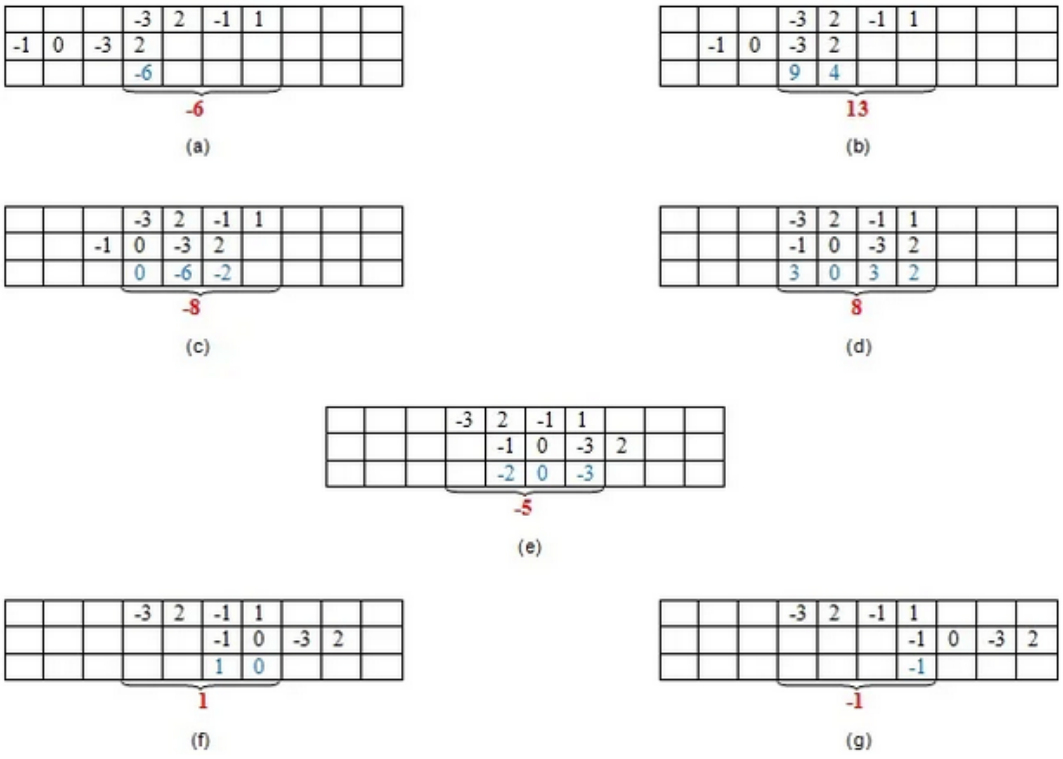


FIGURE 2.16: Cross-Correlation example

In figure 2.16, the two series of numbers are being compared to calculate their similarity. In every step of the algorithm, the one series of numbers shifts by one position, and the sum of the products of the two series is calculated. The one series shifts by one position until the last number of the other series.

# Chapter 3

# Related Work

## 3.1 GPU Security

In recent years, the increasing use of GPUs has emerged as a significant threat to their security as they have evolved from being used for simple graphics tasks such as video and gaming to powerful computational units for heavy loads tasks, for instance, training of CNN, DNN, and machine learning tasks. Consequently, many researchers have begun exploring the capabilities of such attack vectors, focusing on GPU vulnerabilities, GPU attacks, and GPU defense mechanisms.

Andrea Miele [9] presents a preliminary analysis of buffer overflow vulnerabilities in CUDA API running on GPUs. Its work demonstrates how an attacker can overrun a buffer to corrupt sensitive data or steer the execution flow by overwriting function pointers, e.g., manipulating the virtual table of a C++ object.

Bang Di et al. [10] explore security vulnerabilities of CUDA from multiple dimensions. Their work shows a study on GPU stack revealing stack, heap integer, and function pointer overflow can be exploited by manipulating different memory spaces so different threads, blocks, and kernels can be overwritten with other content.

Sang-Ok Park et al. [11] present a novel attack method that makes deep learning predictions no longer different from random guessing by degrading the accuracy of the predictions. Their work demonstrates the first practical attack directly exploiting deep learning frameworks through GPU device memory manipulation. Their attack works on three popular deep learning frameworks, TensorFlow, CNTK, and Caffe, running on CUDA.

Christopher Erb et al. [12] describe a tool to detect buffer overflows caused by GPU kernels performing canary checks similar to CPU. Their work is focused on GPU memory buffer overflows in AMD GPUs, and for this reason, their tool wraps OpenCL API calls and alerts users to any kernel that writes outside of the memory buffer.

Bang Di et al. [13] propose an efficient buffer overflow detector named GMODx, a runtime software system that can detect GPU buffer overflow. This tool monitors allocated memory based on a canary-based design, using several techniques such as delayed memory-free, efficient memory reallocation, garbage collections, and others, improving performance and decreasing overhead compared to other detectors.

In the last couple of years, the trusted execution environment has gained tremendous ground in GPUs as a security defense mechanism, and many researchers have developed and implemented some proof of concepts of GPU TEE.

Stavros Volos et al. [14] propose a GPU Trusted Execution Environment to isolate the execution of the kernels from other code running on the GPU and all software on the host. It proposes modifications in the peripheral components, such as the GPU's command processor, with no changes to existing CPUs, GPU cores, or the GPU's MMU and memory controller. It also proposes extensions to the CUDA runtime for securely copying data and executing kernels on the GPU. Their work is implemented by emulating the new hardware features of the GPU and shows an overhead of around 17%-33% percent.

Insu Jang et al. [15] propose a novel hardware and software architecture as a trusted execution environment. It does not require modifications to the GPU architecture to offer protection. Instead, it offers security by modifying the I/O interconnect between the CPU and GPU and refactoring the GPU device driver to work within the CPU-trusted environment. Their work implements the proposed architecture on an emulated machine with KVM and QEMU, showing that the performance overhead for security is curtailed to 26% on average.

## 3.2 Thesis Approach

GPUs have gained tremendous ground in the last decade, from simple graphics cards to powerful computational units used for heavy loads tasks. This increasing use of them has posed a significant threat to their security. As a result, many surveys and research studies have been developed to explore and illustrate GPU vulnerabilities. Nevertheless, despite all the efforts to mitigate such vulnerabilities and use defense mechanisms such as GPU Trusted Execution Environment to ensure secure code execution in GPUs, most of them are too specific, are implemented at the software level, are complex to deploy, and require tremendous effort or can even be bypassed at the software level.

Building upon the rise of GPU vulnerabilities and the need for an effective way to detect malicious applications targeting the GPUs, a novel notion of defense mechanism can emerge as a solution to ensure secure code execution in heterogeneous system architectures like CPU-GPU, CPU-FPGA, CPU-FPGA-GPU, FPGA-GPU. Monitoring the PCIe traffic of the data being transferred from a host, e.g., CPU, to a GPU where data represents GPU instructions can be a great defense against various attacks.

The state-of-the-art defense approach both in industry and academia to ensure secure code execution in heterogeneous systems architectures is either a Trusted Execution Environment, which adds an extra overhead for the system and the corresponding accelerator, or static analysis of the binary executable code using product tools such as YARA and SNORT, or even in some rare occasions the use of a PCIe analyzer to analyze the PCIe traffic of TLP packets using a distinct tool which costs a lot.

The thesis approach focuses on PCIe monitoring for secure code execution in heterogeneous system architectures that use two FPGAs to emulate a heterogeneous system architecture consisting of CPU-GPU. The aim is to develop a monitoring system that ensures secure and efficient code execution by monitoring the PCIe data traffic between the CPU and GPU and detecting any security issues in real-time. The system will leverage specific GPU instruction patterns of malicious programs at the machine level to detect any attempt to attack the GPU. The proposed approach is expected to enhance the security of heterogeneous system architectures, which are becoming increasingly popular in various fields, including machine learning, data analytics, and high-performance computing.

# Chapter 4

# Platforms and Tools

The platforms and tools used for developing and implementing the PCIe monitoring system consist of two different FPGA boards and three tools provided by Xilinx to develop the hardware architecture and Operating System of the demonstration setup.

## 4.1 Platforms

### 4.1.1 ZYNQ Ultrascale+ MPSoC family of FPGAs

The Xilinx Zynq UltraScale+ MPSoC [16] family merges FPGA technology with multi-core ARM processors, creating a flexible platform for embedded systems. These FPGAs integrate customizable logic, adapting to specific application needs, along with ARM Cortex-A53 and Cortex-R5 processors for real-time processing. This unique combination allows for implementing intricate, high-performance systems on a single chip. The Zynq UltraScale+ MPSoC family finds applications in diverse fields such as automotive, industrial control, communication, and aerospace, providing developers with a scalable and efficient solution for hardware adaptability and robust processing capabilities.

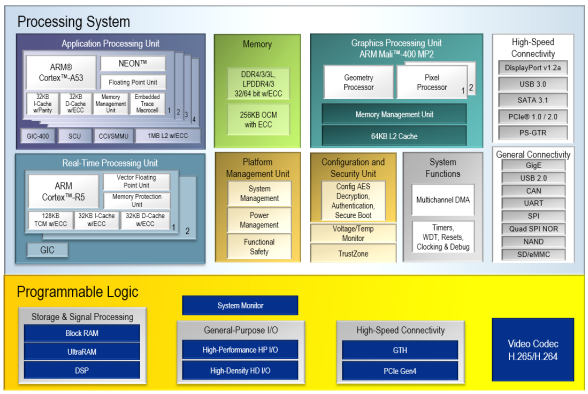


FIGURE 4.1: ZYNQ Ultrascale+ MPSoC Block Diagram [16]

### ZYNQ UltraScale+ MPSoC ZCU102

The ZCU102 [17] provides a rapid prototyping platform using the XCZU9EG-2FFVB1156E device. The ZU9EG contains many useful processor systems (PS) hard block peripherals exposed through the Multi-use I/O (MIO) interface and a variety of FPGA programmable logic (PL), high-density (HD), and high-performance (HP) banks. The ZU9EG device integrates a quad-core Arm® Cortex™-A53 processing system (PS) and a dual-core Arm Cortex-R5 real-time processor, providing application developers unprecedented heterogeneous multiprocessing. The ZCU102 evaluation board provides a flexible prototyping platform with high-speed DDR4 SODIMM and component memory interfaces, FMC expansion ports, multi-gigabit per second serial transceivers, a variety of peripheral interfaces, and FPGA fabric for user-customized designs.


FIGURE 4.2: ZCU102 Board [17]

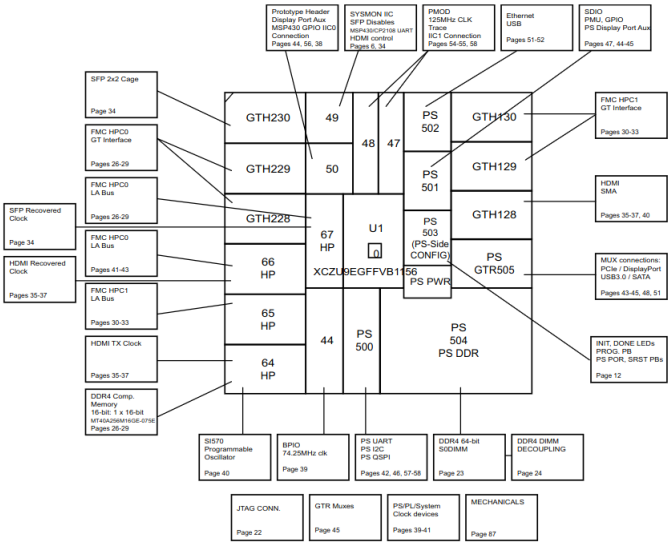


FIGURE 4.3: ZCU102 Block Diagram [17]

### 4.1.2 AMD Kintex 7 Familly of FPGAs

The Kintex family of Field Programmable Gate Arrays (FPGAs) developed by Xilinx, a part of AMD, is optimized for system performance and integration at 28nm, offering exceptional performance per watt, DSP performance, and I/O bandwidth to designs [1]. The Kintex family is used in 10G to 100G networking applications, portable radar, and ASIC Prototyping. The Kintex 7 FPGAs are mainly designed to bring high performance and integration at 28nm and provide up to 2M logic cells, VCXO component, AXI IP, and AMS integration.

The Kintex 7 FPGAs [18] offer increased system performance with up to 2.8 Tb/s total serial bandwidth with up to 96 x 13.1G GTs, up to 16 x 28.05G GTs, 5,335 GMACs, 68Mb BRAM, DDR3-1866. They also significantly reduce the Bill of Materials (BOM) cost, up to 40 percent lower than a multi-chip solution, and total power reduction, up to 50 percent lower power than a multi-chip solution. The Kintex family of FPGAs also offers accelerated design productivity with a scalable optimized architecture, comprehensive tools, IP, and Targeted Design Platforms

#### Kintex 7 KC705 Evaluation Kit

The KC705 evaluation board [19] for the Kintex®-7 FPGA provides a hardware environment for developing and evaluating designs using the device XC7K325T-2FFG900C. In addition, It provides features common to many embedded processing systems, including a DDR3 SODIMM memory, an 8-lane PCI Express® interface, a tri-mode Ethernet PHY, general purpose I/O, a UART interface, etc.

FIGURE 4.4: KC705 Board [19]

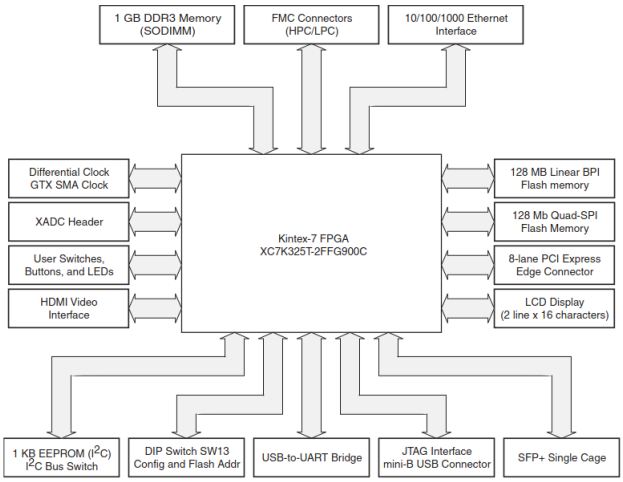


FIGURE 4.5: KC705 Block Diagram [19]

## 4.2 Tools

The essential tools for developing and implementing the system's design are Xilinx Vivado Design Suite [20], Xilinx Vitis HLS [21], and Petalinux [22]. All the tools are used in version 2021.2 to have a stable environment.

