UNIVERSITÀ DEGLI STUDI DI TRIESTE

XXXVII CICLO DEL DOTTORATO DI RICERCA IN

APPLIED DATA SCIENCE AND ARTIFICIAL INTELLIGENCE

Risultati conseguiti con il finanziamento ottenuto a valere sull'Asse IV del PON Ricerca e Innovazione 2014-2020 "Istruzione e ricerca per il recupero – REACT-EU"
Results achieved with funding obtained under Axis IV of NOP Research and Innovation 2014-2020 "Education and research for recovery – REACT-EU"

SUSTAINABLE COMPUTING IN THE AI ERA: ENERGY PROFILING AND RECONFIGURABLE HIGH-PERFORMANCE COMPUTING

Settore Scientifico Disciplinare: ING-INF/01

DOTTORANDO / A:
LUIS GERARDO LEON VEGA

COORDINATORE:
PROF. FRANCESCO PAULI

SUPERVISORE DI TESI:
PROF. STEFANO COZZINI

ANNO ACCADEMICO 2023/2024

Abstract

The surge of Artificial Intelligence (AI) and Deep Learning (DL) workloads has transformed high-performance computing (HPC), increasing demands for both computational power and energy efficiency. This dissertation addresses two key challenges in sustainable computing: energy accounting in next-generation supercomputers and the design of energy-efficient AI accelerators based on Field Programmable Gate Arrays (FPGAs).

First, a comprehensive methodology for fine-grained process-level energy consumption estimation is proposed. EfiMon is introduced to monitor system-wide and process-specific energy metrics without requiring execution isolation, enabling accurate energy profiling on CPUs and GPUs. New analytical models are developed, demonstrating sub-2% relative error for CPU-based measurements and under 10% error for GPU-based measurements, providing valuable insights into energy usage in shared-resource environments.

Second, this work presents the design and evaluation of a Flexible Accelerator Library (FAL), which enables the automatic generation of parameterised FPGA-based AI accelerators. This library supports customising operand size, numerical precision, approximate arithmetic injection, and accelerator reuse. Experimental validation explores standard, Strassen, and Winograd matrix multiplication approaches, assessing trade-offs among resource consumption, performance, and error resilience. Furthermore, approximate computing techniques are incorporated to reduce FPGA resource usage with minimal impact on model accuracy. For MobileNet v2, the resource reduction was approximately 20%, accompanied by an accuracy improvement of 16.6% due to healthy numerical disturbances in the softmax layer. For LeNet 5, it was 18.93% and 9.6% respectively.

The thesis extends its impact by exploring FPGA acceleration for Large Language Models (LLMs), proposing architectures optimised for LLM inference at the edge, and discussing pathways for future AI computing architectures that prioritise energy efficiency, scalability, and reconfigurability. This research achieves a speedup ranging from 1.37× to 10.98× over two AMD EPYC 7H12 CPUs with 64 cores each, outperformed by an NVIDIA Tesla V100 by a factor of 1.66×.

Lastly, this work highlights the use of heterogeneous systems equipped with CPUs, GPUs, ASICs, and reconfigurable devices to mitigate high-arithmetic-intensity tasks

(compute-bound) and the integration of compute units into memory modules to address low-arithmetic-intensity workloads (memory-bound), which can evolve and adapt to the rapidly changing requirements of AI.

Keywords: approximate computing, machine learning, neural networks, hardware acceleration, inference, field programmable gate arrays, sustainable computing, coarse grained reconfigurable arrays, reconfigurable computing, cooperative computing, heterogeneous computing.

To my beloved family, including Kyle

Cara madre,
D'accordo con uno dei tuoi desideri, ho imparato un poco d'Italiano. Probabilmente, scriverò con molti errori questo testo. Devo dire che mi ha piaciuto farlo. Anche questa tesi significa la fine della mia educazione, in cui sono andato in molti luoghi nuovi. Ti ricorderò con molto amore per sempre e continuerò camminando per molti anni più, più lontano, come tu avevi voluto.
Tuo figlio, Gerardo

Acknowledgements

First, I would like to express my sincere gratitude to Dr. Stefano Cozzini for informing me about the opportunities to return to Trieste, entrusting the research to me, and trusting in my potential, which allowed me to complete my academic studies with a PhD. Stefano gave me the freedom to choose any research career path, allowing me to materialise my ideas while providing enough feedback to help me pursue my goals. For that, I will be grateful and look forward to future collaboration.

I would also like to express my profound gratitude to Dr.-Ing. Jorge Castro Godínez. Our paths converged with my degree's thesis and continued through the master's program, ultimately culminating in the PhD. He guided me most of the time, providing support and offering feedback on my research ideas, despite being unlinked to the PhD programme. He provided valuable insights and became a research partner with whom I expect to continue collaborating.

A special thanks goes to all the students who collaborated with my along the master's and the PhD, particularly with Eduardo Salazar, Alejandro Rodríguez, Esteban Campos, Erick Obregón, David Cordero, Alex Chacón, Diego Martínez, Samuel Castro, Nicolás Alfaro, Fabricio Elizondo, Romario Martínez, Manuel Bojorge, Anthony Leiva, Diego Ávila, Luis Prieto, Allan Navarro, Diana Esquivel, Gabriel Blanco, Gabriel Conejo and the new students who are starting their master's, Carlos Soto, Roger Morales, Fabiola Jiménez. Also to Niccolò Tosato for his contributions to the EfiMon project and my sister, Indra León, for her help in improving the diagrams.

The possibility of this PhD was thanks to eXact lab srl as a company that partially funded my PhD, and the PON programme by the Ministry of University and Research of Italy. I am thankful with the chance of participating in the programme. I also extend my gratitude to RidgeRun LLC for their support while I was away and for providing the hardware for my students' experiments. Moreover, I acknowledge the support of the AMD University Programme (HACC ETH Zurich) and AREA Science Park for providing the infrastructure for some of the experiments.

My most profound gratitude goes to my family and Manuela, who provided support throughout the entire process, from the start to the end of my PhD.

To all of them, my sincere thanks and never-ending good wishes.

Luis Gerardo León Vega

Trieste, Italia, January 13, 2026

List of Publications

The following publications were derived during the work presented in this dissertation:

Journals

1. L. G. León-Vega, E. Salazar-Villalobos, A. Rodriguez-Figueroa, *et al.*, "Automatic Generation of Resource and Accuracy Configurable Processing Elements," *ACM Trans. Embed. Comput. Syst.*, Apr. 2023, ISSN: 1539-9087. DOI: 10.1145/3594540

Conferences and Workshops

2. L. G. León-Vega, E. Salazar-Villalobos, and J. Castro-Godínez, "An Exploration of Accuracy Configurable Matrix Multiply-Addition Architectures using HLS," in *2022 IEEE 15th Dallas Circuit And System Conference (DCAS)*, 2022, pp. 1–6. DOI: 10.1109/DCAS53974.2022.9845501

3. L. G. León-Vega and J. Castro-Godínez, "Generic Accuracy Configurable Matrix Multiplication-Addition Accelerator using HLS," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2023, pp. 171–174. DOI: 10.1109/DSN-W58399.2023.00048

4. L. G. León-Vega, A. Chacón-Rodríguez, E. Salazar-Villalobos, *et al.*, "Acceleration of Fully Connected Layers on FPGA using the Strassen Matrix Multiplication," in *2023 IEEE 5th International Conference on BioInspired Processing (BIP)*, 2023, pp. 1–6. DOI: 10.1109/BIP60195.2023.10379257

5. Luis G. León-Vega and Diego Avila-Torres and Jorge Castro-Godínez, "CYNQ: Speeding Up FPGA Applications with Simplicity," in *2024 31st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2024

6. Luis G. León-Vega and Erick Obregón-Fonseca and Jorge Castro-Godínez, "A User-Friendly Ecosystem for AI FPGA-based Accelerators," in *2024 IEEE International Conference on Omni Layer Intelligent Systems (COINS)*, 2024

7. L. G. Leon-Vega, N. Tosato, and S. Cozzini, "A Comprehensive Analysis of Process Energy Consumption on Multi-Socket Systems with GPUs," *Latin American High Performance Computing Conference (CARLA)*, 2024. DOI: 10.1007/978-3-031-80084-9_4

8. L. G. Leon-Vega, N. Tosato, and S. Cozzini, "EfiMon: A Process Analyser for Granular Energy Prediction," *Latin American High Performance Computing Conference (CARLA)*, 2024. DOI: 10.1007/978-3-031-80084-9_8

9. L. D. Prieto-Sibaja *et al.*, "LLM Acceleration on FPGAs: A Comparative Study of Layer and Spatial Accelerators," in *2024 IEEE 42nd Central America and Panama Convention (CONCAPAN XLII)*, 2024, pp. 1–6. DOI: 10.1109/CONCAPAN63470.2024.10933896

10. D. Cordero-Chavarría, L. G. León-Vega, and J. Castro-Godínez, "Configurable High-Level Synthesis Approximate Arithmetic Units for Deep Learning Accelerators," in *2024 IEEE 42nd Central America and Panama Convention (CONCAPAN XLII)*, 2024, pp. 1–6. DOI: 10.1109/CONCAPAN63470.2024.10933846

11. A. Leiva-Valverde, F. Elizondo-Fernández, L. G. León-Vega, *et al.*, "A Quantitative Evaluation of Approximate Softmax Functions for Deep Neural Networks," *Accepted in 10th Workshop on Approximate Computing (AxC 2025)*, 2025. DOI: 10.48550/arXiv.2501.13379

The following software repositories were open-sourced:

Open-Source Software

12. E. Salazar-Villalobos, L. G. Leon-Vega, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Matrix Accelerator*, version v1.1.0, 2022. DOI: 10.5281/zenodo.6413238

13. A. Rodriguez-Figueroa, L. G. Leon-Vega, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Convolution Accelerator*, version v0.1.0, 2022. DOI: 10.5281/zenodo.6413243

14. D. Cordero-Chavarria, L. G. Leon-Vega, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Math Operators*, version v0.1.1, Feb. 2023. DOI: 10.5281/zenodo.7708216

15. L. G. Leon-Vega, D. Cordero-Chavarria, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Computing Executer (AxC Executer)*, version v0.1.0, Mar. 2023. DOI: 10.5281/zenodo.7712042

16. L. G. León-Vega, N. Tosato, and S. Cozzini, *EfiMon: An Instruction-Driven Process Energy Consumption Analyser for Multi-Socket Computers*, version v0.1.0, Apr. 2024. DOI: 10.5281/zenodo.11072569

17. León-Vega, L., Ávila-Torres, D., Castro-Godínez, J., *CYNQ (v0.2)*, 2024. [Online]. Available: https://github.com/ECASLab/cynq

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High-performance computing (HPC) has undergone a significant transformation in recent years, evolving from a domain focused primarily on traditional simulations and scientific computing to one increasingly driven by the needs of artificial intelligence (AI). At its core, HPC involves the accelerated transfer and processing of vast volumes of data, historically centred around Big Data and Accelerated Computing [18]. However, the emergence of modern AI workloads has drastically reshaped this landscape.