### 4.2.1 Xilinx Vivado Design Suite 2021.2

Xilinx Vivado Design Suite [20] is a powerful software-based tool for designing more or less complex hardware designs in Xilinx FPGAs, MPSoCs, and SoCs. It is a complete design suite for FPGAs, providing capabilities such as synthesis, implementation, simulation, and generating bitstreams.

### 4.2.2 Xilinx Vitis HLS 2021.2

The Xilinx Vitis High-Level Synthesis (HLS) [21] tool is a powerful software suite that enables developers to create complex FPGA algorithms by synthesizing C/C++ functions into Register Transfer Level (RTL). This tool is tightly integrated with the Vivado Design Suite for synthesis and place & route and the Vitis unified software platform for heterogeneous system designs and applications. Using the Vitis HLS flow, users can apply directives to the C code to create the RTL specific to a desired implementation. This allows for creating multiple design architectures from the C source code and enables a path for high-quality, correct-by-construction RTL. The Vitis HLS

tool also features rich analysis and debugging tools that facilitate design optimization.

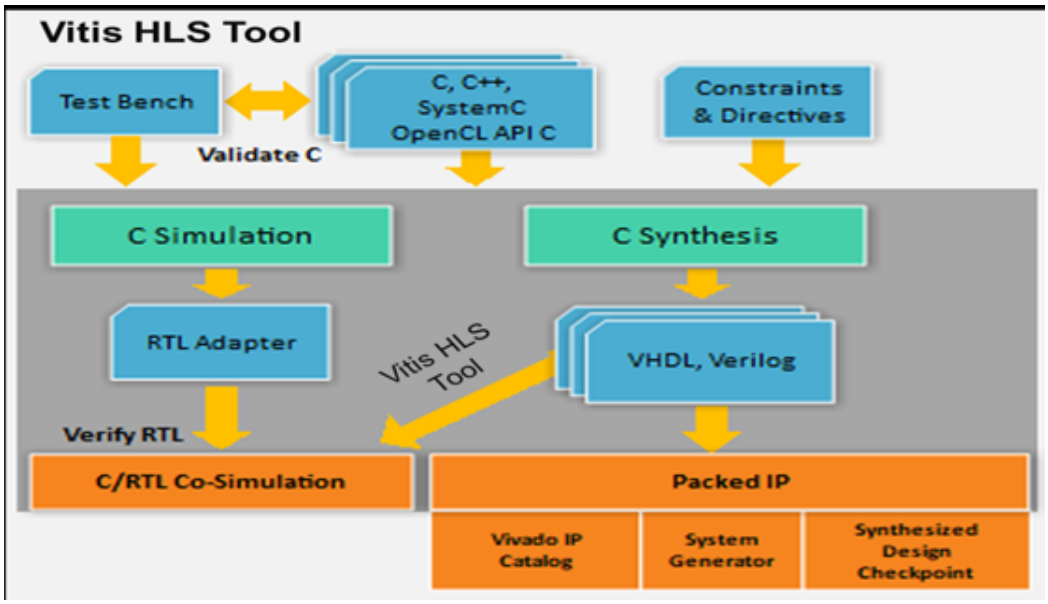


FIGURE 4.6: Vitis HLS work flow [23]

The Vitis HLS tool is designed to take advantage of the benefits and characteristics offered by the architecture of AMD FPGAs. It supports parallel programming constructs to model a desired implementation. These constructs include HLS tasks that allow process-level concurrency, HLS vectors that allow data-level parallelism, and HLS streams that enable communication between concurrent tasks. Synthesis pragmas can be used to control the results. These pragmas include pipeline, unroll, array partitioning, and interface protocols. The output of the Vitis HLS tool is an RTL implementation that can be packaged into a compiled object file (.xo) or exported to an RTL IP.

### 4.2.3 Petalinux 2021.2

Petalinux [22] is an Embedded Linux System Development Kit targeting Xilinx FPGA-based System-on-Chip designs. It is a command line-based tool, there is no graphical user interface (GUI), and it consists of three modules, Yocto Extensible SDK, XSCT (Software Command-Line Tool), and Petalinux Command Line Interface (CLI) tools. In addition to this, it provides the ability to create and customize lightweight Linux OS targeting specific FPGA boards and their processors.

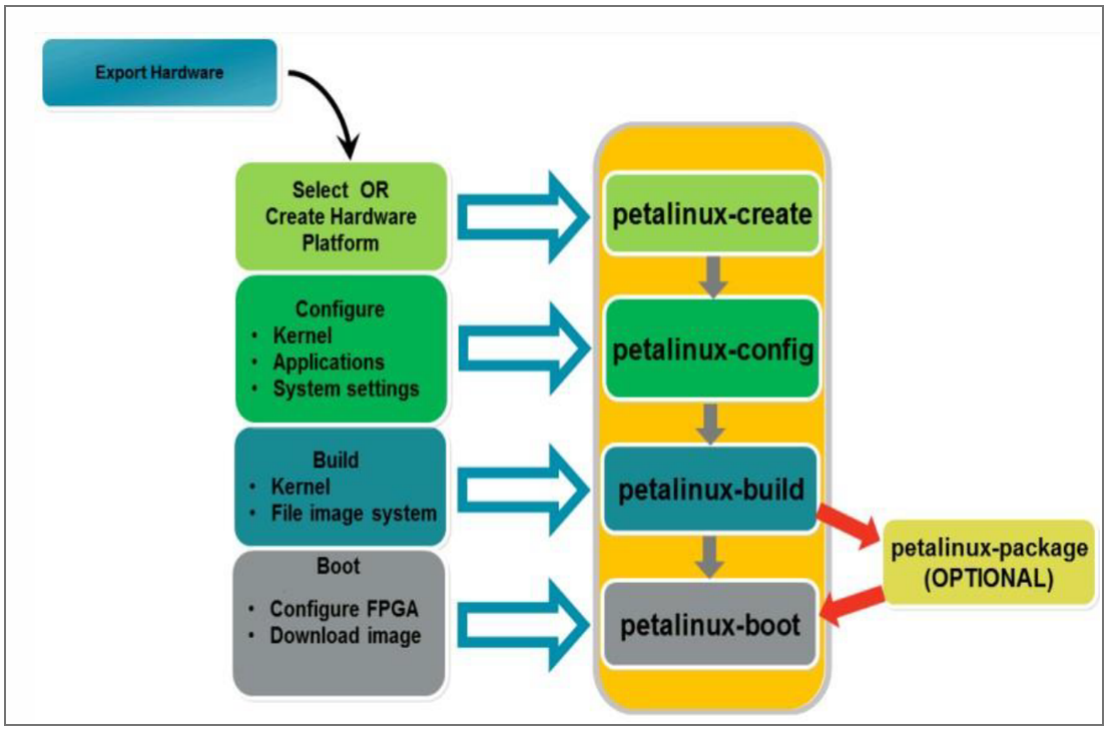


FIGURE 4.7: Petalinux workflow [24]

# Chapter 5

# System Architecture

The system emulates a heterogeneous system architecture composed of a CPU and a GPU by using two FPGA boards to integrate a custom hardware module between the PCIe interconnect and the GPU. The module aims to monitor the GPU code sent to the GPU by the CPU to be executed in real time and detect any malicious patterns of instructions on the machine level.

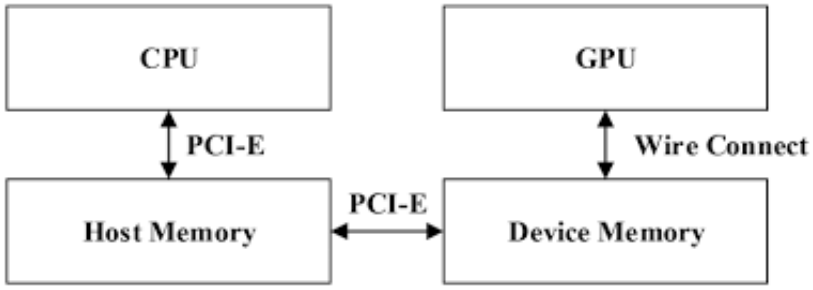


FIGURE 5.1: CPU-GPU Heterogeneous system

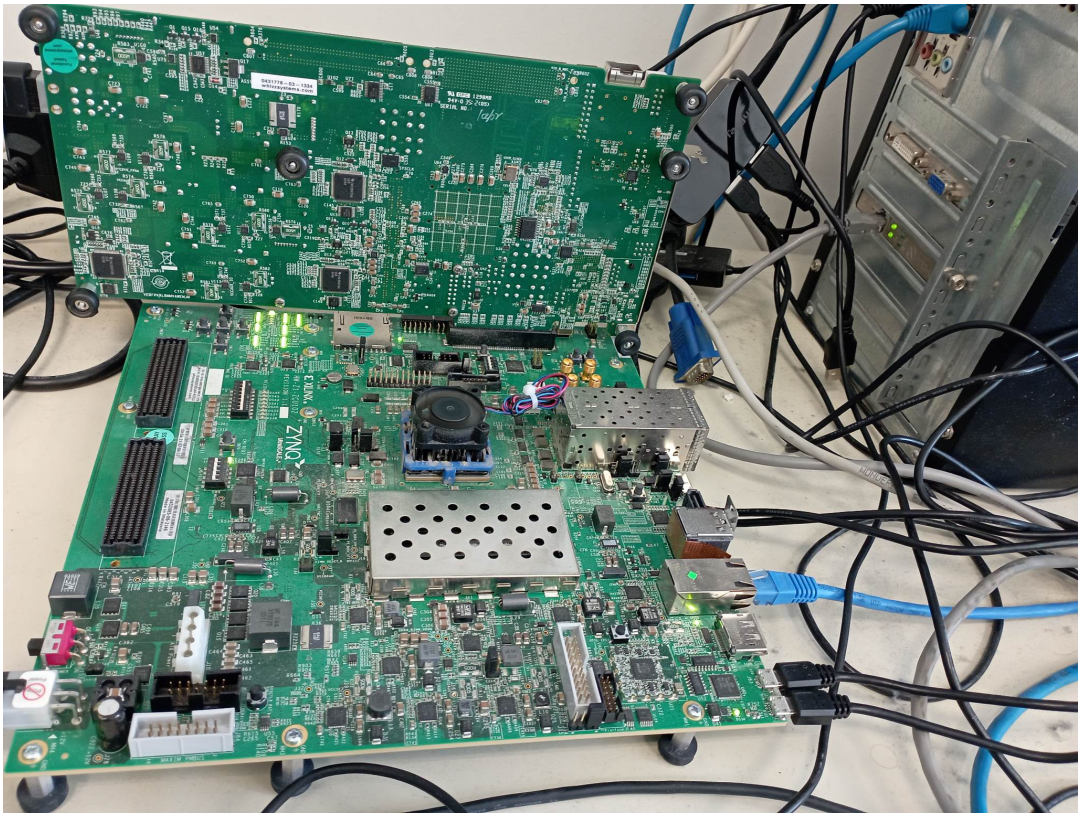

FIGURE 5.2: MPSoC-FPGA Heterogeneous Setup

The system architecture consists of an MPSoC and an FPGA board. The MPSoC emulates the CPU, sending the GPU executable code in binary to the GPU emulation. The FPGA is a PCIe endpoint containing the GPU emulation and the monitoring IP core. Figure 5.2 shows the real demonstration setup of the PCIe monitoring system. In addition, figure 5.3 illustrates the system's emulation of a CPU-GPU in block diagram without the custom hardware monitoring tool and with it.

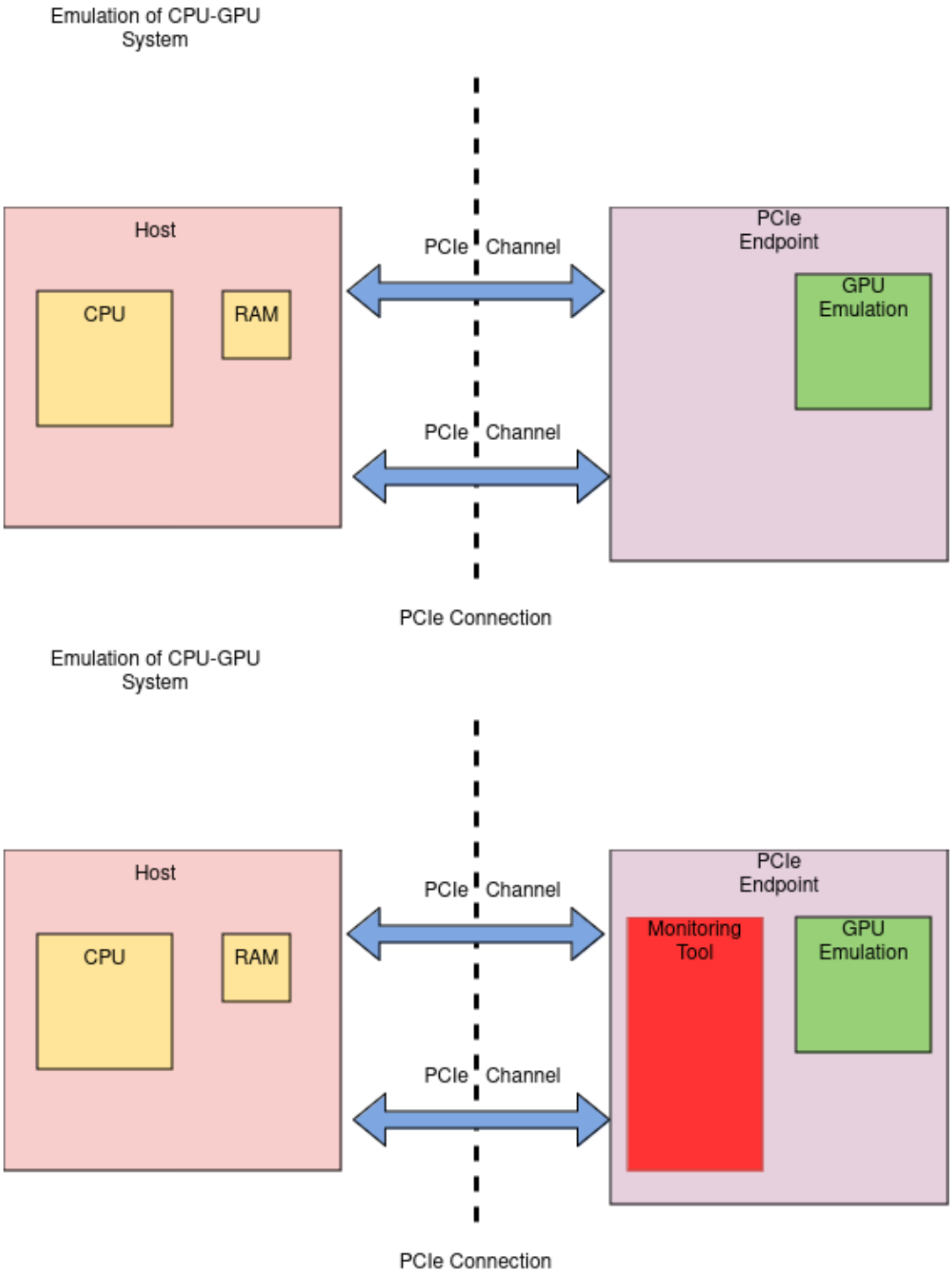


FIGURE 5.3: Block Diagram of system's emulation without and with the monitoring tool

## 5.1 Host

The host is implemented in a zcu102, mainly using its processor and PCIe interconnection. It emulates the data transfer to the GPU through the PCIe communication between its CPU and PCIe EP. In the same way, a CPU sends the GPU code to a typical GPU to execute a GPU program. The following subsections describe the host's hardware design and Operating System.