Initially, the massive parallelism offered by Graphics Processing Units (GPUs) revolutionised HPC, providing substantial speedups over Central Processing Units (CPUs) for certain classes of problems. The Single Instruction Multiple Thread (SIMT) paradigm enabled GPUs to process thousands of threads simultaneously [19], ideally suited for the growing demands of data-intensive applications. As AI, particularly Deep Learning (DL), became the dominant workload, the computational requirements outpaced the capabilities of traditional HPC architectures [20]. CPUs and GPUs have been evolving to include optimised hardware extensions for matrix multiplications, which are crucial for most computations [19], [21].

Large Language Models (LLMs) have further exacerbated these demands. Their versatility, robustness, and broad applicability across natural language processing, robotics, and productivity tools have made them central in the last decade. Consequently, companies like OpenAI, Meta, and xAI have invested heavily in GPU-accelerated infrastructures to support the training and inference of increasingly sophisticated models [22]–[24]. These expansions have significantly increased computational and energy footprints in data centres, intensifying sustainability concerns [25].

At the same time, the exponential growth of AI computing has highlighted the limitations of existing hardware. While GPUs excel at general-purpose parallelism, they are not always the most energy-efficient solution for AI-specific tasks. The race for faster and larger models has sparked interest in specialised hardware platforms that offer better performance per watt, prompting the exploration of alternatives such as Field Programmable Gate Arrays (FPGAs) and Processing-In-Memory (PIM) architectures [26]. It has also

made room for the exploration of new architectures that extend beyond traditional CPU and GPU architectures, encompassing a range of devices, from Application-Specific Integrated Circuits (ASICs) to reconfigurable devices like Field-Programmable Gate Arrays (FPGAs) [26]. Moreover, other non-traditional computing paradigms have emerged and demonstrated promising results, such as Approximate Computing (AxC) and deeply optimised code development, which focuses more on optimisation through specialisation [27].

AI computations have plenty of room for further innovation, from accelerating code with traditional architectures and non-standard optimisations to the growth of new AI-driven computing architectures.

Scope of This Thesis

This dissertation addresses two critical challenges arising from this new computational paradigm. First, it investigates the need for fine-grained energy accounting methodologies to monitor and predict process-level energy consumption in shared, heterogeneous environments without requiring execution isolation. The proposed EfiMon framework enables energy studies of CPUs and GPUs, providing a foundation for sustainable resource management and energy-driven optimisation.

Second, the dissertation explores the design of energy-efficient AI accelerators by leveraging the inherent reconfigurability of FPGAs. The development of the Flexible Accelerator Library (FAL) demonstrates how operand size tuning, approximate computing, and custom architecture generation can lead to significant improvements in both performance and energy efficiency for AI workloads. The analysis is taken beyond through the analysis of multiple architectures for accelerating the cutting-edge LLMs, studying the impact of each architecture and providing insights for further implementation guided by the operation fusion and approximate computing. The study concludes with democratising FPGAs and their application in AI acceleration, while also analysing existing FPGA solutions and proposing a hybrid library for FPGA access optimised for low latency and simplicity.

It concludes with an exploration of newly researched trends in computer architecture, emphasising the introduction of reconfigurability in Cooperative Heterogeneous Computing as a potential alternative for the rapidly evolving AI solutions that overwhelm hardware development times.

Contributions

Regarding the energy accounting front (Chapter 2), this work contributes:

- The development of a tool, called *EfiMon*, capable of gathering detailed process information regarding load footprint, system occupancy, and power consumption,

enabling the proposal of new prediction models to estimate the energy consumed by a Process of Interest (PoI) without requiring execution isolation.

- A comprehensive analysis of computing devices during the execution of running processes on multi-socket computers.
- An energy model for estimating the consumption of CPU and GPU resources during PoI execution, contribution in energy consumption research for shared computing nodes and next-generation supercomputers based on XaaS, where energy usage must be measured without dedicating full node exclusivity.

On the other hand, in energy-efficient AI acceleration, the contributions are:

- A framework for automatically generating vector processing elements (PEs) and accelerators for matrix multiplication-addition and convolution operations, supporting adaptable operand size, data bit-width, datatype, and arithmetic structure. The design leverages standard C++ and High-Level Synthesis (HLS), and allows the integration of approximate computing (AxC) to reduce resource consumption at the cost of minimal accuracy trade-offs (Chapter 3).
- The implementation and evaluation of Strassen and Winograd Matrix Multiplication algorithms on FPGA for Deep Neural Networks (Chapter 3).
- Implementation of a basic set of accuracy-configurable arithmetic units for addition and multiplication, developed in untimed C++ for HLS, parameterised by data type, data width, integer part size, and number of approximated least-significant bits (Chapter 4).
- An evaluation of the arithmetic units' effectiveness when deployed in a LeNet-5 model and a Multi-Layer Perceptron (MLP) autoencoder for anomaly detection tasks (Chapter 4).
- An analysis of different architectures for LLM inference, comparing various implementations and pioneering a feasibility study based on resource consumption and latency, aimed at guiding the design of future accelerators for LLAMA-based and Vision Transformer (ViT) models (Chapter 5).
- An analysis of the XRT and PYNQ frameworks regarding latency when scaling workload sizes, leading to the development of a simplified runtime library, *CYNQ RT*, featuring a PYNQ-like API for C/C++ application development (Chapter 6).
- A framework to lower the barriers to FPGA usage at the software level by proposing an open-source framework, CYNQ Framework, that provides access to a set of selected generic accelerators through a simple API, eliminating the need for in-depth hardware knowledge (Chapter 6).

Ultimately, this work positions reconfigurable, energy-conscious computing as a cornerstone for future AI-driven high-performance systems (Chapter 7). It aims to contribute to a broader vision where adaptability, sustainability, and computational excellence coexist to meet the demands of the next generation of intelligent applications.

Document Structure

The remainder of this dissertation is organised as follows:

- Chapter 1 introduces this dissertation.
- Chapter 2 discusses energy accounting in next-generation supercomputers and introduces the EfiMon tool.
- Chapter 3 describes the design principles and automated generation of FPGA-based AI accelerators.
- Chapter 4 explores the integration of approximate computing techniques in AI accelerators.
- Chapter 5 presents architectural designs for accelerating LLMs using FPGAs and analyses their performance.
- Chapter 6 introduces the CYNQ framework for facilitating FPGA usage by software developers.
- Chapter 7 outlines insights and future directions for next-generation AI computing architectures.
- Chapter 8 concludes the dissertation, highlighting the contributions and proposing future research directions.

Chapter 2

Energy Accounting in Next-Generation Supercomputers

Artificial Intelligence (AI) is currently the most prevalent and computationally demanding workload in supercomputing. Some workloads are executed on the cloud, where several users share resources. The energy footprint of these workloads is often inadvertently overlooked when releasing training and inference results. However, the rapid growth in computational demands is pushing these resources to their limits, making energy the most critical asset in data centres. Overall, AI is driving large-scale investments in supercomputing, increasing power consumption exponentially to meet demand, posing a risk to carbon emissions reductions. The Top 5 most powerful supercomputers require 7 MW up to 30 MW to reach the maximum capacity [25]. In Italy in 2022, 63.4% of electricity was produced by fossil fuels, and 27.5% by renewable energy sources such as wind, solar, and hydro [28], posing an issue for how electricity is produced to meet the increasing demand from HPC.

Currently, accounting for energy in HPC workloads is limited to the node level, without a clear distinction between the actual distribution of computation, communication, and energy consumption among running processes. This lack of granularity hampers our understanding of how power-hungry the workloads are and which parts are the most energy-demanding.

This chapter focuses on the quantification of energy consumption in supercomputing environments. It presents a comprehensive model derived analytically from computer architecture principles and introduces *EfiMon*, a tool designed to extract metrics that estimate the energy consumption of processes during execution. Notably, this estimation does not require the system to be fully dedicated to the process under analysis. Together, the model and tool show promising results, guiding the future of energy estimation for supercomputing workloads, including those associated with AI.

2.1 Related Work

Quantifying the energy consumption of computational systems is an open research area, encompassing everything from embedded high-performance computing (HPC) systems to supercomputers. Most modern systems include sensors that allow for either direct or indirect measurement of power consumption by various subsystems, including processors, main memory, power supply units (PSUs), and acceleration cards [29]–[31]. These measurements generally offer a coarse view of system-wide energy use.

Current server hardware provided by major vendors offers limited tools for exploring and analysing energy consumption in HPC facilities. This leads to limited instrumentation available on compute nodes for accurately quantifying energy expenses, which, in turn, restricts decision-making on when and how to allocate resources efficiently.

For example, Intel provides tools such as PCM and RAPL that allow measurement of aggregate core, socket-level consumption (including non-processing components), and DRAM power consumption [29]. In contrast, AMD's offerings are more limited, often excluding DRAM measurements [29], and some tools, like AMD uProf, are closed-source [30], restricting comprehensive energy consumption analyses.

SLURM [32], the most popular job scheduling software, does not optimise or make any decision on job scheduling based on energy consumption. Instead, it only provides the energy consumed system-wide by a job based on the energy meters available, like IPMI and Intel RAPL, and turns on or off the nodes to save energy.

One of the main concerns regarding SLURM's approach is that it fails to account for the energy consumption of individual processes or tasks, unless they run in complete isolation within the system. This is because Intel RAPL and IPMI operate at a system-wide level. In some cases, these tools are also inaccessible or inaccurate, leaving a gap in the available metrics. To address this, other methods have been proposed to approximate the power consumption for server computers, such as CPU utilisation, temperature, and fan speed as input for estimations [33]–[37]. Some researchers have also suggested more accurate measurement approaches using stochastic methods or probabilistic models to describe the system as a whole [35], [38], which have shown promising results in experiments.

Fine-grained energy consumption analyses typically focus on embedded systems, which are simpler and often powered by batteries. These studies have analysed the impact of executing assembly instructions, such as load/store [39], Very Long Instruction Words (VLIW) [40], [41], including some outdated GPUs [42]. Some proposals include power models to extract energy consumption based on the dispatched instructions, while others utilise heuristics at compile time [43]. These approaches are feasible given that the computing hardware (i.e. CPU, DSP or GPUs) consumes most of the power. Nevertheless, EAR [44] is an open-source framework for energy management in HPC that targets architectures with Intel Node Manager and RAPL, providing near-fine-grained control. It uses the Performance API (PAPI) library [45] to capture performance metrics, such as clocks-per-instruction (CPI), time-per-instruction (TPI), and running/iteration times.

It attempts to link the information to the running environment, particularly the node's power modes (pstates) and the energy supplied by the power supply. EAR makes an adequate step towards energy accounting for processes and energy-driven node optimisation in Intel-based and NVIDIA-based architectures.

From the GPU perspective, some studies evaluate the effect of memory transactions depending on the type of access pattern [46], where some interesting insights have been found regarding the sequential and strided memory accesses. There are also studies analysing the kernel dispatching, suggesting that the multiple kernel execution also plays a role in how the energy is consumed [47]. Others have benchmarked several workloads to obtain power profiles [48] and developed static analysis estimation tools after a power profiling [49], [50], which are suitable for execution in isolation.

The scenario changes significantly for server-grade computers with increased complexity, where some fine-grained energy strategies are insufficient, unsuitable, or require further study. The study of the energy consumption in these computers must include cooling, storage, networking, memory and multi-socket CPU consumption. A survey from 2016 suggests that 50% is used for cooling, 10% for storage and 10% for networking [51]. Another survey from 2020 reported that the CPU takes 32% of the power consumption [52], changing the scenario concerning the embedded system case, implying more components than only analysing the CPU's power.

The estimation of energy consumption in server-class computers has primarily been approached by the research community through modelling techniques that incorporate parameters such as CPU utilisation and frequency, memory and disk usage, temperature, fan speed, and data throughput (e.g., bandwidth) [52]. The models proposed by related work are classified into different types, depending on the formulation. The most fundamental is the additive model, which describes the total power as the sum of all the loads [51], [53], [54]. The Baseline + Active (BA) models decompose the power load into base power (idle), active power (during computation), and a correction term, which can be interpreted as static power [51], [55]. Regression models are fitted using training data, which can be linear, non-linear, or based on an existing power model [51]. However, all these models focus on general system-wide prediction rather than single-process estimation.

Therefore, there is an opportunity to contribute to the field of process-level energy consumption (fine-grained) and to use executed instructions as a parameter to enhance energy consumption estimation. This work will focus on proposing a tool to gather information to fit a regression model that takes into account the instructions executed by a PoI, and its load on the system. It will also focus primarily on studying the energy impact of different types of instructions (scalar, vector, branching, and memory), the use of other computational resources, and how these can be used as parameters for energy consumption estimation of a single process by combining additive, BA and regression approaches.

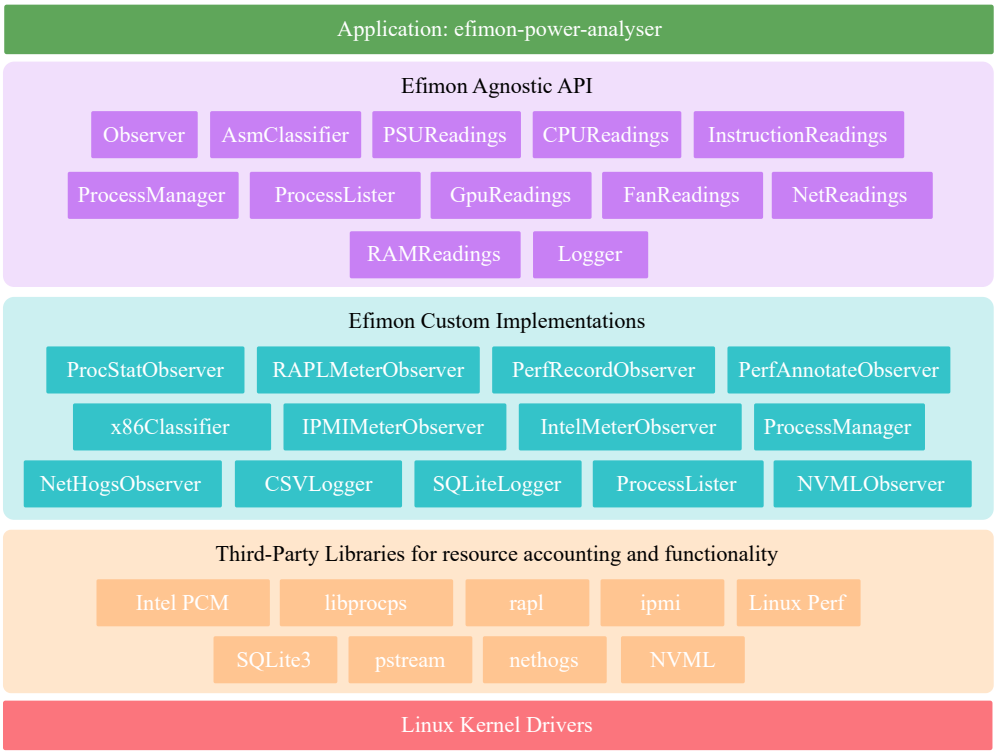


Figure 2.1: Efimon software stack. The architecture uses the interface-adapter pattern to decouple the dependencies and keep the API uniform and agnostic.

2.2 EfiMon: Agnostic Performance Monitor

EfiMon is a tool composed of a C++ library and applications to extract information from the process execution and the overall system metrics. From the process perspective, EfiMon extracts the instructions executed, sampled at a certain rate during a time window, and includes the CPU, RAM and network utilisation. From the system perspective, it extracts critical information about power consumption from the CPU sockets and Power Supply Units (PSU), as well as the overall system load with respect to the CPU, RAM, GPU and network.

EfiMon's library follows the interface-adapter architecture combined with the observer pattern. The architecture allows EfiMon to standardise the API, making it extensible through the adapter implementation. For instance, CPU socket power consumption can be obtained from Intel PCM, RAPL, or AMD uProf. Depending on the system where it is compiled, the adapters are enabled, offering the possibility to get measurements from any of them while preserving the API, making the changes to the final application less impactful.

Figure 2.1 shows the software architecture of the EfiMon project, summarising the different levels of abstractions from the hardware drivers up to the final user application. From bottom to top, EfiMon's software stack uses Linux libraries like *libprocps* to ac-

Table 2.1: Observer implementations and their readings specialisation

Observer Implementation	Dependency Used	Readings Implemented
ProcStatObserver	libprocps	CPUReadings, RAMReadings
RAPLMeterObserver	rapl	CPUReadings
PerfRecordObserver	perf, trace	N/A
PerfAnnotateObserver	perf, x86Classifier	InstructionReadings
IPMIMeterObserver	ipmi	FanReadings, PSUReadings
IntelMeterObserver	Intel PCM	CPUReadings
NetHogsObserver	nethogs	NetReadings
NVMLObserver	nvml, cuda compiler	GPUReadings

cess the process statistics, *Linux Perf* to access kernel events and hardware counters, the *RAPL* interface and *Intel PCM* to get information about the CPU energy statistics, and *IPMI* for the PSU mean power and fan speeds. For housekeeping and auxiliary functions, EfiMon uses *SQLite3* to save recordings in a database file and *pstream* to process deployment and signalling. EfiMon also includes tools to measure network traffic using *NetHogs* and GPU statistics using *NVML* for a specific process; however, they will be analysed in future work.

For abstractions, it proposes two user-accessible interface classes called *Observer* and *Readings*. The *Observer* standardises basic functionalities such as reading results, triggering metering and common constructors that set the process ID, sampling frequency, metering interval and reading scope. The Observer interface class is implemented using adapters, concrete classes that wrap up the above dependencies. Table 2.1 shows all the implementations in EfiMon. Each implementation uses a single dependency, helping with conditional compilation and the availability of tools after the project's construction. The *Readings* specialisable structure is a base data container that sets basic members such as timestamp and time difference. This structure is intended to be extendible to include measurements, depending on the observer. Each *Observer* adapter fills a specialisation of the Readings class, standardising the metrics' storage and keeping the API uniform across observers. For instance, the *ProcStatObserver* and the *RAPLMeterObserver* specialise the *Readings* class into *CPUReadings* to hold the CPU metrics. The *Readings* subclasses can have unfilled members. In the case of *ProcStatObserver*, it only fills the CPU utilisation members, whereas the *RAPLMeterObserver* fills the power measurements.

Listing 1 shows an oversimplified example of using EfiMon's API within a metering application. Lines 2 and 3 show how to create system-wide observers, and line 4 shows how to create a process-scoped observer (attached to `pid`). By default, if the constructor is left empty, it sets the PID and the interval to 0 and the scope to the default for the observer. This may vary from one observer to another.

The `Trigger` method performs a reading to the adapter, gathering the information about the CPU socket (line 7), IPMI (line 8) and libprocps (line 9). Internally, this method queries all the required sensors or functions to refresh the internal measurements for later use with the `GetReadings` method.

Listing 1 Oversimplified example of the EfiMon's library usage for RAPL, IPMI and libprocps measurements.

```
1  // Create Observers
2  auto rapl_obs = RAPLMeterObserver{};
3  auto ipmi_obs = IPMIMeterObserver{};
4  auto proc_obs = ProcStatObserver{pid, efimon::ObserverScope::PROCESS};
5
6  // Trigger a reading
7  rapl_obs.Trigger();
8  ipmi_obs.Trigger();
9  proc_obs.Trigger();
10
11 // Get readings
12 CPUReadings *cpu_readings =
13     dynamic_cast<CPUReadings *>(rapl_obs.GetReadings()[0]);
14 PSUReadings *psu_readings =
15     dynamic_cast<PSUReadings *>(ipmi_obs.GetReadings()[0]);
16 FanReadings *fan_readings =
17     dynamic_cast<FanReadings *>(ipmi_obs.GetReadings()[1]);
18 CPUReadings *cpu_readings_pid =
19     dynamic_cast<CPUReadings *>(proc_obs.GetReadings()[0]);
20 RAMReadings *ram_readings_pid =
21     dynamic_cast<RAMReadings *>(proc_obs.GetReadings()[1]);
22
23 // Accessing to a member
24 float psu_power = -1.f;
25 if (psu_readings) psu_power = psu_readings->psu_power.at(0);
```

The `GetReadings` method returns a vector of pointers to *Readings*, whose memory segment points to the internal readings of the observer instance. To access the observer's measurements, the *Readings* pointer type must be cast to the actual subclass, as illustrated in lines 12-21, containing the members for a specific measurement (line 25). Using this approach, the API is standardised, and the readings can include polymorphism, making them flexible for future extensions.

The tool is available online on Zenodo [8].