### 5.1.1 Hardware Design

The block diagram of the host is composed of a processor and its peripheral interconnection with the outside world, e.g., a PCIe connection. Figure 5.4 illustrates the hardware design on ZCU102, in which it can be observed that it doesn't use any of the programmable regions of the MPSoC.


FIGURE 5.4: Block Design of Host

### 5.1.2 Operating System (OS)

The OS running on the host is being developed using Petalinux tools, which support the CPU architecture of ZCU102 and its configurations on the Linux kernel to enable the PCIe libraries and packages on the system. In addition, a PCIe driver specifically for the PCIe IP core, which was used in PCIe EP, needed to be installed, and it was offered from the official GitHub repo of Xilinx.

Furthermore, the operating system is booted using an NFS filesystem through tftpboot. The OS system image is transferred and booted up remotely through the jtag interface using the tftp protocol. After the first-stage boot loader has succeeded, the filesystem is connected to the host through the network and lies on a remote server. Figures 5.5, 5.6 illustrate the boot of the OS in ZCU102.

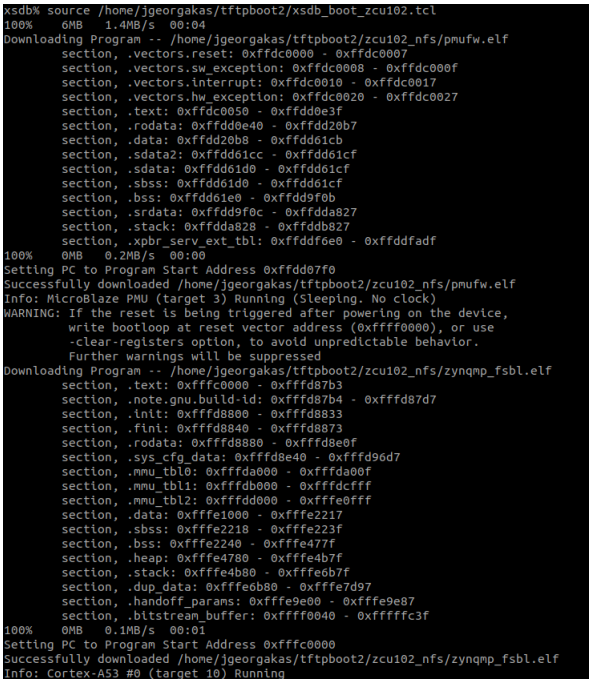


FIGURE 5.5: OS boot(cont.)

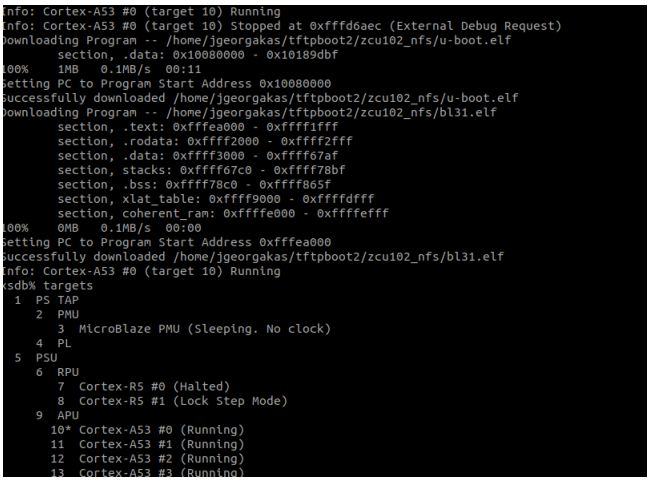


FIGURE 5.6: OS boot

The ZCU102 board, to enable tftpboot, needs to change the configuration of the dip switches appropriately. Figure 5.7 demonstrates the configuration of the dip switches, and it selected the jtag configuration.

*Table 2-4:* **Switch SW6 Configuration Option Settings**

| Boot Mode | Mode Pins [3:0] | Mode SW6 [4:1] |
|---|---|---|
| JTAG | 0000 | on, on, on, on |
| QSPI32 | 0010(1) | on, on, off, on |
| SD | 1110 | off, off, off, on |

**Notes:**
1. Default switch setting.

FIGURE 5.7: Dip switches configuration [17]

Figure 5.8 illustrates the configuration of dip switches to enable tftpboot using NFS filesystem in the board ZCU102.

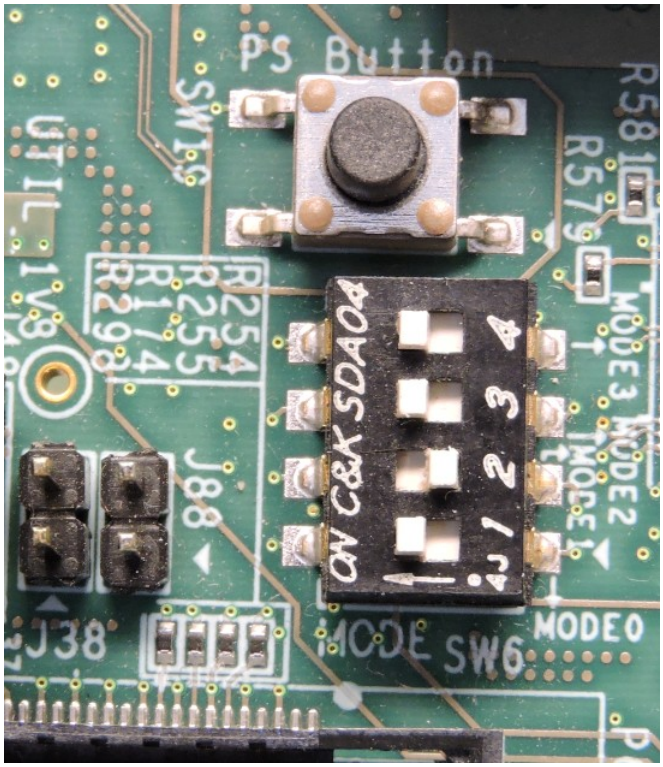


FIGURE 5.8: Dip switches configuration on zcu102 [17]

## 5.2 PCIe Endpoint

The PCIe endpoint emulates the functionality of the GPU and the custom monitoring tool using an FPGA. It comprises several IP cores, the most characteristic being the PCIe IP core, the custom monitoring IP core, and the core for the GPU emulation. Figure 5.9 illustrates the hardware architecture of the PCIe endpoint at an abstract level.

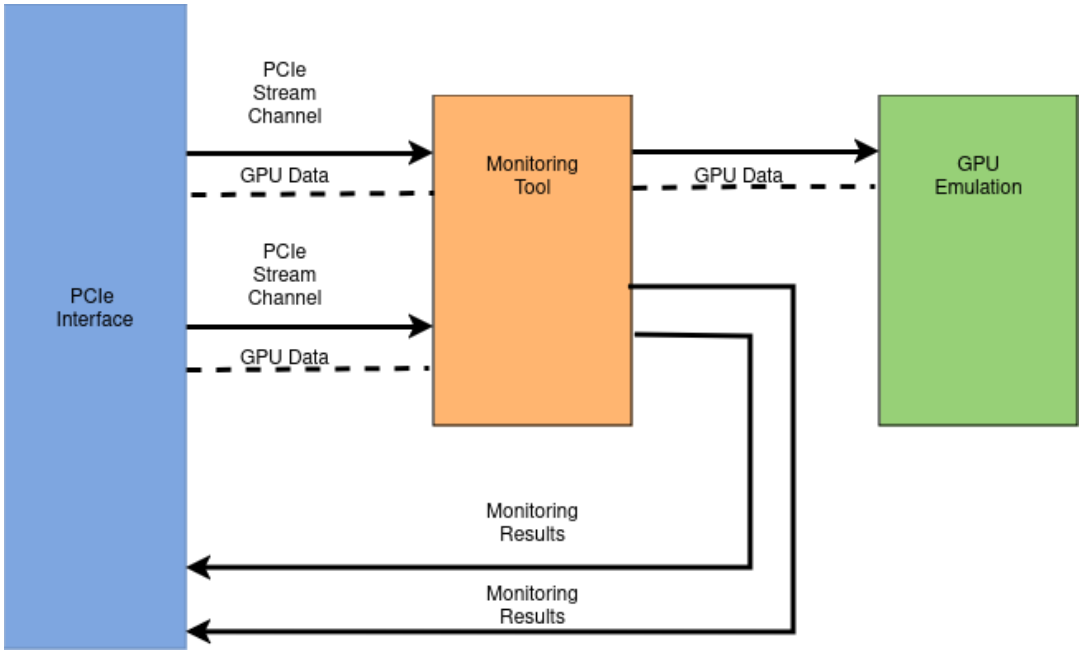


FIGURE 5.9: Hardware Architecture of PCIe endpoint

### 5.2.1 Monitoring Tool Hardware Implementaion

The monitoring tool has been implemented and developed using the Aho-Corasick algorithm, with some modifications to adapt to using GPU instructions instead of characters as input. The monitoring tool comprises several Aho-Corasick IP cores according to the PCIe stream channels of the PCIe interface being used. Figure 5.10 illustrates the hardware architecture of the monitoring tool at an abstract level.

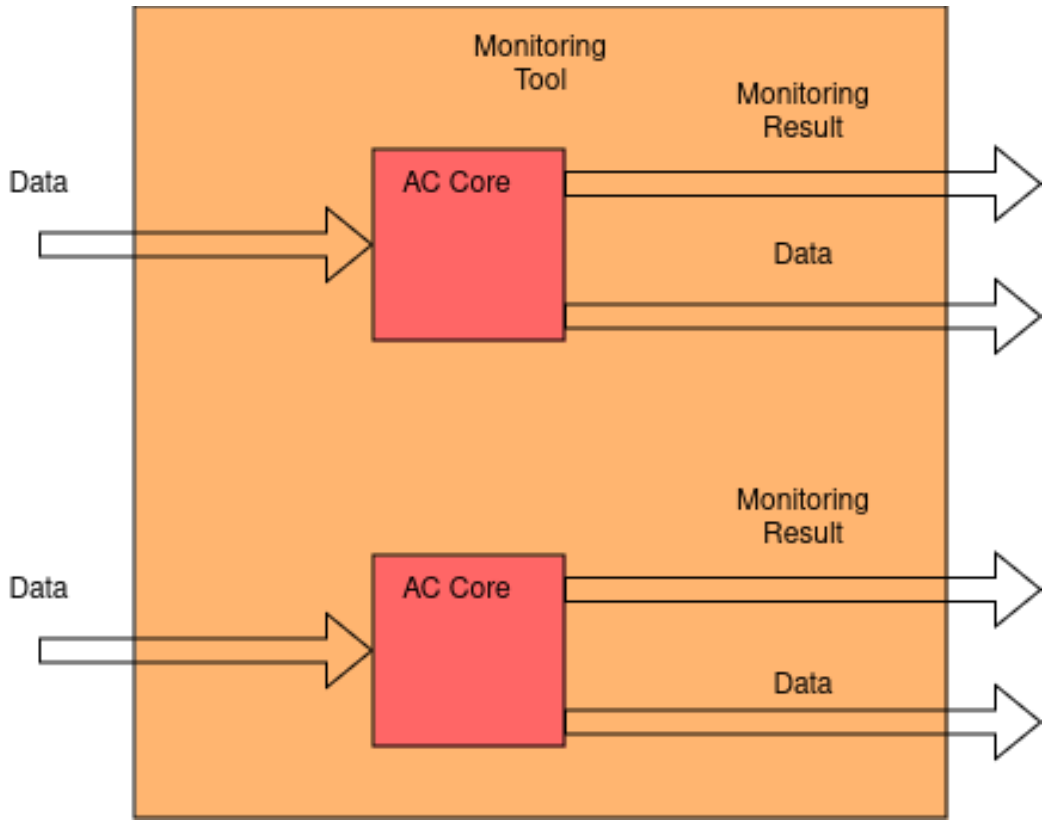


FIGURE 5.10: Hardware Architecture of Monitoring Tool

**Aho-Corasick IP core**

The architecture of the Aho-Corasick algorithm consists of a pipeline architecture as presented in figure 5.11. The GPU instructions are streamed as input by the host CPU through the PCIe channels to the Aho-Corascik IP cores. In addition, the malicious patterns of GPU instructions are being loaded to the Aho-Corasick module in local memory as the rules in which the Aho-Corasick algorithm operates. The implemented architecture of each of the Aho-Corasick IP cores consists of two high-level pipeline stages, which are internally split into two lower-level pipeline stages. The processing results from the second pipeline stage are streamed back to the host CPU through the PCIe channels. The pipeline stages implement a low-level finite state machine (FSM), responsible for moving into the stored tree structure according to the streaming input data. The movement on the restored tree structure produces a final score, which indicates how many patterns of instructions are malicious.