2.3 Fine Grain Power Consumption Analysis

The instantaneous power consumption of a server computer can be modelled as an additive model, including the power consumption of the CPU, hardware accelerators (i.e. GPUs), RAM, storage, network interface cards (NIC), cooling (fans), and other electronic components [54], which involves the following power model at an instant t:

$$P_{\text{system},t} = P_{\text{CPU,t}} + P_{\text{Accel,t}} + P_{\text{RAM,t}} + P_{\text{Disk,t}} + P_{\text{NIC,t}} + P_{\text{Cool,t}} + P_{\text{Aux,t}} \quad (2.1)$$

All these power components can vary their values over time based on their utilisation and power domain status (on or off). For instance, CPU-intensive workloads will cause

an increase in the $P_{\text{CPU,t}}$, as well as for HW-accelerator workloads, which increase the $P_{\text{Accel,t}}$. Nevertheless, the increase starts from a base power value, which corresponds to the idle state of the device, which approximates its static power consumption. In this case, the first proposal is to decompose each power component into its static and dynamic components, where the latter depends on the activity of the device:

$$P_{\text{device},t} = P_{\text{dynamic},t} + P_{\text{static},t} \tag{2.2}$$

which corresponds to a BA model [55]. This model can be further extended to consider the status of the hardware, taking into account the existence of power domains that can be switched off for energy savings:

$$P_{\text{device},t} = \big(P_{\text{dynamic},t} + P_{\text{static},t}\big)u + P_{\text{suspend},t}\big(1-u\big) \tag{2.3}$$

where $u = \{0,1\}$ is a status variable, that takes a 0 value if the device is suspended or 1 if it is on, similar to a Heaviside function. For this work, assume that the device is always on ($u = 1$), leaving the revisit of this assumption for future work.

On the other hand, some workloads utilise more than one device at a time. For instance, a CPU-based matrix multiplication uses the CPU and the RAM (if all operands are loaded into memory). Therefore, the workloads are classified based on their computational nature, i.e. CPU-based workloads, GPU-based workloads, storage operations and network communication. This is possible due to the conservation of energy, which states that the power supplied by a power source is equal to the sum of all the power loads. Moreover, it is possible to characterise any electrical circuit using the superposition principle, meaning that activating parts of the circuit allows for characterising their individual load profiles [54].

This section will be divided into computational natures (CPU and GPU). It will analyse each component and use actual measurements to define the behaviours of the loads, using a set of benchmarks that can exercise several parts of the computer. Network and disk activity will be left for future contributions.

2.3.1 CPU Energy Model

For this work, the first assumption is that obtaining metrics of the total CPU consumption and the power delivered by the PSU is possible. For simplicity, this work excludes other computing components like GPUs during this analysis and uses the superposition principle explained earlier. In this context, the model can be arranged as follows:

$$P_{\text{system},t} \approx P_{\text{PSU},t} = P_{\text{CPU,t}} + P_{\text{Other,t}} \tag{2.4}$$

where $P_{\text{Other,t}}$ encapsulates the consumption of all hardware except the CPU. Additionally, this assumes that no other devices are active during the computation. The CPU

consumption involves the addition of non-processing components, such as memory and peripheral controllers, plus the consumption of each core:

$$P_{\text{CPU,t}}(\mathbf{f}, \mathbf{T}, \mathbf{w}) = \sum_{i=1}^{N_c} P^i_{\text{core},t}(f^i, T^i, w^i) + P_{\text{CPU,other},t}(\mathbf{f}, \mathbf{T}) \tag{2.5}$$

where $P^i_{\text{core},t}$ shows the power consumption of the i-th core and $P_{\text{CPU,other}}$ represents the power consumed by the other parts that integrate the processor [36]. In this model, we consider that the frequency (f), temperature (T), and system workload (w) can be different for each core, represented as vectors ($\mathbf{f}, \mathbf{T}, \mathbf{w}$, respectively). Assuming that $P_{\text{CPU,other}}$ is not part of any core and does not depend on the $\mathbf{w}$ and most of the power is consumed by the cores, we can focus on these latter components, which can be approximated by a BA submodel:

$$P^i_{\text{core},t}(f, T, w) = P^i_{\text{static},t}(f, T) + P^i_{\text{dynamic},t}(f, T, w) \tag{2.6}$$

where $P^i_{\text{static},t}$ represents the static power consumed by the core and does not depend on the system workload, and $P^i_{\text{dynamic},t}$ the dynamic power [37], [56]. Up to this point, the dynamic power is required, which varies depending on the frequency, operation type, workload, and temperature. One of the most comprehensive dynamic power models is given by $P^i_{\text{dynamic},t} = y_i \alpha_i f_i^{\beta_i}$, where y_i is the core state, α_i and β_i are system-specific parameters associated to the voltage, switching activity (implicitly the workload) and capacitance of the transistors, and f_i is the core frequency [56].

To handle the CPU usage, some works estimate the average full-load dynamic power $P^i_{\text{max(dynamic)},t}$ and multiply it by the usage, such that $P^i_{\text{dynamic},t} = w^i P^i_{\text{max(dynamic)},t}$, where w^i is the workload measured as core utilisation from 0 to 1 [37]. Nevertheless, this approximation is inaccurate since the core utilisation is often represented by the portion of the time the CPU is active in a given period, ignoring the type of instruction executed. The operating system (OS) may report 100% CPU utilisation on scalar operations and 100% on vector operations, yet energy consumption varies between these numbers.

The second assumption in the power model is that the power per core is unquantifiable, and the CPU is fixed at its base clock frequency and average temperature, with low variance, to maintain simplicity in the first iteration of this work. This is a limitation of some systems that do not have instruments per core, like ARM-based systems or x86, which provide metrics based on heuristics. It simplifies the analysis for further steps and the scope of this work. This assumption will be deferred to future work, in which the clock frequency will be reintegrated. In this case, refer back to equation (2.5), such that

$$P_{\text{CPU,t}}(\bar{f}, \bar{T}, \hat{w}) = P_{\text{dynamic},t}(\bar{f}, \bar{T}, \hat{w}) + P^i_{\text{static},t}(\bar{f}, \bar{T}) + P_{\text{CPU,other},t}(\bar{f}, \bar{T}) \tag{2.7}$$

where equation (2.6) is reformulated to rewrite the sum of the dynamic power per core as the dynamic power of the entire CPU, which now depends on the mean frequency ($\bar{f}$) and temperature $\bar{T}$ and the total CPU workload $\hat{w}$. Likewise, we can encapsulate the sum of

the static power per core to be the $P_{\text{CPU,other},t}(f^i, T^i)$, assuming that the entire CPU will always be on, therefore

$$P_{\text{CPU,t}}(\bar{f}, \bar{T}, \hat{w}) = P_{\text{dynamic},t}(\bar{f}, \bar{T}, \hat{w}) + P_{\text{CPU,other},t}(\bar{f}, \bar{T}) \tag{2.8}$$

Building on the model from equation (2.8), this work defines the dynamic power of the CPU ($P_{\text{dynamic},t}$) and proposes the introduction of a more comprehensive utilisation parameter based on the types of instructions executed in a time window, such as the scalar/vector arithmetic, logic, memory and branching, and distributing the utilisation of the CPU and power consumption according to the impact of the instruction types. This modifies (2.8) to

$$P_{\text{CPU,t}}(\bar{f}, \bar{T}, \hat{w}, \mathbf{h}) = P_{\text{dynamic},t}(\bar{f}, \bar{T}, \hat{w}, \mathbf{h}) + P_{\text{CPU,other},t}(\bar{f}, \bar{T}) \tag{2.9}$$

where the dynamic power is

$$P_{\text{dynamic},t}(f, T, w, \mathbf{h}) = l(w, \mathbf{h}, f, T)\theta(f, T)\bar{P}_{\text{max(dynamic)}}(f, T) \tag{2.10}$$

$l(w, \mathbf{h}, f, T)$ is a function that quantifies the impact of each instruction given a histogram vector $\mathbf{h}$ in a time window, $\theta(f, T)$ is a non-linear function that depends on the CPU intrinsic parameters, like electrical impedance, $\bar{P}_{\text{max(dynamic)}}$ the mean power when the CPU is loaded to its maximum capacity and w is the CPU utilisation reported by the OS, whose values can oscillate between 0 and N_c, where $w = N_c$ means that all CPU cores are being used 100%.

For the assumptions made so far, the CPU's frequency is fixed to a single value, and the temperature is almost constant by setting the fans to the maximum speed during the experiment. It simplifies the problem of finding the function l that describes the impact of the instructions and the process utilisation. Besides, we are grouping the instructions into scalar and vector families and arithmetic, logic, memory and branch types, where each family has all instruction types. Therefore, the CPU power can be defined as

$$P_{\text{CPU},t}(w, \mathbf{h}) = \hat{\theta} l(w, \mathbf{h})\bar{P}_{\text{max(dynamic)}} + \hat{P}_{\text{CPU,other},t} \tag{2.11}$$

where $\hat{\theta}$ is an estimation parameter that represents the CPU intrinsic parameters and their affectation by the operating frequency and temperature, and $\hat{P}_{\text{CPU,other},t}$ is an estimated parameter that represents the static and base power of the CPU. The equation (2.11) can be decomposed to apply a linear regression with a transformation on w and $\mathbf{h}$ that can describe possible non-linearities:

$$P_{\text{CPU},t}(w, \mathbf{h}) = \sum_{k=1}^{S} \gamma_k \sigma(h_k, w) + \hat{P}_{\text{CPU,other},t} \tag{2.12}$$

where γ_k are the coefficients to determine through linear regression, compressing $\hat{\theta}$ and $\bar{P}_{\text{max(dynamic)}}$, S is the number of instruction types and σ is a function that maps each

instruction type and the process utilisation. The linear regression also determines the intercept $\hat{P}_{\text{CPU,other},t}$, corresponding to the processor's idle consumption.

Equation (2.12) must adhere to energy conservation principles. It implies that the total dynamic power consumed by N_p processes is the result of the sum of all the dynamic power consumed by each process p:

$$P_{\text{CPU},t} = \sum_{p=1}^{N_p} \sum_{k=1}^{S} \gamma_k \sigma(h_k^p, w_p) + \hat{P}_{\text{CPU,other},t} \tag{2.13}$$

where N_p is the number of processes, h_k^p is the probability of the instruction type k and w_p is the CPU utilisation of the process p. Therefore, the model from equation (2.13) assumes that we have measurements of the power from either the PSU or the CPU; the frequency and the temperature are fixed, with low variance and the same for all cores, the power domain is always on, and the instructions can be collected for the processes using an OS tool.