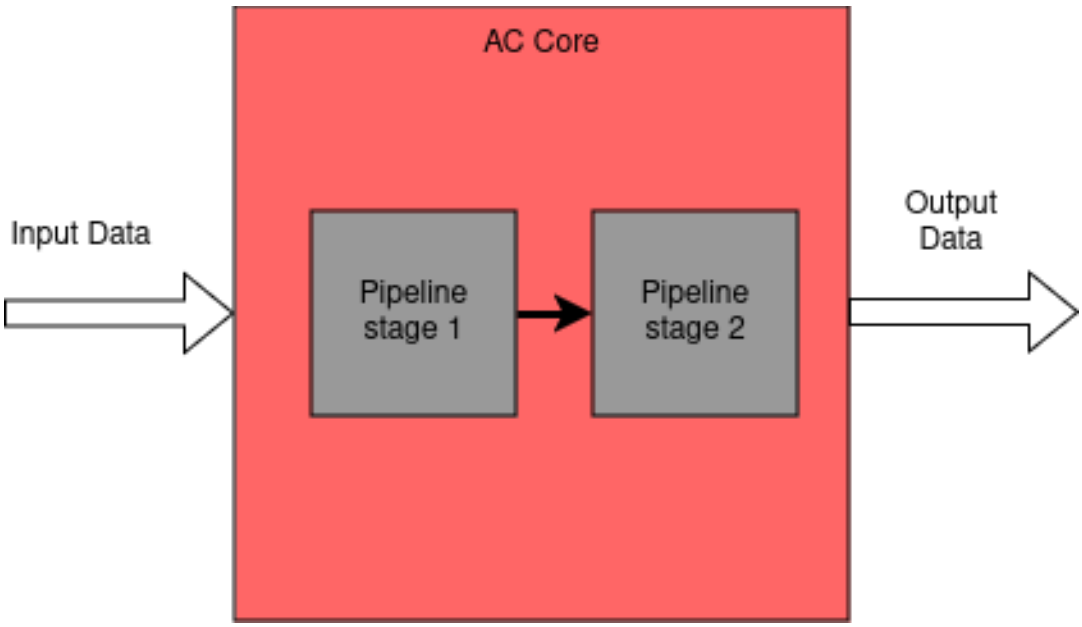


FIGURE 5.11: Hardware Architecture of Aho-Corasick IP core

### 5.2.2 Final design

The overall design of the PCIe endpoint consists of an XDMA IP core, an Aho-Corasick (AC) IP core, an AXI Interconnect, an AXI BRAM controller, a Block Memory Generator, and an AXI DMA Stream.

Figure 5.12 illustrates the hardware design of the PCIe endpoint. There are three main components: the PCIe interface, the PCIe monitor, and the GPU emulation. It can be observed that the flow of PCIe traffic is from the PCIe Interface to the PCIe monitor and then to GPU Emulation.

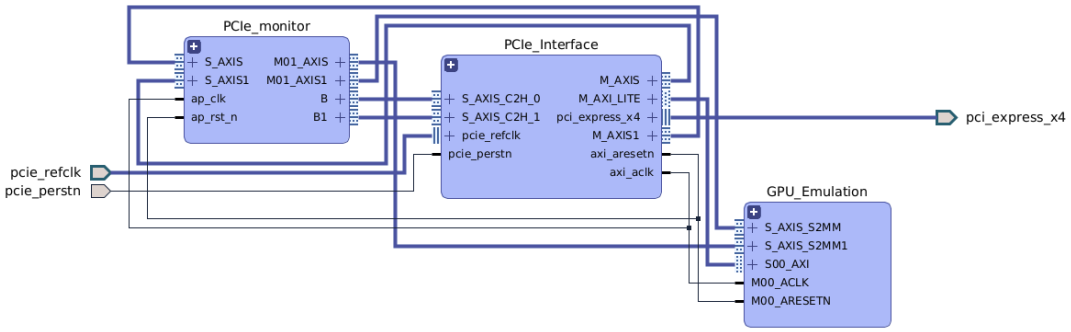


FIGURE 5.12: PCIe Monitoring System

The PCIe interface contains the XDMA IP core responsible for communication through PCIe, the AXI stream fifo IP cores, and the utility buffer used

for the PCIe reference clock. Figure 5.13 illustrates the PCIe interface block's sub-blocks and the interconnection between the IP cores.

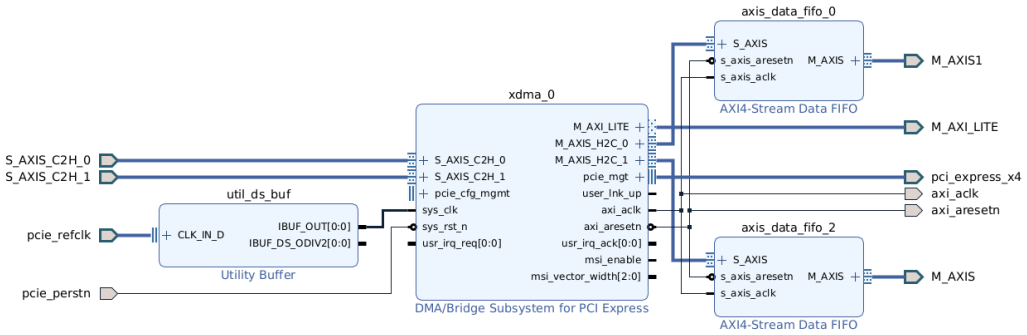


FIGURE 5.13: PCIe Interface Block Design

The PCIe monitor block comprises two Aho-Corasick IP cores to monitor the PCIe traffic and two AXI stream Broadcaster IP cores to broadcast the PCIe traffic to the GPU emulation concurrently with the monitoring. Figure 5.14 shows the PCIe monitor block's sub-blocks and the interconnection between the IP cores.

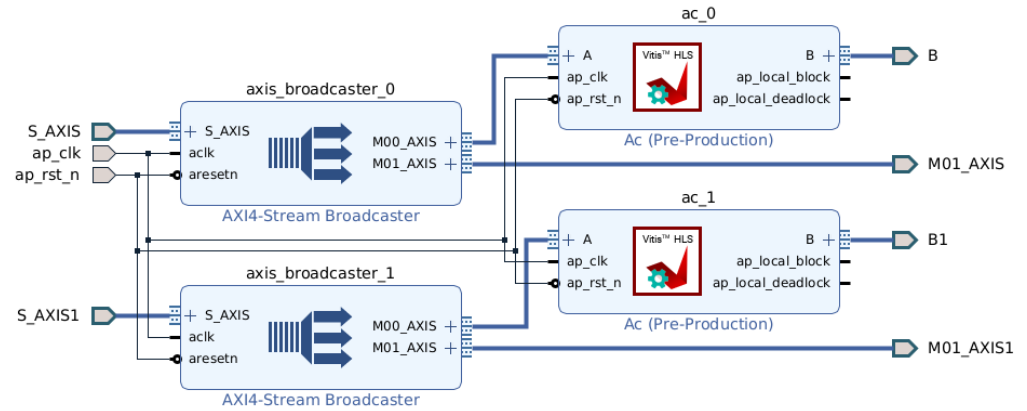


FIGURE 5.14: PCIe Monitor Block Design

Last but not least, the GPU emulation contains one BRAM to store the GPU executable code as it would happen to an actual GPU, two AXI stream DMA to convert the AXI stream to AXI memory mapped, two AXI BRAM Controller to send the GPU executable code from the DMA to the BRAM and an

AXI interconnect for the interconnection of the two AXI DMA with the PCIe interface to set up some configurations of the DMA. Figure 5.15 illustrates the GPU emulation block's sub-blocks and the interconnection between the IP cores.

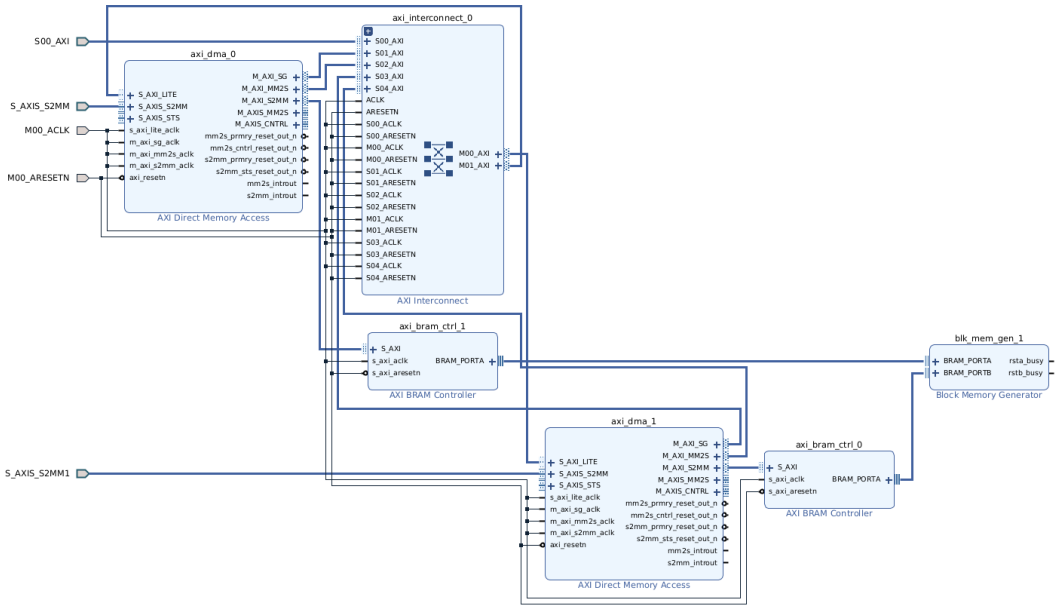


FIGURE 5.15: GPU emulation Block Design

## 5.3 Malicious GPU Patterns

The discovery of the GPU instructions, which may be insecure due to their malicious intentions, is being made to protect GPU from DOS (Denial of Service) attacks such as executing GPU Miners. It is widespread to deploy a GPU miner into a system to overload GPU usage and, as a result, to break the heterogeneous system.

The examined miners are the xmrMiner, yadaGPUminer, a simple GPU miner, and the Blake's2 miner. They were compared with a variety of regular programs, from sorting algorithms to AI and genetic algorithms, to ensure that there is diversity in the samples and that the patterns discovered in miners aren't random.

The GPU programs have been compiled using NVIDIA's compiler nvcc version 10.4 for GPUs supporting the architecture SM30 (Keppler Architecture) because it is a widespread architecture in NVIDIA's GPUs.

Initially, both regular programs and miners are analyzed at the instruction level to observe if any group of instructions is more commonly used in miners and extract any valuable statistical pieces of information. Developing a simple Python and bash script and using NVIDIA's cuobjdump tool to extract the GPU assembly commands of each GPU program revealed some interesting statistics.

Figure 5.16 shows the number of each instruction in the four GPU-examined miners, and it can be observed that the number of some specific instructions is greater than the rest of them. Specifically, the instructions SHF and LOP appear in a percentage around 20% of the total instructions of the program and 40% accordingly in the miners. Figures 5.17,5.18 illustrate the percentage of these assembly instructions for the xmrMiner, yadaGPUminer, and in absolute values.

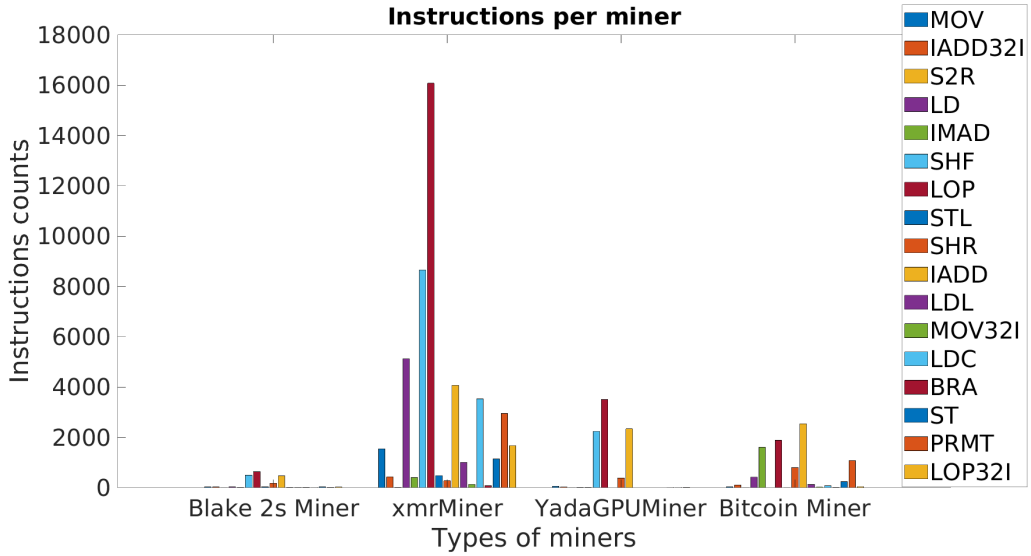


FIGURE 5.16: Instructions per Miner

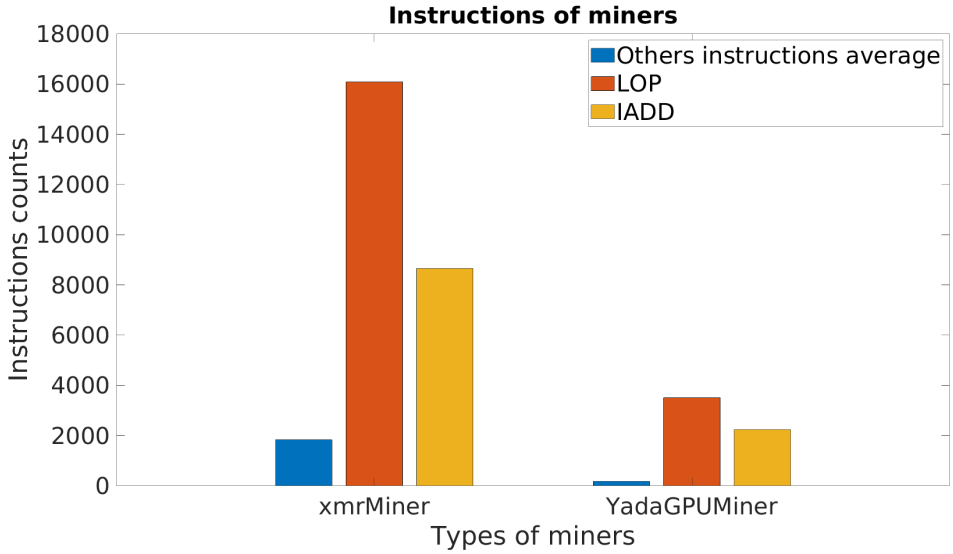


FIGURE 5.17: Instructions of Miners

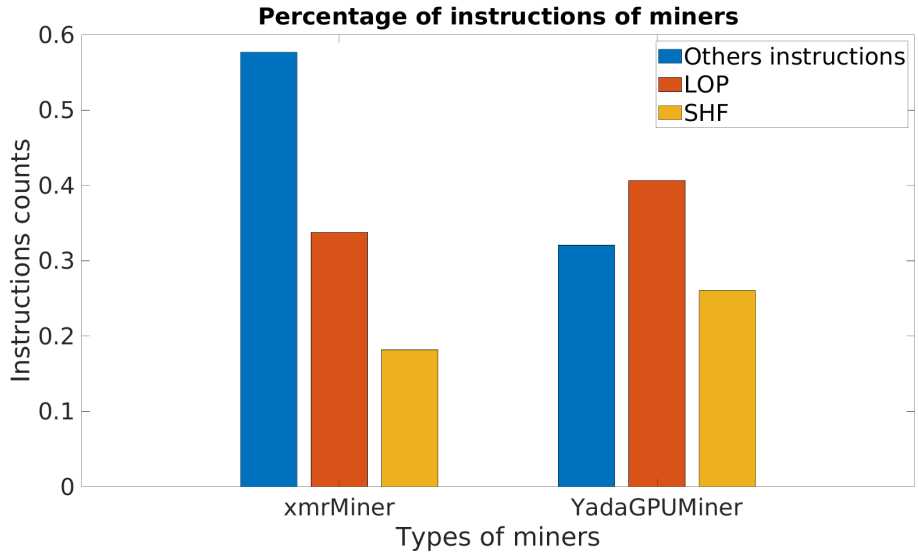


FIGURE 5.18: Instructions of Miners (%)

Furthermore, the same analysis of GPU instructions was performed on simple GPU programs. Figure 5.19 illustrates the number of instructions for a variety of simple GPU programs, figures 5.20, 5.21 shows the percentage of the exact GPU instructions SHF and LOP, which were found to appear at a higher rate in the GPU miners. It can be observed that the percentage of the LOP instruction occurs in a percentage less than 10% and the SHF instruction less than 5%. As a result, it can be considered that the possibility of the

appearance of the SHF and LOP instructions in a GPU miner is higher than that of a simple GPU program and approximately around 4x times more possible. This led to examining the GPU miner code of instructions for patterns of instructions containing the LOP and SHF.

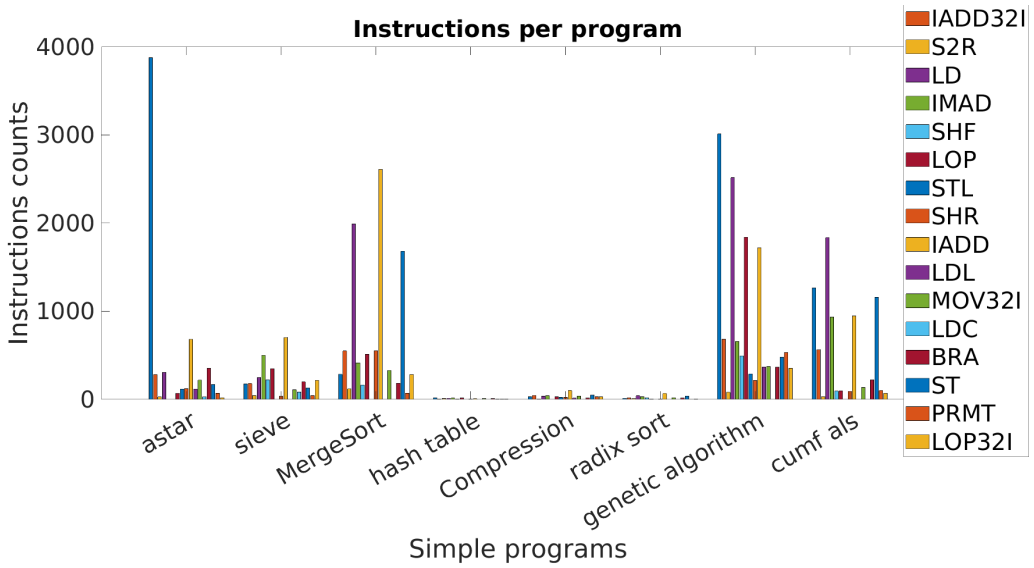


FIGURE 5.19: Instructions per program

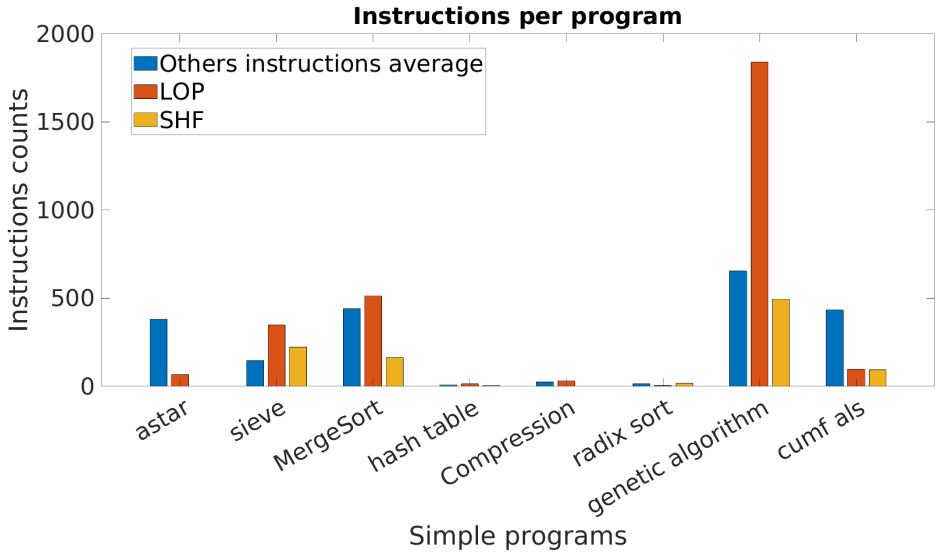


FIGURE 5.20: Instructions of programs

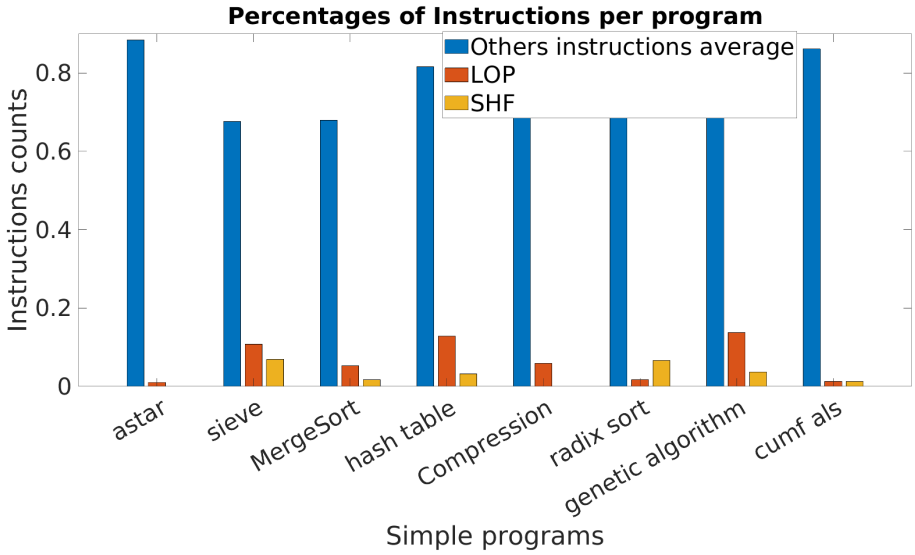


FIGURE 5.21: Instructions of programs (%)

Examining and extracting malicious patterns of GPU instructions that declare the use of GPU miners was achieved using the well-known framework SMPF with some modifications. It mined frequent closed sequential patterns using the BIDE+ algorithm. Due to the nature of mining algorithms examining for patterns, all the GPU instructions of a GPU miner executable program would require a tremendous amount of time, so a slightly different methodology was followed. In addition, it is observed in GPU code that every several instructions, a control code is sent after the instructions, which depends on GPU architecture. For instance, in the Kepler architecture of a GPU, the CPU sends a control code for every seven GPU instructions. As a result, the patterns of GPU instructions mined within the framework were between every seven GPU instructions.

The table 5.1 denotes with True value if the GPU miner has the corresponding pattern of instructions in its code and with false if not(True negative). The tables 5.2, 5.3 illustrate which pattern may be in simple GPU programs and be considered as false positives.

TABLE 5.1: The Malicious GPU Patterns of Instructions in Miners .

| Malicious GPU Patterns | miner1 | miner2 | miner3 | miner4 |
|---|---|---|---|---|
| SHF SHF LOP | True | True | True | False |
| SHF LOP SHF | True | True | True | False |
| SHF LOP LOP | True | True | True | False |
| LOP SHF SHF | True | True | True | False |
| LOP SHF LOP | True | True | True | False |
| LOP LOP SHF | True | True | True | False |
| LOP LOP LOP | True | True | True | True |
| LOP LOP LOP LOP | True | True | True | True |

miner1 : xmrMiner miner2 : yadaGPUMiner miner3 : Simple GPU miner miner4 : Blake's2 GPU Miner

TABLE 5.2: The Malicious GPU Patterns of Instructions in Simple Programs.

| Malicious GPU Patterns | Astar | Compression | MergeSort | Sieve |
|---|---|---|---|---|
| SHF SHF LOP | False | False | False | False |
| SHF LOP SHF | False | False | False | True |
| SHF LOP LOP | False | False | False | True |
| LOP SHF SHF | False | False | False | False |
| LOP SHF LOP | False | False | False | True |
| LOP LOP SHF | False | False | False | True |
| LOP LOP LOP | False | False | True | True |
| LOP LOP LOP LOP | False | False | False | False |

TABLE 5.3: The Malicious GPU Patterns of Instructions in Simple Programs.

| Malicious GPU Patterns | cumf als | genetic | hash table | Radix sort |
|---|---|---|---|---|
| SHF SHF LOP | False | False | False | False |
| SHF LOP SHF | False | False | False | False |
| SHF LOP LOP | False | False | False | False |
| LOP SHF SHF | False | False | False | False |
| LOP SHF LOP | False | False | False | False |
| LOP LOP SHF | False | False | False | False |
| LOP LOP LOP | True | True | False | False |
| LOP LOP LOP LOP | False | True | False | False |

# Chapter 6

# Evaluation

The Evaluation of the PCIe monitoring system involved verifying the system's functional demands and calculating the performance metrics. The functional verification process involves cross-validating the simulation results with the FPGA board results. In contrast, the performance metrics in the FPGA board consist of calculating the overall design's power and energy consumption and the IP cores of the Aho-Corasick algorithm's computational throughput and latency.

## 6.1 Functional Verification

The process of verifying the functional demands of the PCIe monitoring system involves cross-validating many samples of inputs containing typical cases and edge cases with the results taken from the FPGA board using a System Integrated Logic Analyzer (System ILA) to check the most important signals of the IP core of Aho-Corasick.

The samples of inputs contain both cases in which there may not exist any malicious pattern of GPU code, or it may exist from 1 to 8 different malicious patterns. They have been extracted from real-life GPU codes using an NVIDIA tool named cuobjdump to extract GPU assembly code and its hex representation. The hex representation is being used to create a new file containing only the opcode of the GPU assembly commands.

### 6.1.1 Simulation Results

Given the inputs mentioned in the above section, the simulation results are viewed in a waveform in the simulation environment of Xilinx Vivado

Suite using the testbench produced by the co-simulation process of the Vitis HLS tool of Xilinx.

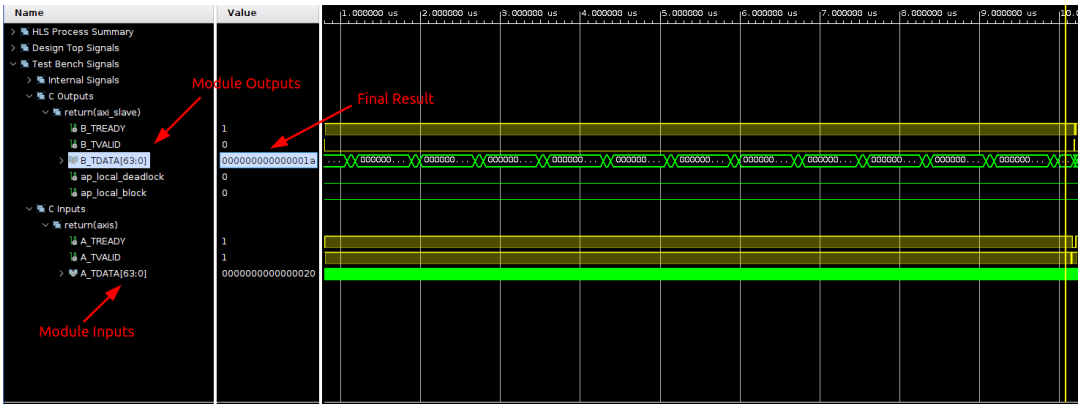


FIGURE 6.1: Simulation waveform of Aho-Corasick IP core

Figure 6.1 illustrates the results from a simulation of the Aho-Corasick IP core in a waveform in the Xilinx Vivado suite simulation environment.

### 6.1.2 FPGA Prototype Results

The FPGA board results are being extracted in the hardware manager environment of Xilinx Vivado Suite in a waveform using the system ILA's features given the same inputs as in the simulation.