2.3.2 GPU Energy Model

Graphics Processing Units (GPUs) can be seen as a particular type of CPU architecture with a massive number of execution units. According to NVIDIA naming conventions, a GPU is composed mainly of Stream Multiprocessors (SM), a DRAM memory bank, and caches. The SMs can be likened to simplified CPU processing cores with in-order execution, having their own scheduler, caches, and several execution units, usually a multiple of the warp size (32). Unlike CPUs, each SM launches many threads that execute the same instruction simultaneously across the execution units. This is the opposite of how CPUs assign threads for execution; CPUs assign threads to each core, whereas GPUs collect instructions from threads to execute them all simultaneously [19], [57]. Recent architectures have optimised this process to avoid underutilisation [57].

To the best of the knowledge presented in this work, non-invasive, fine-grained GPU energy measurement has not been explored using online analysis. However, a tool within the CUDA toolchain called NVML [58] captures information about GPU energy consumption, temperatures, utilisation, and clock speeds. Furthermore, with the assumption stated above regarding the architecture, we can proceed with an analysis similar to the one performed for the CPU. Equation (2.12) shows the power consumed by the CPU. We can use this equation as a reference, considering that the GPUs are often expansion cards that communicate with the host systems through PCIe. In the case of NVIDIA GPUs, they have power meters to account for the energy consumption of the entire GPU module [58]. Therefore, the power consumption of the GPU card can be expressed as:

$$P_{\text{GPU},t}(w, \mathbf{h}) = \sum_{k=1}^{S} \gamma_k \sigma(h_k, w) + \hat{P}_{\text{GPU,other},t} \tag{2.14}$$

Table 2.2: Hardware Configuration

Comp.	AMD-based Node	Intel-based Node
CPU	2 X AMD EPYC 7H12 64 cores	2 X Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
RAM	16 X 32GB 3200 MT/s DDR4 ECC	12 X 64GB 2666 MT/s DDR4 ECC
Disk	2 X 480GB SSD - RAID 0	2 X 480GB SSD - RAID 0
NIC	MT28908 Family 100Gb (Infiniband)	MT28908 Family 100Gb (Infiniband)
Cooling	16 x Dell (Silver grade) fan	16 x Dell (High Performance) fan
PSU	2 x 1400 W redundant PSU	2 x 1100 W redundant PSU

where $\mathbf{h}$ is the instruction histogram obtained from the acceleration kernel object (GPU assembly code), σ is a function that summarises the behaviour of each instruction type h_k and the GPU compute usage w_p linked to the process p to the power consumption, γ_k represents the impact of the instruction on the power consumption and $\hat{P}_{\text{GPU,other},t}$ represents the power consumption from non-computational peripherals inside of the GPU card.

2.4 Quantitative Assessment of the Fine Grain Power Consumption Model

2.4.1 Setup and Benchmarks

To assess the models proposed in the previous section, the analysis will divide the experiments into computational natures (CPU and GPU), analysing each component and using actual measurements to define the behaviours of the loads, using a set of benchmarks that can exercise the several parts of the computer.

For the CPU, this work specifically uses the hardware from Table 2.2 and, to reduce the noise during the measures, the clock speeds have been fixed at 1.5 GHz (AMD) and 1.0 GHz (Intel), C-states and boost are disabled, and the fan speed is set to maximum. Likewise, the experiments performed are the following:

- **idle**: measures the idle power of the system when executing nothing.
- **copy**: performs memory copies using scalar instructions.
- **copy_mem** and **copy_mem_avx**: performs memory copies using scalar and vector units (AVX) but avoiding temporal storage, exploiting the bandwidth in NUMA machines.
- **daxpy_mem_avx_fma**: a double-precision linear combination of two vectors optimised for AVX FMAs and non-temporal stores.
- **stream_mem_avx_fma**: a double-precision stream triad $a_i = kb_i + c_i$, optimised for AVX FMAs and non-temporal stores.

Table 2.3: PowerEdge R740xd Hardware Configuration

Component	Description
CPU	2 X Intel(R) Xeon(R) Gold 6226 CPU @ 2.70GHz 12 cores
RAM	8 X 32GB of 2933 MT/s DDR4 RAM ECC
GPU	2 X NVIDIA Tesla V100 PCIe 32GB
Disk	2 X 480GB SSD - RAID 0
NIC	MT28908 Family [ConnectX-6] 100Gb (Infiniband)
Cooling	6 x Performance Fans for R740/740XD
PSU	2 x 1600 W redundant PSU

- **load**: double-precision memory loads using non-temporal data and AVX-vector instructions.
- **store**: double-precision memory stores using non-temporal data and AVX vector instructions.
- **peakflops_avx_fma**: double-precision multiplications and additions with a single load, optimised for AVX FMA.
- **update_avx**: AVX double-precision vector update (load and store).
- **dgemm**: double-precision matrix multiplication from Level 3 BLAS.

These experiments imply the running of several scenarios given the instructions activated during execution, implying a change in the probability distribution of instructions and multi-threading using OpenMP to stress power consumption during execution. All benchmarks can be found in the likwid-benchmark utility [29], and the **dgemm** is implemented with the OpenBLAS library [59].

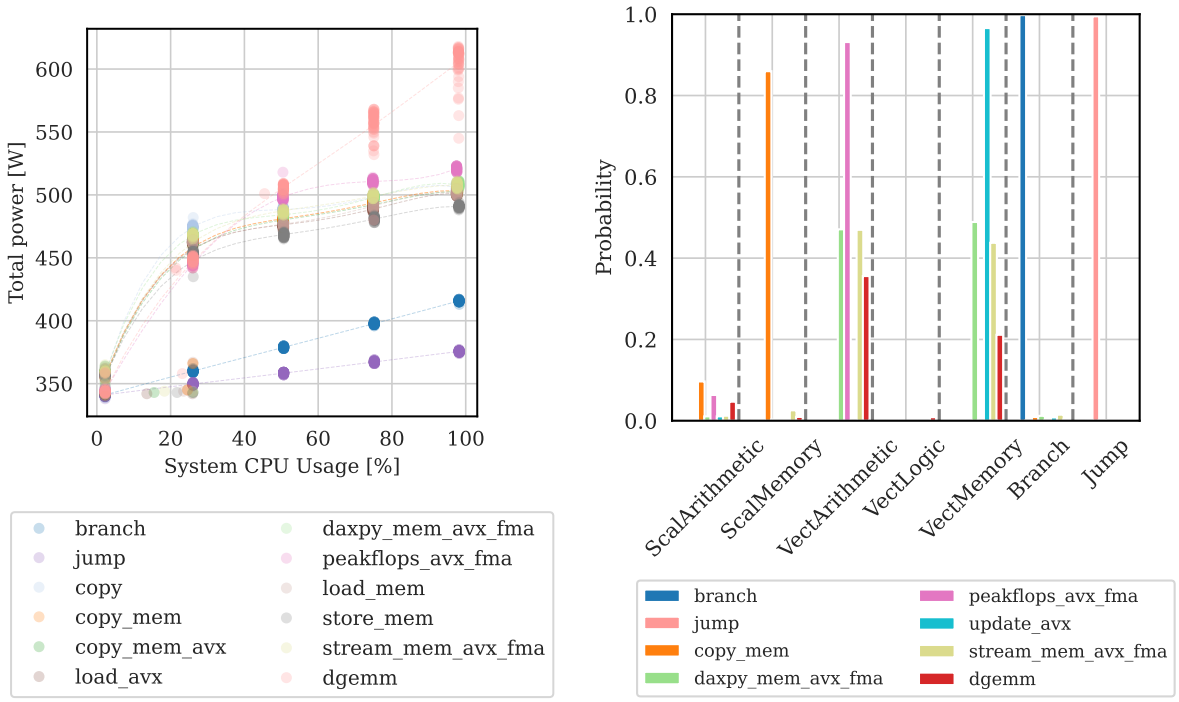
For the GPU, the hardware from Table 2.3 is used, and the benchmarks are a matrix multiplication implemented in three different ways: pure CUDA, CUBLAS, and vectorised through Tensor Cores [19]. The power profile of the GPU has also been adjusted to minimise variations during execution.

Experiments using hardware acceleration, storage, network and Inter-Process Communication (IPC) are excluded from the scope of this work and will be left for future contributions.

2.4.2 CPU Model Results

Behavioural Instruction Analysis

Figure 2.2 depicts a series of experiments that perform computations using scalar, vector and memory operations. Figure 2.2a shows how the aggregated power consumption (measured from the PSU) increases as the system CPU utilisation scales, illustrating a



(a) CPU Power Consumption

(b) Histogram of Operation Types

Figure 2.2: Experiments behaviour when scaling in degree of parallelism (System CPU Usage) and the instructions executed. On the left, the socket power consumption is reported as the usage scales, whereas on the right, the histogram of the instructions illustrates the experiment's nature.

logarithmic behaviour in most experiments except in the branch and jump experiments, where the behaviour is linear. Figure 2.2b serves as a reference to highlight that each experiment has a different probability of launching certain types of instructions. Both plots highlight the crucial role of the instructions on the overall power consumption, where the former varies according to the type of instruction launched by each benchmark, proving the need to integrate the instructions into the power modelling. Furthermore, the data was collected using EfiMon and the machine from Table 2.2.

According to the Figure 2.2, the nature of the impact of each instruction can be modelled by observing the instruction histograms and the utilisation, resulting in a hybrid linear-logarithmic combination. The impact of the instruction is also conditioned by the degree of parallelism, given implicitly by the system usage, which denotes how many CPU cores are used during the execution. Therefore, the σ functions (proposed by Equation (2.12)) that summarise the impact of w and $\mathbf{h}$ are:

$$\sigma(h_k, w_p)_1 = h_k^p \ln(w_p + 1) \tag{2.15}$$

$$\sigma(h_k, w_p)_2 = h_k^p w_p \tag{2.16}$$

where k and p are the indices for the instruction type (classified into scalar and vector types, and memory, branching, arithmetic and logic families) and the process index, respectively. The $\sigma(h_k, w_p)_1$ is used by the memory instructions (scalar and vector) and

Table 2.4: Parameters Estimated using NNLS Regression for CPU models using PSU Power Model from equation (2.17) as prediction reference

Parameter Name	Power Model (AMD)	Power Model (Intel)
Intercept ($\hat{P}_{\text{static},t}$)	336.5031	282.934
Weight of Scalar Arithmetic (γ_{sa})	0.6717	0.000
Weight of Scalar Memory (γ_{sm})	35.6589	41.117
Weight of Scalar Logic (γ_{sl})	0.00000	1.3363
Weight of Vector Arithmetic (γ_{va})	38.6822	34.510
Weight of Vector Memory (γ_{vm})	35.3435	42.242
Weight of Vector Logic (γ_{vl})	154.5258	59.149
Weight of Branch (γ_b)	0.6459	10.800
Weight of Jumps (γ_j)	0.3239	0.00000

the vector arithmetic. The $\sigma(h_k, w_p)_2$ is used for the rest of instruction types (scalar arithmetic, logic, branch and jumps).

On the other hand, considering the previous statement that, in a computing execution, more than one device is involved, the equation can be generalised to include other peripherals, such as caches, memories, and controllers, leading to:

$$P_{\text{PSU},t} \cong \sum_{p=1}^{N_p} \sum_{k=1}^{S} \gamma_k \sigma(h_k^p, w_p) + \hat{P}_{\text{other},t} \tag{2.17}$$

where the model includes the PSU and other peripherals, including the CPU, RAM, and other electronics with a passive consumption in $\hat{P}_{\text{other},t}$. In order to perform a regression of the model, we can stick to the isolation principle, executing a process at a time, thanks to the superposition principle, allowing the removal of the sum over $p \in \{1,2,\ldots,N_p\}$. Finally, in order to extract the process energy consumption, we can use the calculated regression model and the dynamic part:

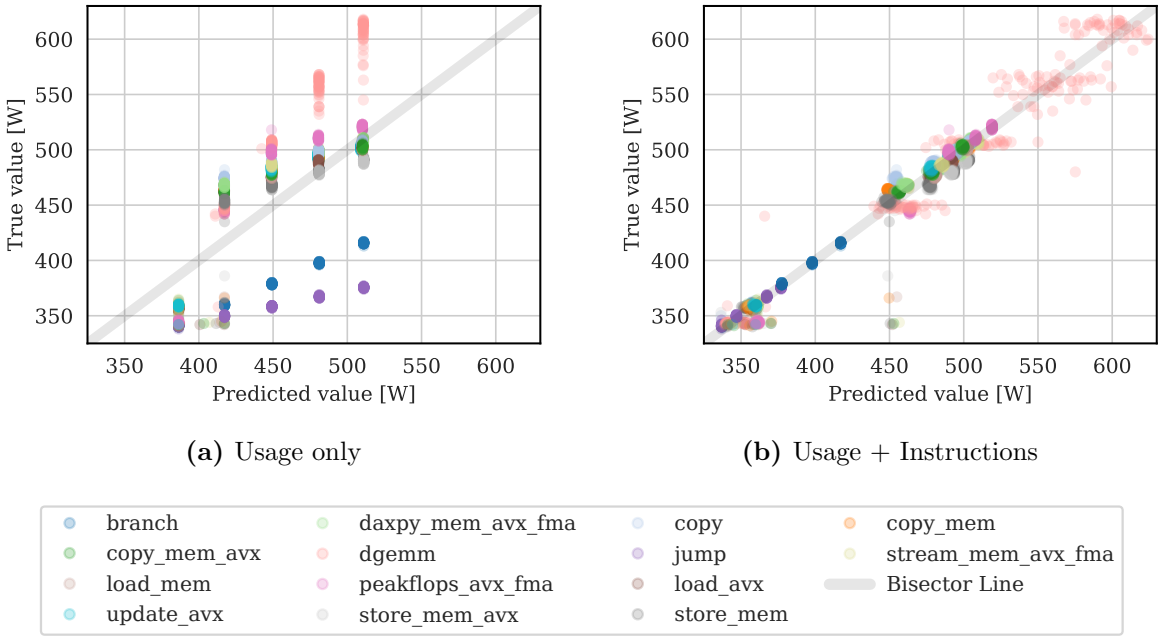
$$P_{\text{p},t} = \sum_{k=1}^{S} \gamma_k \sigma(h_k^p, w_p) \tag{2.18}$$

Standalone Model Fitting

With σ defined, a linear regression using the equation (2.17) can be performed to find $\hat{P}_{\text{static},t}$ and the parameters γ_k. The linear regression estimator is fit with the total PSU power consumption (adding both PSUs), the histogram, process CPU utilisation, and the number of cores.

Table 2.4 shows the non-negative least square regression (NNLS) results for eight types of instructions using the PSU as the power consumption reference and the AMD and Intel-based nodes. Other types were negligible and close to zero. Two outstanding weights were given to the vector operations followed by the memory transactions. This is expected

since the CPU vector consumes most of the energy [7], followed by the memory, where the memory transactions involve interacting with the DMA controllers and DRAM, causing significant dynamic power drain. This also matches with findings from other authors [39] who have studied the effect of memory movements. The intercept compresses the entire system's static power, including fans, control hardware, and unused hardware such as the NIC and secondary storage. On the Intel architecture, the σ changes the behaviour on the instructions, having a more linear behaviour similar to σ_2. When setting the σ function for all instruction types, the root mean squared error (RMSE) reduces from 14 Watts (with 3.2% of central relative error) to 11 Watts (2.8% error) and improves the predictions. On the AMD architecture, the σ is preserved as explained in [7], getting 9.69 Watts of RMSE (less than 2% error). It results in a hardware dependency of the σ function on the CPU micro-architecture.



(a) Usage only

(b) Usage + Instructions

Figure 2.3: Model prediction performances of different power consumption estimation models contrasted against the true value from IPMI (PSU) measurements. All models consider the CPU usage logarithmic.

Figure 2.3a and 2.3b illustrate the models using only the natural logarithm of the CPU usage as a parameter (setting $h_0^p = 1, h_k^p = 0; k \neq 0$, *Usage only*) and the proposed model using the parameters from Table 2.4 (*Usage + Instructions*). The model with CPU usage variance, as seen in Figure 2.3a, is illustrated through same-color vertical point alignment, indicating a lack of parameters (underfitting) within the model. Figure 2.3b illustrates how the proposed model performs in terms of prediction when involving the histogram of instructions in it. It demonstrates that the proposed model outperforms the CPU usage-only model, removing the variance among experiments and suggesting a better fit. The proposed model is robust in most experiments but tends to have more variance when performing the **dgemm** experiment, which involves both memory and arithmetic, suggesting the presence of disturbances in the measurements performed by the tool and

affecting the model prediction. Further research on new prediction techniques is required to minimise the disturbances.

Multi-Process Analysis

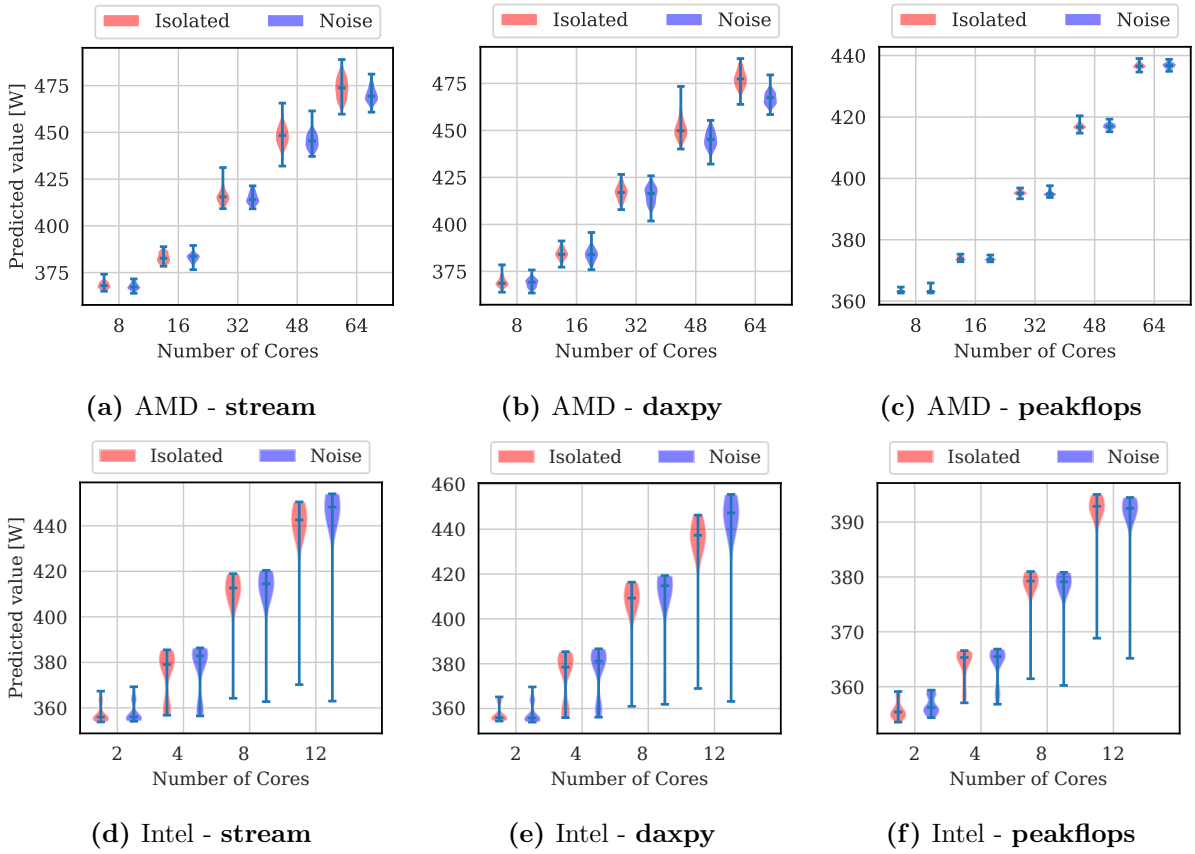
One novel contribution of this work is a method and model to quantify energy consumed by a single process without execution isolation. This section presents how the proposed model and tool perform when measuring three different workloads of interest (PoI):

- peakflops_avx_fma (peakflops)
- daxpy_mem_avx_fma (daxpy)
- stream_mem_avx_fma (stream)

with different numbers of cores and fixed workload size, while running other unrelated processes on the computing nodes specified in Table 2.2, maintaining the same experimental conditions: fixing the frequency and the fan speed. For the unrelated processes, the **dgemm** computation is included, occupying $1/4^{\text{th}}$ of the total machine capability. Regarding the model and parameters, the model from equation (2.18) with the parameters PSU-based NNLS of Table 2.4 is used, given their performance and generalisation over all the system power consumption.

Figure 2.4 shows the assessment of the tool and the model when predicting the power consumption of the PoIs (organised in columns) on the two compute machines under study. The figures illustrate two execution cases: 1) the PoI running in isolation and 2) the workload running with the **dgemm** to break the isolation (noise), all while scaling the PoI in the number of cores. The medians for a fixed number of cores and within the same PoI are close, giving the same power prediction for isolated and noise cases. In the AMD-based node, the **daxpy** has the maximum deviations when executing with 48 cores and 64 cores, with 15.4 and 20.7 Watts, corresponding to a relative error of 4.6% and 4.4% with respect to the isolated measurement, respectively. The best scores were achieved by the **peakflops** workload with a maximum deviation of 0.6 Watts. The key difference among the experiments lies in memory access. **daxpy** and **stream** have more memory accesses than the **peakflops**, which is a pure computational workload.