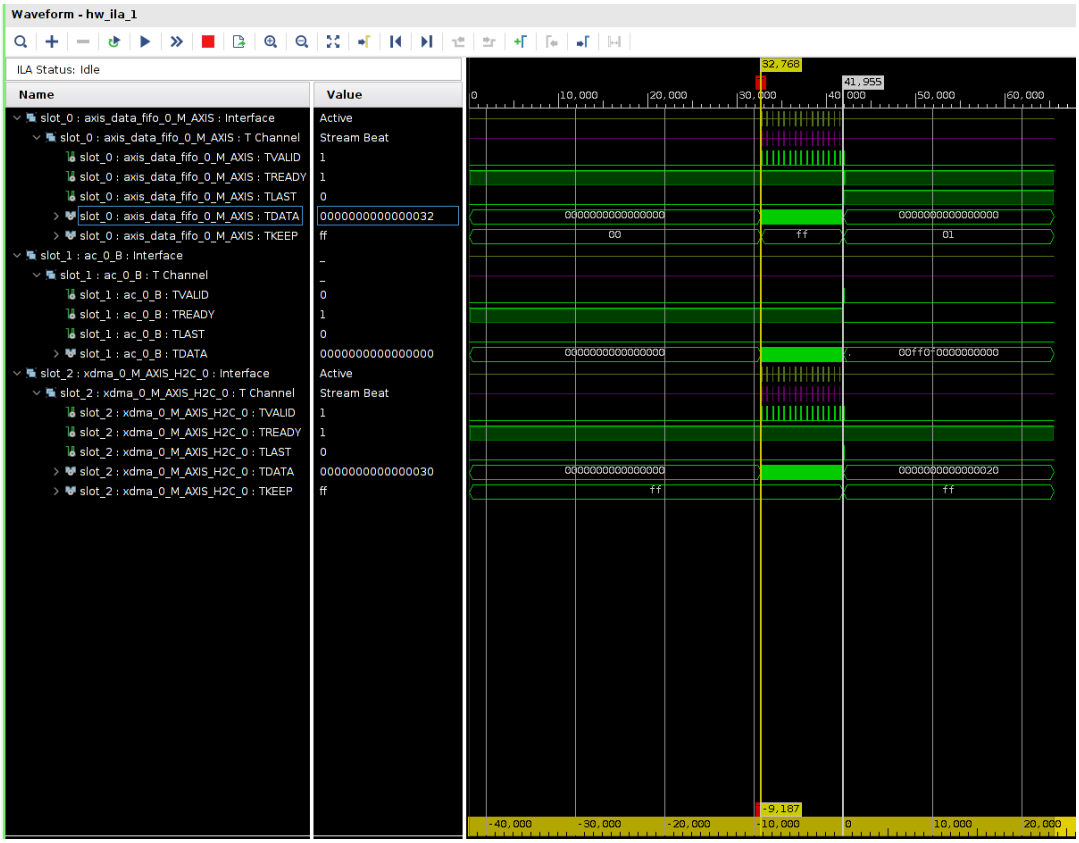


FIGURE 6.2: ILA waveform of Aho-Corasick IP core

Figure 6.2 illustrates the results of the Aho-Corasick IP core from the FPGA board in a waveform and in a CVS file to verify the correct functionality and cross-validate the results with the simulation results.

## 6.2 Performance Metrics

The performance evaluation of the PCIe monitoring system is being compared with an implementation of the Aho-Corasick algorithm in a CPU and, more specifically, with the SNORT program, which is being fully optimized, publicly available, and in its core functionality to find patterns based on a set of rules is using the algorithm of Aho-Corasick since it is considered state-of-the-art.

TABLE 6.1: FPGA Board KC705 Specifications

| FPGA Board KC705 | |
|---|---|
| Frequency (MHz) | 100 |
| PCIe Lanes | x8 |
| PCIe gen | 2x4 Gen 2 |
| Solid DDR | 1 GB |
| LUTs | 203800 |
| Flip Flops | 407600 |
| BRAMs | 445 |

TABLE 6.2: CPU's System Specifications

| CPU's System | |
|---|---|
| CPU | i5 13th Gen |
| Frequency (MHz) | 4300 |
| RAM (GB) | 32 |
| OS | Ubuntu 22.04 |
| SSD | 1 TB |
| MotherBoard | |

### 6.2.1 Specifications of Compared Platforms

The platforms where the algorithm of Aho-Corasick is being compared are an Intel CPU i5 of the 13th generation and the FPGA board of the Kintex 7-series family named kc705. The following tables 6.1, 6.2 present the specifications of each platform separately in great detail.

### 6.2.2 Power and Energy Consumption

Power and energy consumption are calculated using a Linux tool, perf [25], on the Intel CPU, which needs root privileges to run. The last leverages the feature of Intel CPUs to measure the energy consumption of an executable program in Joules by writing the measurements in a specific directory in Linux. The figures 6.3,6.4 below illustrate the use of the perf Linux tool to calculate the power and energy consumption of the Aho-Corasick algorithm, given diverse inputs of different sizes and the result of the tool.

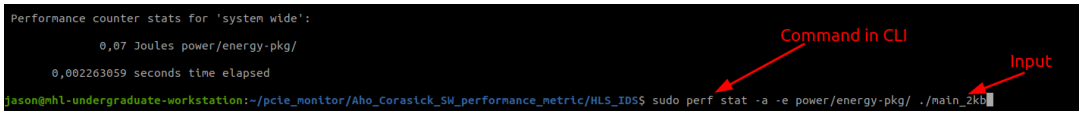


FIGURE 6.3: Use of the perf tool in Command Line


FIGURE 6.4: Results of the perf tool in the terminal

The power and energy consumption calculation in the FPGA board uses the power produced from Xilinx Vivado Suite by leveraging the type of power p = E / t measured in Watts, where E is the symbol of energy and t is the time needed. Figures 6.5,6.6 illustrate the power distribution in the FPGA fabric in 3 main categories: GTX, Hard IP, and Dynamic. It can be observed that the power consumption increases according to the rise in the number of IP cores of the AC from 2.267 W to 2.294 W. In addition, the power consumption increase is in the dynamic region and neither in the Hard IP nor the GTX region. It happens because the overall design is the same in the two cases, with the only difference being the number of AC IP cores.

Another significant observation is that the greater percentage of power consumption in the FPGA board happens in the GTX region because the GTX transceivers are used for the PCIe IP core.

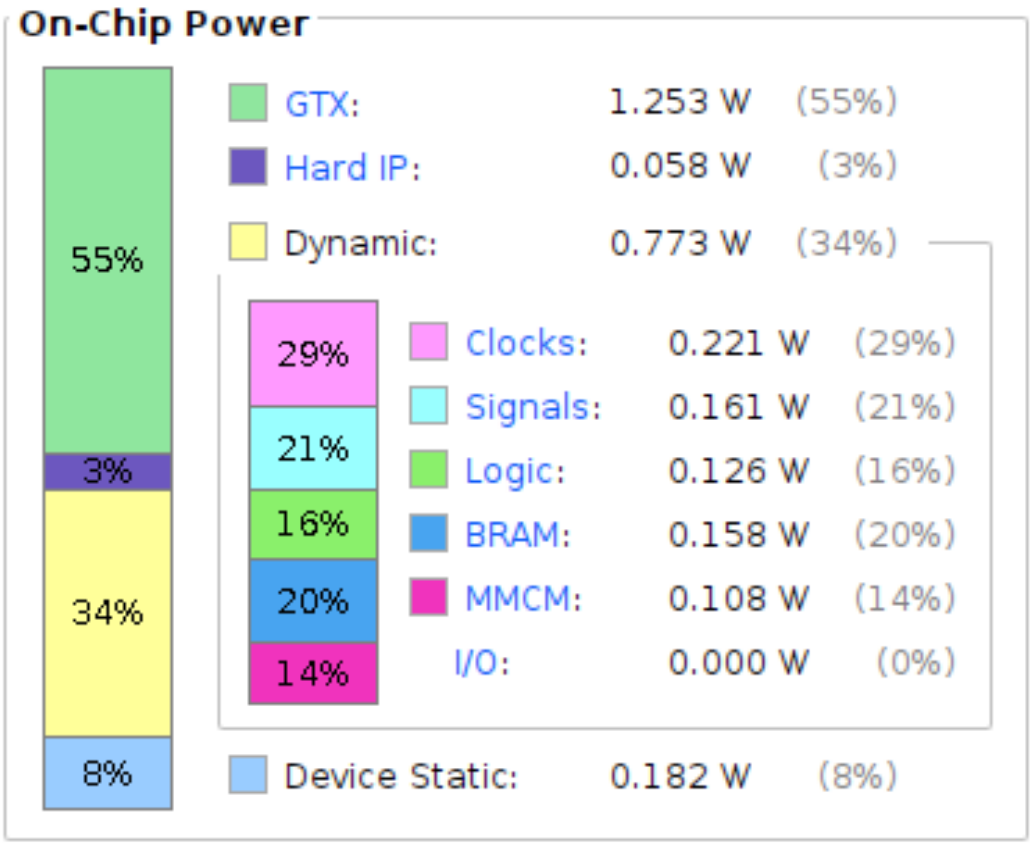


FIGURE 6.5: Power Consumption of PCIe monitoring system using 1 IP core of AC

| PCIe monitoring system | Overall | GTX | Hard IP | Dynamic |
|---|---|---|---|---|
| 1 AC IP core | 2.267 | 1.253 | 0.058 | 0.773 |
| 2 AC IP cores | 2.294 | 1.253 | 0.058 | 0.800 |

TABLE 6.3: Power Consumption of PCIe monitoring system (Watts)

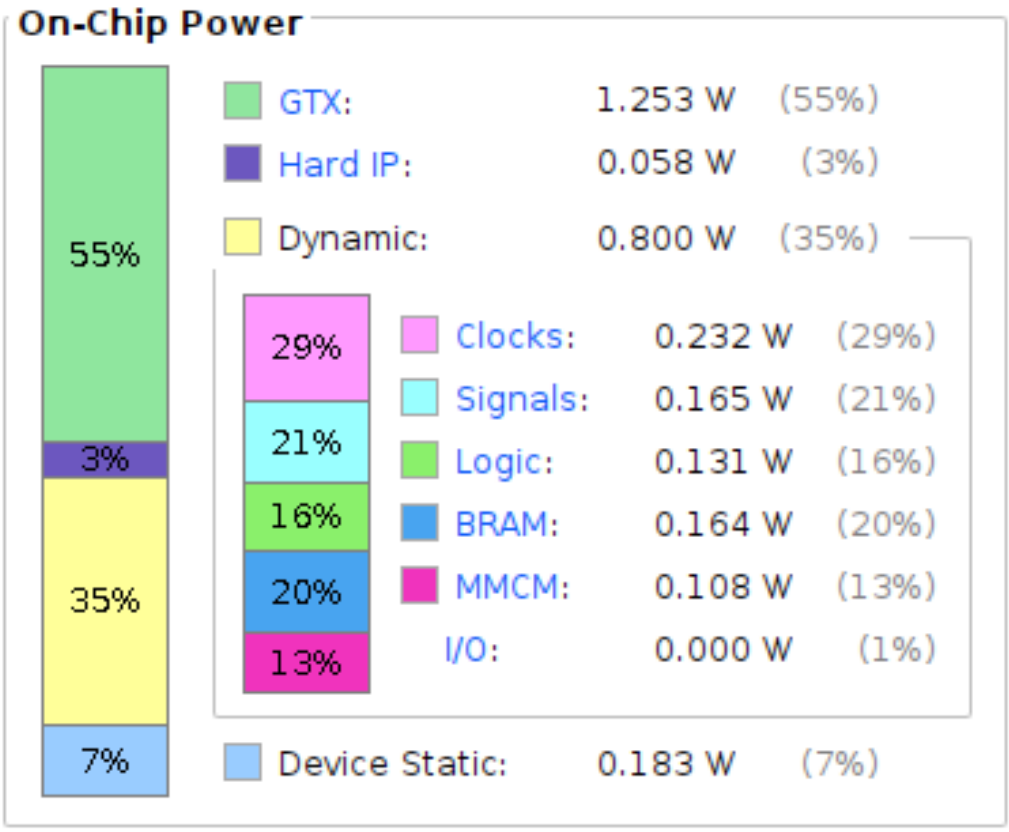


FIGURE 6.6: Power Consumption of PCIe monitoring system using 2 IP cores of AC

Lastly, a summary table 6.3 presents the numerical values of the PCIe monitoring system's power consumption for 1 and 2 AC IP cores.

Figure 6.7 illustrates the energy consumption of the algorithm of Aho-Corasick (AC) in the compared platforms mentioned in the above tables 6.1,6.2, in the FPGA board KC705 there are two distinct cases: one with 1 IP core of Aho-Corasick and another one with 2 IP cores of Aho-Corasick.

The energy consumption of the Aho-Corasick in the CPU is much more energy-consuming than that of implementing the same algorithm in the FPGA board.

This is happening because, in the system, the CPU functions need much more power than a single FPGA. Specifically, it requires approximately 30 Watts (W) to run the algorithm Aho-Corasick, while the FPGA needs from 2.267 W to 2.294 W, corresponding to 1 IP core of AC and 2 IP cores of AC. The increase in the power consumption in the FPGA board is rational because it needs 2x times more resources to implement two cores of AC than the implementation of 1 core of AC.

Consequently, the CPU needs 275x times more energy than an FPGA, which uses a single IP core of Aho-Corasick for the same input. On the other hand, when the FPGA uses 2 IP cores of Aho-Corasick, it needs 543x times less energy than the CPU. This obnoxious behavior may feel a little bit strange. However, it is utterly rational since using two IP cores of Aho-Corasick running in parallel can reduce the algorithm's computation time in half without considering the I/O.

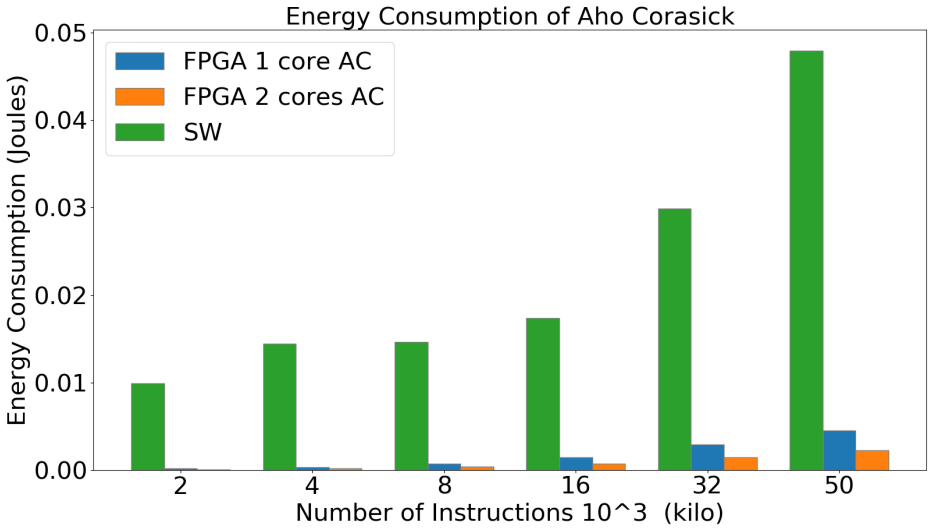


FIGURE 6.7: Energy Consumption Diagram

### 6.2.3 Resources Utilization

The resources of the PCIe monitoring system have been calculated through the Xilinx Vivado Suite for the two cases: one with 1 AC IP core and the other with 2 AC IP cores. The resource utilization of the two cases can be shown in figures 6.8,6.9 in total numbers and figures 6.10,6.11 as a percentage of the total resources of the FPGA board.

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 17108 | 203800 | 8.39 |
| LUTRAM | 1672 | 64000 | 2.61 |
| FF | 18495 | 407600 | 4.54 |
| BRAM | 33 | 445 | 7.42 |
| IO | 1 | 500 | 0.20 |
| GT | 4 | 16 | 25.00 |
| BUFG | 5 | 32 | 15.63 |
| MMCM | 1 | 10 | 10.00 |
| PCIe | 1 | 1 | 100.00 |

FIGURE 6.8: Resources Utilization of PCIe monitoring system using 1 AC IP core

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 18048 | 203800 | 8.86 |
| LUTRAM | 1741 | 64000 | 2.72 |
| FF | 19549 | 407600 | 4.80 |
| BRAM | 39.50 | 445 | 8.88 |
| IO | 1 | 500 | 0.20 |
| GT | 4 | 16 | 25.00 |
| BUFG | 5 | 32 | 15.63 |
| MMCM | 1 | 10 | 10.00 |
| PCIe | 1 | 1 | 100.00 |

FIGURE 6.9: Resources Utilization of PCIe monitoring system using 2 AC IP cores

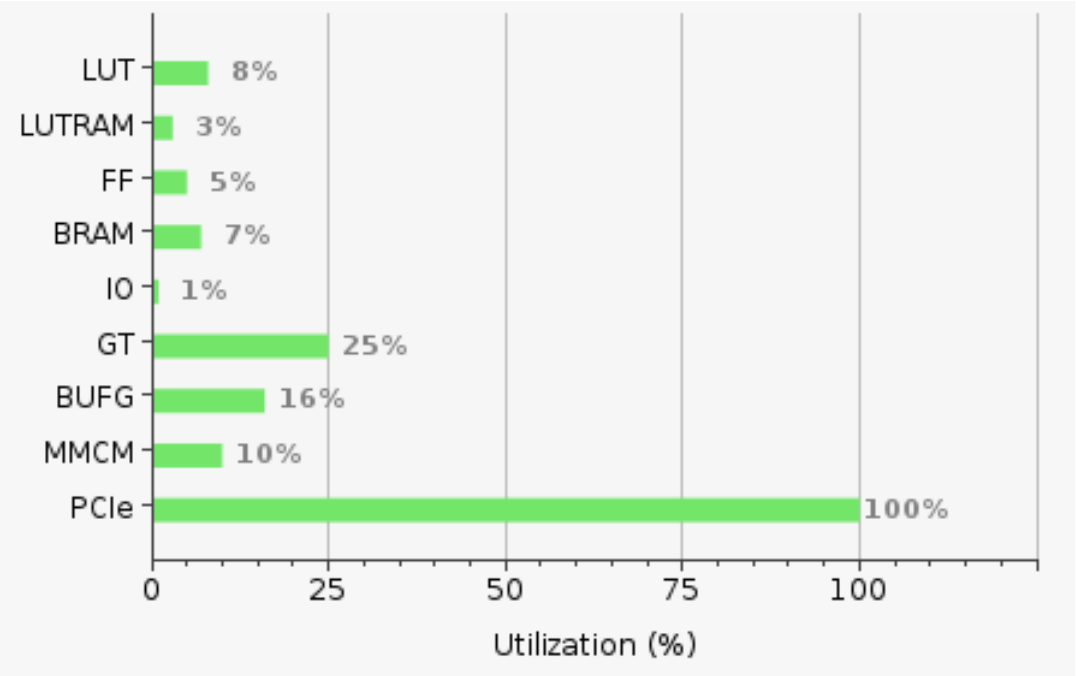


FIGURE 6.10: Resources Utilization of PCIe monitoring system using 1 AC IP core (%)

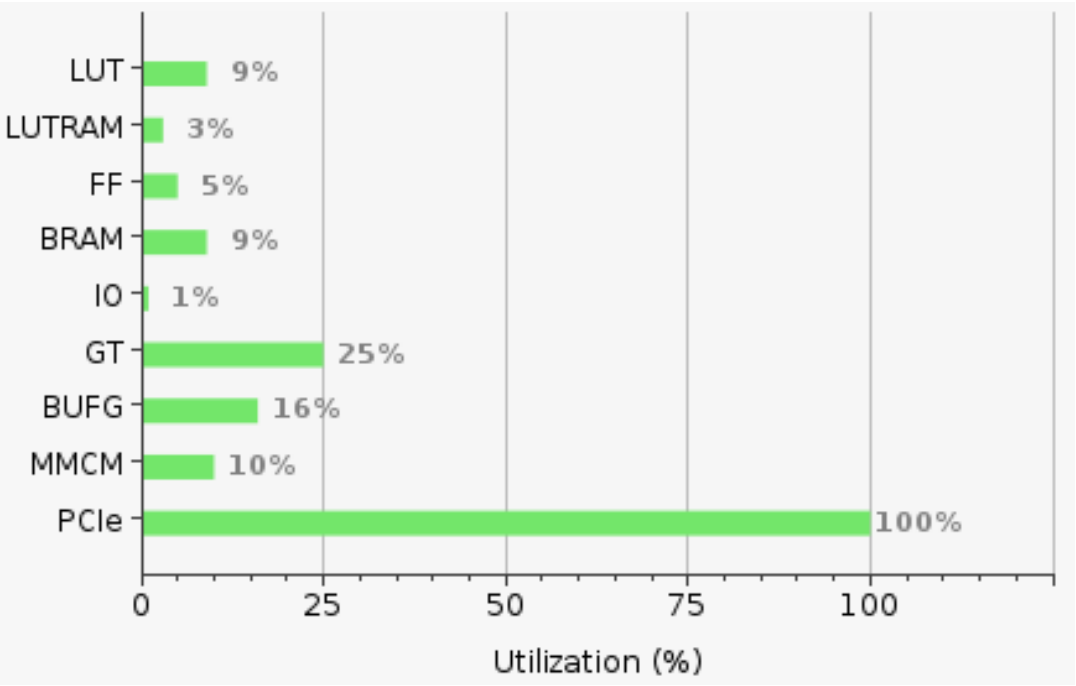


FIGURE 6.11: Resources Utilization of PCIe monitoring system using 2 AC IP cores (%)

Figures 6.12,6.13 show the distribution of the FPGA board resources in the fabric of the FPGA.

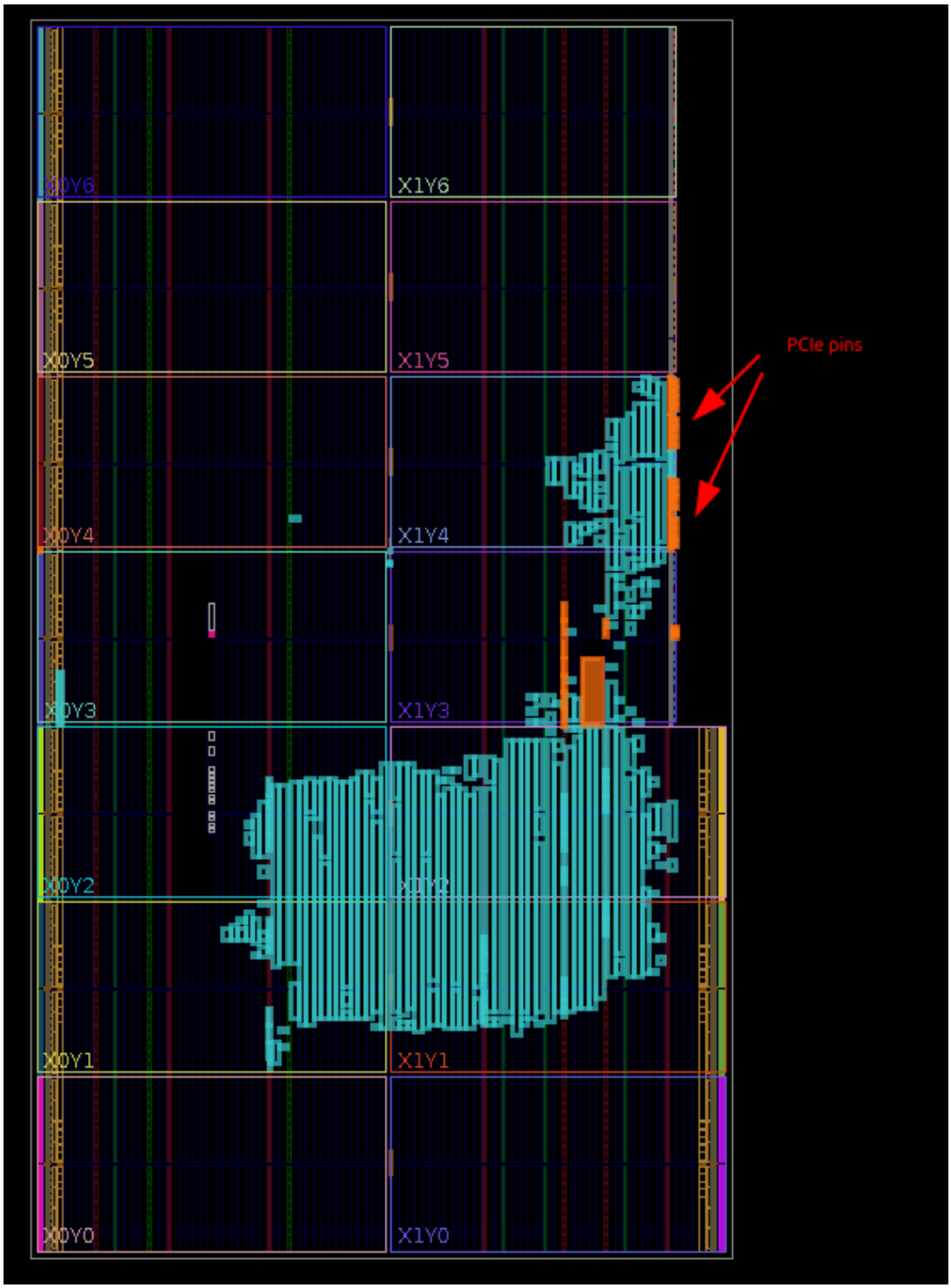


FIGURE 6.12: Resources Distribution of PCIe monitoring system using 1 AC IP core in the FPGA fabric

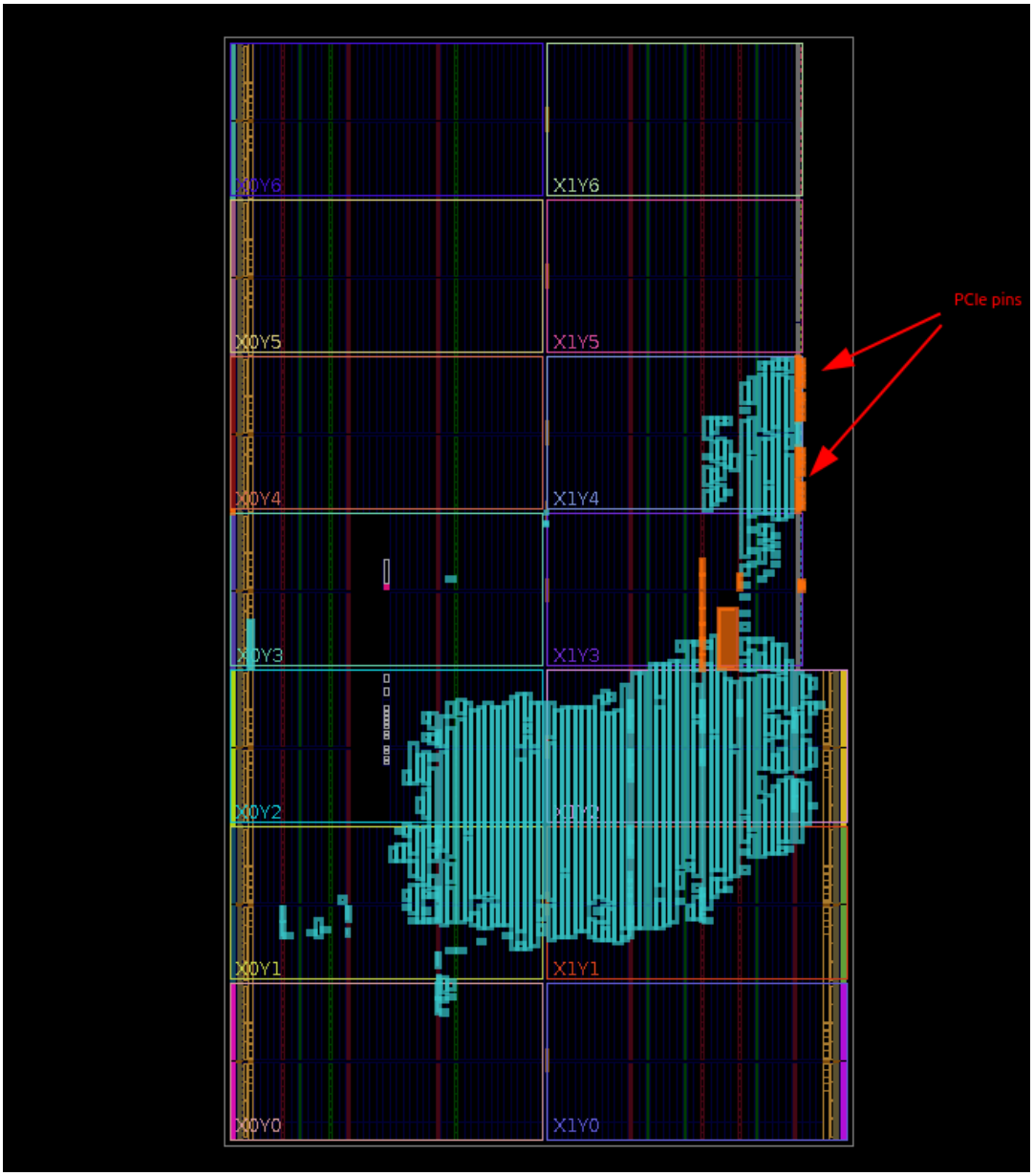


FIGURE 6.13: Resources Distribution of PCIe monitoring system using 2 AC IP cores in the FPGA fabric

Comparing the total resources needed for both implementations of the PCIe monitoring system using 1 AC IP core and 2 AC IP cores, it can be shown that using two cores of AC requires more resources than using one core, which is utterly rational and expected. Figure 6.14 illustrates this rise of resources of the FPGA board in the case of the 2 AC IP cores in contrast with the case of the 1 AC IP core. However, it seems that compared to the whole PCIe design, this rise is a very small fraction of the total design.