In the Intel-based node, the distributions are larger, but the behaviour is preserved within the experiment pairs, which implies that the behaviour of the noise environment comes from the measurements and not from the model. The medians preserve the alignment, suggesting that the predictions from the noise experiments are close to the isolation counterpart. Likewise, **daxpy** remains the most deviated as the number of cores increases, with a maximum deviation of 9.67 Watts (2.2% of relative error) when running at 12 cores.



(a) AMD - **stream** (b) AMD - **daxpy** (c) AMD - **peakflops**

(d) Intel - **stream** (e) Intel - **daxpy** (f) Intel - **peakflops**

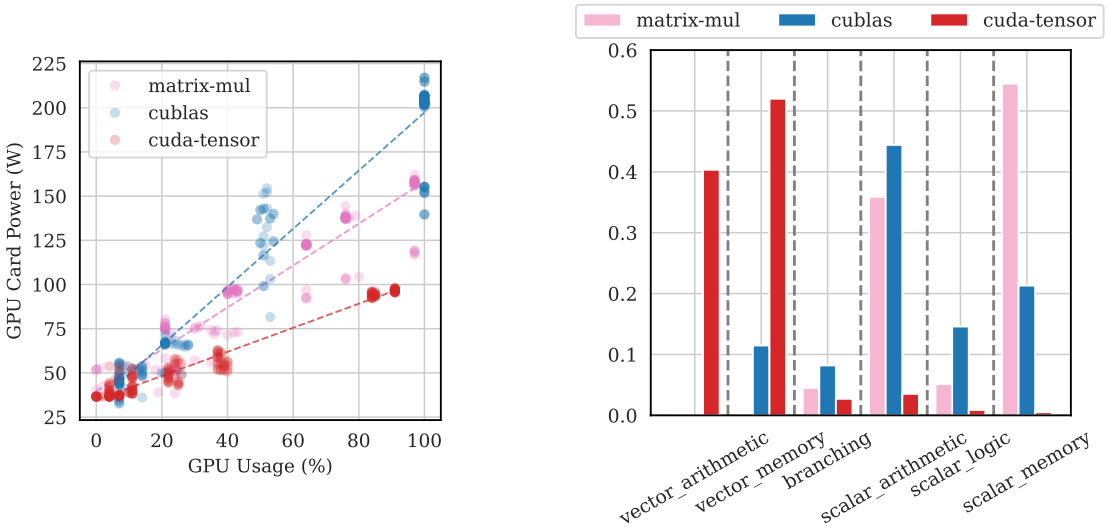
Figure 2.4: Assessment of the prediction in the absence and presence of noise for different levels of parallelism (number of cores).

2.4.3 GPU Model Results

Behavioural Instruction Analysis

Similarly to the CPU case, this work performs a series of experiments that stress different GPU instructions. For this case, different implementations of a $4096 \times 4096 \times 4096$ matrix multiplication were explored, with emphasis on memory instructions, vector instructions (Tensor Cores from NVIDIA [60]) and an optimised algorithm in CUBLAS [57]. Figure 2.5 shows the behaviour of the experiments with respect to their power consumption and the histogram of the instructions executed by each implementation. While performing the same matrix multiplication operation workload, it is possible to observe that each experiment uses different instructions to perform the same task and consumes power differently. The CUBLAS implementation has more scalar arithmetic, is the most power-consuming and less predictable, suggesting hidden features that need exploration. On the other hand, the Matrix Multiplication is a hand-crafted kernel that stands as a middle point in power consumption but has the most memory transactions. The implementation based on CUDA Tensor Cores is the most energy efficient, as it uses the Tensor Cores (a special execution unit for SIMD operations) and has the least power consumption.

To determine the nature of σ, we can perform a variable analysis to see how the product



(a) GPU Power Consumption

(b) Histogram of Operation Types

Figure 2.5: Experiments behaviour when scaling the GPU occupation and the instructions. On the left, the GPU card power consumption is reported as the usage scales, whereas on the right, the histogram of the instructions illustrates the nature of the experiment.

between the probability of the instruction k and the GPU usage w is related to the power consumption. From Figure 2.5a, the variables involved in the analysis, such as the utilisation and the instruction affectation, exhibit linear behaviour, suggesting that the product between the instruction and the GPU usage has a proportional effect on the power consumption. Therefore, we can assume that σ for the GPU is:

$$\sigma\big(h_k, w_p\big) = h_k^p w_p \tag{2.19}$$

This leads to the following power consumption model for a granular GPU utilisation:

$$P_{\text{GPU},t} = \sum_{p=1}^{N_p} \sum_{k=1}^{S} \gamma_k h_k^p w_p + \hat{P}_{\text{GPU,other},t} \tag{2.20}$$

where the power consumption for a single process running on the GPU can be expressed as:

$$P_{\text{p},t} = \sum_{k=1}^{S} \gamma_k h_k^p w_p \tag{2.21}$$

which is useful for a linear regression study to determine the parameters γ_k and the intercept $\hat{P}_{\text{GPU,other},t}$ to complete the model. A relevant side note to highlight is that, if the power measurement during the linear regression is accounted at the card level, the γ_k includes the impact of executing a process on the GPU compute units and the video RAM included in the GPU.

Table 2.5: Parameters Estimated using Least Squares Linear Regression for models using the GPU model from (2.20) as prediction reference.

Parameter Name	GPU Power Model
Intercept ($\hat{P}_{\text{static},t}$)	34.9818
Weight of Scalar Arithmetic (γ_{sa})	276.1728
Weight of Scalar Memory (γ_{sm})	33.0339
Weight of Scalar Logic (γ_{sl})	108.412
Weight of Vector Arithmetic (γ_{va})	4.9488
Weight of Vector Memory (γ_{vm})	102.3084
Weight of Vector Logic (γ_{vl})	0.0000
Weight of Branch (γ_{b})	0.00000
Weight of Jumps (γ_{j})	0.00000

Standalone Model Fitting

Similar to the experiments for extracting parameters in the CPU model, the GPU model estimation used the hardware from Table 2.3 and also required fixing the clock, fan speeds, and power profiles to minimise the variations due to the workload and thermal pressure. Moreover, a dataset created from the latter hardware is utilised, taking 50 samples per experiment run (number of CUDA threads).

Table 2.5 shows the parameters obtained after the Least Squares Linear Regression for the models described by Equation (2.20). The intercept gives the static power of the GPU card, which is 34.98 Watts, matching the ground truth of the idle measurements. Furthermore, the scalar, involving the CUDA cores and memory transactions, is the most relevant instruction. The vector arithmetic seems less relevant given that the Tensor Cores are more optimised than the CUDA cores [60]. It is also consistent with the observations from Figure 2.5a, suggesting the hardware optimisations in the GPU under study. Likewise, the vector memory loads are more relevant than the scalar version because they can spend more cycles filling different registers.

Figure 2.6 shows the fitted model for the GPU. The GPU model has an RMSE of 12.3 Watts, which corresponds to a 9.7% of relative error with respect to the central point of the dataset. Besides, the figure shows that most of the points concentrate on the bisection line, representing the model's optimal line. This suggests that the outliers play a role in affecting the overall model performance. The outliers are generated due to the runtime metrics, influenced mainly by the sampling time of the kernel execution, the power accounting tool and the profiler used to take the measurements. A deep analysis of these effects is left for future work.

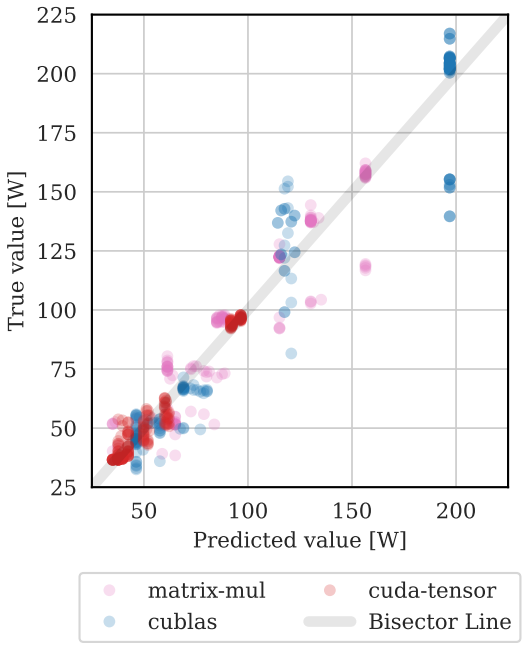


Figure 2.6: GPU model performance in power consumption prediction. The GPU model is fitted using least-squares linear regression from Table 2.5.

2.5 Final Remarks

The CPU model results show that the model and tool are robust against noise, keeping a low gap between the noise and no-noise measurements, and the law of preservation of energy (from equation (2.13)) is achieved with our proposed model and parameters.

The experimental results showed outstanding performance, with the CPU model achieving a relative error of 1.93% and the GPU model achieving a relative error of 9.7%. These results highlight that the tool and the methodology are promising in fine-grained energy accounting, contributing to the future of HPC, where the directions point to sharing computational resources. Variables such as frequency and fan speed still need to be studied to make our tool more robust and complete. Moreover, extending the work to other CPU and GPU architectures and removing the assumptions made during the analysis poses a new analysis of the impact of the instructions on the energy consumption. In future work, it is possible to mitigate this through feature-based learning techniques such as deep learning. Also, a comparison with the EAR framework is recommended to validate on Intel platforms.

The project is currently being used by Dualistic S.R.L to prepare energy-aware scheduling algorithms in datacentres digital twins. *EfiMon* can be found open-source in the repository cited below.

Related Publications:

- L. G. Leon-Vega, N. Tosato, and S. Cozzini, "A Comprehensive Analysis of Process Energy Consumption on Multi-Socket Systems with GPUs," *Latin American High Performance Computing Conference (CARLA)*, 2024. DOI: 10.1007/978-3-031-80084-9_4

- L. G. Leon-Vega, N. Tosato, and S. Cozzini, "EfiMon: A Process Analyser for Granular Energy Prediction," *Latin American High Performance Computing Conference (CARLA)*, 2024. DOI: 10.1007/978-3-031-80084-9_8

Repository:

- L. G. León-Vega, N. Tosato, and S. Cozzini, *EfiMon: An Instruction-Driven Process Energy Consumption Analyser for Multi-Socket Computers*, version v0.1.0, Apr. 2024. DOI: 10.5281/zenodo.11072569

Chapter 3

Automating the Design of FPGA-based AI Accelerators

Beyond addressing energy quantification challenges in supercomputing, sustainable computing must also confront hardware efficiency issues. AI inference has become the dominant workload in High-Performance Computing (HPC), spanning both high-end embedded systems and large-scale cloud infrastructures. Today, General-Purpose Graphics Processing Units (GPGPUs) are widely used to accelerate AI workloads due to their programmability, performance, and extensive development tool support. However, their significant energy consumption [61] raises an essential question: *Are GPGPUs truly the most suitable architecture for AI workloads?*

The scale and complexity of state-of-the-art AI models, often comprising billions of parameters, impose considerable challenges for storage, memory access, and computation. To alleviate these demands, researchers have developed techniques such as parameter quantisation, model pruning, and other forms of complexity reduction [62].