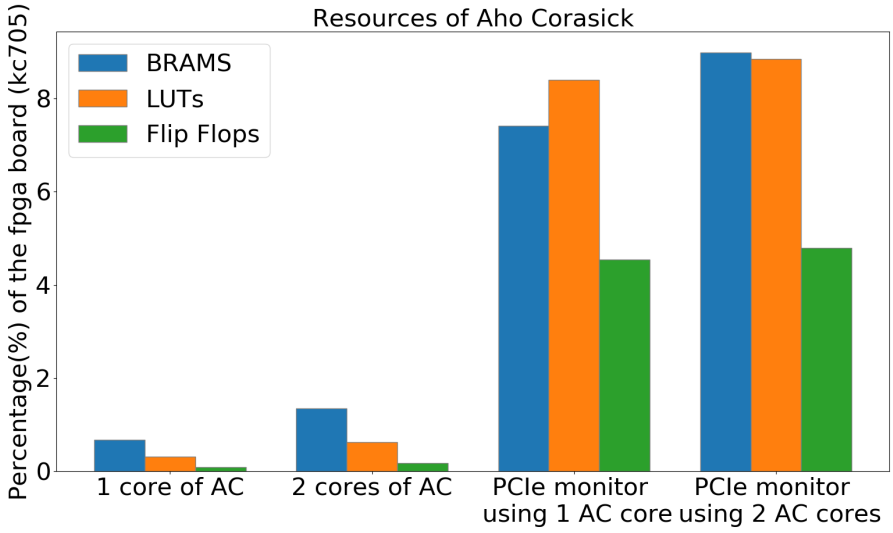


FIGURE 6.14: Resources Utilization of PCIe monitoring system

### 6.2.4 Throughput and Latency

The latency and processing throughput of the Aho-Corasick algorithm are calculated using the compared platforms, CPU, and FPGA board accordingly. However, due to the structure and nature of the AC algorithm, it is not parallelizable. For this reason, to leverage the potential of the FGPA board to parallelize tasks, two AC IP cores have been used, tested, and evaluated in the computational performance of the algorithm.

Figures 6.15, 6.16 illustrate the computational performance of the Aho-Corascik in the compared platforms in the metrics of latency and processing throughput. The performance of the Aho-Corasick algorithm in the FPGA board using two AC IP cores is faster and has greater processing throughput than the implementation in the CPU and in the FPGA of 1 AC IP Core. Furthermore, it can be observed from the plots of latency and processing throughput that both of these metrics are better in the CPU than in the FPGA when using 1 AC IP core. This happens because the algorithm's structure is not parallelizable since the process of the element has a memory dependency on the previous element. As a result, to run in parallel, the Aho-Corasick algorithm must run on multiple AC IP cores by separating the input into smaller inputs, each corresponding to one AC IP core.

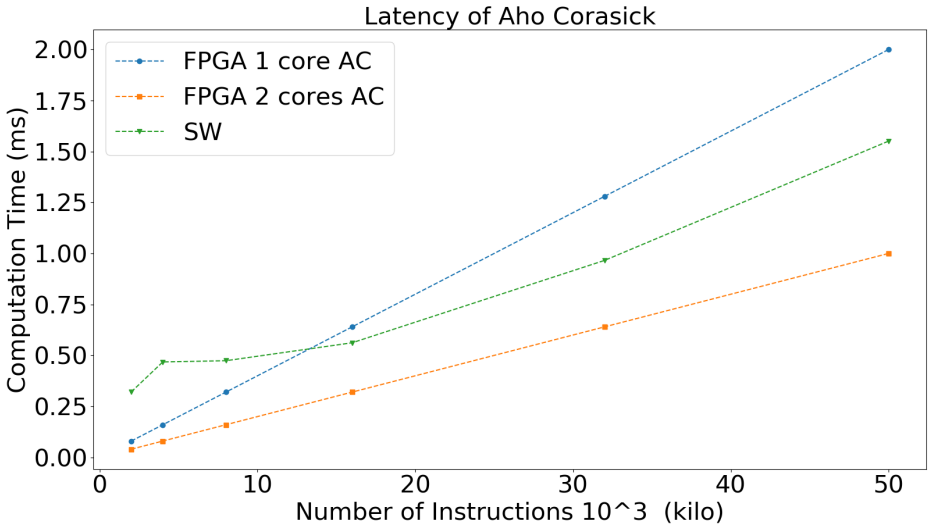


FIGURE 6.15: Latency of Aho-Corasick in the Compared Platforms

Another significant observation is that the processing throughput of AC in the FPGA Board using two AC IP cores is greater than the throughput of the CPU, which is rational since the latency was accordingly greater. According to their latencies, the processing throughput of the FPGA board using one AC IP core is worse than that of the FPGA board with two AC IP cores and the CPU.

The latency speedup of FPGA using two AC IP cores compared to the CPU implementation of the Aho-Corasick Algorithm is 1.55x, and the speedup of the according processing throughput is 1.55x compared to the CPU.

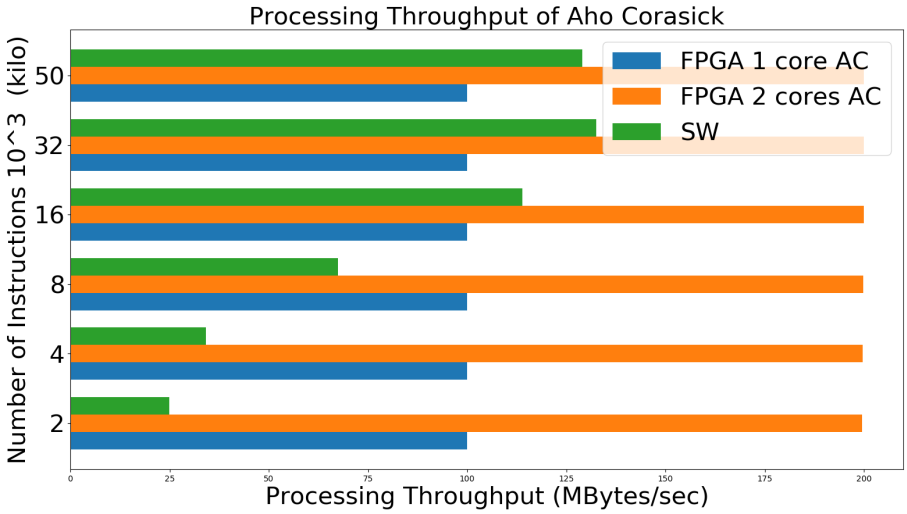


FIGURE 6.16: Processing Throughput of Aho-Corasick in the Compared Platforms

## 6.3 Evaluation Summary

Overall, the final performance of the algorithm Aho-Corasick used to monitor the PCIe traffic is better than the corresponding implementation of the algorithm in CPU. The final speedup of latency in the specific FPGA board is 1.55x compared to the CPU, and the speedup of the processing throughput is approximately 1.55x compared to the CPU. Furthermore, the implementation in the FPGA board using two AC IP cores is energy efficient compared to the CPU by a factor of 543x times less energy consumed than the CPU. Last but not least, the Aho-Corasick IP core uses a minimum amount of resources, making it a great choice for monitoring, and it can be leveraged by using multiple AC IP cores to parallel the process and make it faster.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

With many companies such as Amazon Web Services (AWS) or Microsoft Azure investing in building vast data centers consisting of hundreds of thousands of diverse accelerators, including FPGAs, TPUs, and GPUs, many developers and researchers rely on using these accelerators of the Data Centers instead of buying these costly products to execute heavy load computational tasks. The problem that emerges is that when using these accelerators, the providers need to ensure the secure code execution of the customer's applications, the confidentiality and integrity of the user's data, and the availability of the accelerators. Many providers restrict the use of accelerators with standard specifications that need to be followed, or software tools and statical analyzers to check for possible malware use instead of executing normal applications. These defense mechanisms consume a lot of the resources of the data centers, such as energy, power, and time, which are of critical importance. This thesis work could provide a different defense mechanism, a real-time monitoring system at a low level for detecting such malicious actions with minimum resources and great performance metrics. Finally, it could offer a novel perspective in detecting malicious executable codes in the accelerators used in various heterogeneous systems. All in all, it is a system that can contribute to the community in various ways.

## 7.2 Future Work

This thesis implementation is a proof of concept. Therefore, many improvements and applications can be made due to the nature of the system

setup and the vast rise of heterogeneous systems. First of all, the GPU emulation, instead of being a memory in which the GPU binary code is only stored, could be replaced by a softcore GPU, which is a common solution for many GPU applications since it is energy efficient compared with typical GPUs and has approximately the same time of execution for the majority of applications. Furthermore, providing a larger FPGA with more streaming channels and monitoring systems can increase the number of Aho-Corasick IP cores running concurrently. As a result, it can lead to even greater performance. In addition to this, more streaming channels could provide the capability to partially reconfigure the state table of Aho-Corasick IP cores use and, therefore, the set of rules, giving the administrator of the monitoring system the ability to modify the rules in real-time. Moreover, due to the minimum resources the monitoring system needs to operate correctly, it could be ported into a real heterogeneous system composed of a CPU and a GPU by adding between them a PCIe interposer with a PL part and integrating the monitoring system into the PCIe interposer providing it with the ability to monitor the PCIe traffic between CPU and GPU. Last, but not least, it could be a further analysis of malicious GPU applications, except in the case of the miners, and how they could be detected at the instruction level using this real-time monitoring system.

# References


[4] Ari B. Hayes et al. “Decoding CUDA binary”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019. DOI: https://doi.org/10.5281/zenodo.2339027.

[5] Ari B. Hayes et al. “Decoding CUDA binary - file format”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Feb. 2019. DOI: https://doi.org/10.5281/zenodo.2339027.

[6] Ari B. Hayes et al. “Decoding CUDA binary - decoded instructions”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Feb. 2019. DOI: 10.5281/zenodo.2339116.

[7] Ari B. Hayes et al. “Decoding CUDA binary - opcodes”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Feb. 2019. DOI: 10.5281/zenodo.2339154.

[8] Ari B. Hayes et al. “Decoding CUDA binary - special registers”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Feb. 2019. DOI: 10.5281/zenodo.2339176.

[9] Andrea Miele. “Buffer overflow vulnerabilities in CUDA: a preliminary analysis”. In: *2015 21$^{th}$ Journal of Computer Virology and Hacking Techniques, 2015)*. IEEE, 2015. DOI: 10.1109/fpl.2018.00031.

[10] Bang Di, Jianhua Sun, and Hao Chen. “A Study of Overflow Vulnerabilities on GPUs”. In: *Network and Parallel Computing*. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-47099-3_9.

[11] Sang-Ok Park et al. “Mind control attack: Undermining deep learning with GPU memory exploitation”. In: *Computers  Security* 102 (2021), p. 102115. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2020.102115. URL: https://www.sciencedirect.com/science/article/pii/S0167404820303886.

[12] Christopher Erb, Mike Collins, and Joseph L. Greathouse. “Dynamic buffer overflow detection for GPGPUs”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 61–73. DOI: 10.1109/CGO.2017.7863729.

[13] Bang Di et al. “GMOD: a dynamic GPU memory overflow detector”. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. New York, NY, USA: Association for Computing Machinery, 2018. ISBN: 9781450359863. DOI: 10.1145/3243176.3243194. URL: https://doi.org/10.1145/3243176.3243194.

[14] Volos Stavros, Vaswani Kapil, and Bruno Rodrigo. “Graviton: trusted execution environments on GPUs”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI’18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 681–696. ISBN: 9781931971478. URL: https://dl.acm.org/doi/10.5555/3291168.3291219.

[15] Jang Insu et al. “Heterogeneous Isolated Execution for Commodity GPUs”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 455–468. ISBN: 9781450362405. DOI: 10.1145/3297858.3304021. URL: https://doi.org/10.1145/3297858.3304021.

# Misc


[1] Microsoft. *Cryptojacking: Understanding and defending against cloud compute resource abuse*. https://www.microsoft.com/en-us/security/blog/2023/07/25/cryptojacking-understanding-and-defending-against-cloud-compute-resource-abuse/. 2023.

[2] Cisco. *Cybercriminals target graphic designers with GPU miners*. https://blog.talosintelligence.com/cybercriminals-target-graphic-designers-with-gpu-miners/. 2023.

[3] Xilinx. *XDMA Driver*. https://github.com/Xilinx/dma_ip_drivers/. 2020.

[16] *Xilinx ZYNQ UltraScale+ MPSoC Family*. https://developer.ridgerun.com/wiki/index.php/Xilinx_ZYNQ_UltraScale+_MPSoC/Introduction/Family. 2023.

[17] Xilinx. *ZCU102 Evaluation Board User Guide*. https://docs.amd.com/v/u/en-US/ug1182-zcu102-eval-bd. 2023.

[18] Xilinx. *Kintex 7 FPGA series*. https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/kintex-7.html. 2023.

[19] Xilinx. *KC705 Evaluation Board User Guide*. https://docs.amd.com/v/u/en-US/ug810_KC705_Eval_Bd. 2023.

[20] Xilinx. *Vivado Design Suite User Guide*. https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_1/ug892-vivado-design-flows-overview.pdf. 2023.

[21] Xilinx. *Vitis High-Level Synthesis User Guide*. https://docs.amd.com/r/2023.2-English/ug1399-vitis-hls/Introduction. 2023.

[22] Xilinx. *PetaLinux Tools Documentation: Reference Guide*. https://docs.amd.com/r/en-US/ug1144-petalinux-tools-reference-guide/Prerequisites?tocId=LmPHRX6fcGHQao4bt01erw. 2023.

[23] *Vitis High-Level Synthesis work flow*. https://www.techsource-asia.com/our_courses/high-level-synthesis-with-the-vitis-hls-tool/.

[24] *Petalinux work flow*. https://support.xilinx.com/s/article/1066813?language=en_US.

[25] Milica Dancuk. *Linux perf: How to Use the Command and Profiler*. https://phoenixnap.com/kb/linux-perf. 2022.