Despite these advances, achieving optimal execution on traditional hardware remains challenging. For example, some neural network layers perform adequately with 14-bit fixed-point precision, while others require 20-bit fixed-point representation. Conventional hardware platforms typically offer only standard 16-bit or 32-bit data widths for these cases, leading to inefficiencies in resource utilisation and increased energy consumption.

Field-Programmable Gate Arrays (FPGAs) offer a compelling alternative. Their reconfigurability allows the hardware to be tailored specifically to the precision requirements of AI workloads. Following the previous example, instead of using standard 16-bit resources for 14-bit calculations, FPGAs enable the creation of arithmetic units with exactly the required bit width, reducing both energy usage and resource consumption.

This chapter presents the design and development of a library of generic accelerators capable of optimising common deep learning inference tasks—such as matrix operations, convolutions, and vector mappings. These accelerators offer configurable operand sizes,

data types, and operator structures, facilitating further research in operator optimisation and approximate computing. While the primary focus is *edge inference*, the approaches discussed can also be scaled to supercomputing applications.

3.1 Related Work

FPGAs are becoming an option for AI inference due to their energy efficiency. The essential advantage of these platforms over the other alternatives is their high performance and flexibility, offering a better trade-off between computation power and energy consumption [61]. In the case of the ZYNQ 7000 family, they have a maximum power consumption of 5W and the Xilinx K60, up to 7.6W with better performance than the Jetson Nano [63], [64]. Beyond, FPGAs have been demonstrated to be superior in AI inference to GPUs. The FlightLLM project leverages LLM inference to cloud-grade (or high-end) FPGAs, with 1.6× more performance and more than 3× more energy efficiency than the NVIDIA A100 [65]. However, the code is not available online and closed for optimisation.

Most solutions target high-end[1] FPGAs such as the Xilinx UltraScale+, Alveo, Kintex, Virtex and Versal, and their equivalents in other vendors. It leaves aside proper support for low-end FPGAs like the Artix-7, which consumes up to 1.6[2]W [66]. Xilinx offers the Vitis AI software development kit (SDK), and Intel provides OpenVINO and OneAPI [67]. Most of them employ closed-source generic IP Cores such as the Deep Learning Processor (DPU) from Xilinx [68], closing the opportunity for tuning according to the error resiliency of the application.

In the research community, attempts have been made to synthesise models such as LeNet-5, VGG16, and YOLO using high-level synthesis (HLS). Some alternatives synthesise entire models on Vivado HLS, performing the whole inference within the FPGA, resulting in a 4.7× speedup on a Zybo 100 MHz with respect to an Intel Core i5 4590 3.3 GHz in single-precision floating-point (FP32) without applying any quantisation [69]. Besides, there are case studies of CNN optimisation on FPGAs. Some of these implementations manipulate the numerical representation, using fixed-point representation, increasing the performance compared to FP32. An object detector based on the You Only Look Once (YOLO) model managed to run at 1.88 TOP/s at 200 MHz, favouring performance at a low clock speed. In this case, the FPGA acceleration offers from 17.6 to 29.4 times less energy consumption than CPU/GPU processing for equivalent workloads. It shows the potential of FPGAs in this field in terms of speed and computation efficiency [70]. Other research addresses powerful and cloud-based FPGAs, like Cloud-DNN, that optimise the execution in a streamlined fashion to deal with latency and take advantage of these FPGAs [71]. On the other hand, Caffeine shows the case where the inference task can be leveraged to middle-end FPGAs (Xilinx Kintex) in synergy with software optimisations in Caffe (unpopular nowadays) to achieve better performance [72].

[1] FPGAs with more than 100K logic cells

[2] According to Xilinx Vivado 2018.2 and Xilinx Power Estimator

FPGA DL inference can be optimised by approximating and refactoring Multiply-Accumulate (MAC) operations, and making the accelerators lighter and less power-consuming. MAC operations spend 99% of the energy in Deep Neural Networks [73], making MAC operators a suitable target for DNNs optimisation. Moreover, there are other fronts of optimisation. A survey presented in [74] highlights the importance of FPGA research for Internet-of-Things (IoT) since GPUs and CPUs are unsuitable because of energy consumption constraints. It mentions classical contributions to model compression and approximation, taking advantage of the inaccurate nature of neural networks. Some novel proposals include non-linear quantisations [75], accelerator-aware pruning [76], [77], sparse algebra acceleration, and knowledge distillation [78]. Automated analysis of solutions is also crucial when determining optimal design solutions. DNNExplorer proposes a framework to explore the design space that involves the complexity of dealing with high-dimensional spaces when having granularity in the design parameters [79].

There are successful frameworks within the scientific community that significantly impact the deep learning acceleration of FPGAs, from edge to the cloud. At CERN, the HLS4ML framework [80] implements a workflow that receives the compressed model in either TensorFlow or PyTorch. Then, it converts to HLS and creates a project that allows the user to tune the design: establishing optimisation goals, and adjusting the numerical precision and the reuse factor (how much the execution units can be recycled to save resources). However, it lacks granularity in the design, limiting the possibility of implementing designs in low-end FPGAs, and has approximation limitations, restricting the integration of approximate arithmetic units. FINN is another framework that accelerates Binary Neural Networks (BNNs). It focuses on mapping and quantising neural networks to BNNs, testing the performance by using the roofline model [81] as an exemplification metric for comparing the performance of each solution. A second version of the FINN framework extends the support for CNNs [82], allowing for generating high-performance accelerators, taking into account the dataflow style (similar to HLS4ML) and running the inference aided by the PYNQ framework [83]. FINN proposes operation-cost functions to get the near-optimal implementations, performing an automated design exploration. Similar to HLS4ML. However, it also lacks granularity in the design and focuses on cloud FPGAs, leaving aside low-end FPGAs. It also limits the integration of approximate arithmetic units.

Both solutions were conceived for high-end FPGAs and cloud facilities, but lack granularity in the design, limiting the possibility of implementing designs in low-end FPGAs. FINN addresses this issue by splitting the accelerator into IPs per layer. However, the optimisations are still difficult for research in optimisation tasks and approximate computing.

In terms of custom and manually-designed solutions, most research shows no chance to adapt the Processing Elements (PE) according to the FPGA resources or comply with resource constraints [84]–[87]. In such work, the data width is fixed to a certain value as well as the operand dimensions.

Another challenge is the integration of approximate units. DNNExplorer, Caffeine, FINN and *HLS4ML* support reduced-precision arithmetic, but their execution elements are still exact and do not integrate any approximation beyond quantisations.

Hence, both frameworks are powerful and popular in the acceleration of FPGAs. They still have some open challenges and opportunities to improve. For instance:

- Granularity: both approaches implement the units of the whole model into the FPGA. There is no chance to distribute the computation between the FPGA and a host CPU for a hybrid execution.
- Approximate computing: both frameworks are exact in their computations. It opens the chance to experiment with approximate computing techniques.
- Model size: the designs produced by both frameworks can still be unsuitable for low-end FPGAs like the Artix-7 in their smallest configurations, and it is more difficult in the context of LLMs. The combination of the first two challenges can lead to an improvement in this issue.
- Opening to other applications: both frameworks are highly tailored to DLI and restrict their usage to other applications, such as Linear Algebra. A more generic approach could satisfy the DLI and other fields as well.

3.2 Generic Accelerator Design

The most widely used frameworks for offloading AI inference to FPGAs, such as HLS4ML and FINN, typically adopt a full-model mapping approach. In this method, the entire neural network is implemented on the FPGA using a dataflow architecture, where each layer is translated into a dedicated accelerator module. These modules are interconnected to mirror the model's computational graph, often resulting in the number of hardware accelerators matching the number of layers. Additionally, model weights are frequently hard-coded into the accelerators, which maximises throughput for small models. However, this approach limits the flexibility of the design, as accelerators cannot be reused for different operations or layers. Furthermore, each accelerator is generated from a specialised template based on the characteristics of the corresponding layer. Consequently, optimisation strategies are uniformly applied across all accelerators performing the same type of operation, making aggressive and fine-grained optimisations impractical.

This full-model mapping approach and template-based design reveal clear opportunities for improvement. Transitioning from a model-centric to an operation-driven synthesis methodology, replacing rigid templates with more general-purpose accelerator designs, would offer several advantages. Specifically, such a shift would: 1) enable support for larger and more complex models, 2) improve hardware resource reuse across different

operations and layers, and 3) facilitate more targeted, operation-level optimisations, including the integration of approximate computing techniques.

This work tackles these opportunities by proposing a granular and configurable accelerator with the following characteristics:

1. **Replaceable data type**: integer, fixed-point, and floating-point.
2. **Data width**: arbitrary precision of the aforementioned data types.
3. **Operands' size**: following a Tensor Core-like approach [60], where a fixed matrix size and larger matrices are processed through tiling.
4. **Function operators**: allowing to select between exact or approximate function operators, i.e. adders, multipliers, exponential, hyperbolic tangent.
5. **Number of parallel Processing Elements**: for independent same-operation processing.
6. **Replaceable operation algorithm**: the implementation of how an operation is executed can be replaced by an alternative way. For instance, replacing the standard matrix multiplication algorithm by a Strassen-Winograd alternative.

The accelerator that adheres to these characteristics shall be modular and template-based to meet most of them. The execution units of these accelerators shall comply with an Application Programming Interface (API) to ensure the replaceability of the operation algorithm.

This work proposes a template accelerators, presented in Figures 3.1, with three hierarchy levels in a top-bottom approach:

- Math Operator Level (Bottom)
- Processing Element Level
- Accelerator Level (Top)

From top to bottom, the *first level* involves accelerator templates that contain two hardware structures: static blocks (including register blocks) and template or dynamic blocks. The static blocks have a fixed implementation, meaning the communication protocol and register implementations remain fixed but still optimisable at the HLS directive level. However, they can scale to meet the target data width and the number of PEs. This meets the characteristics 2 and 5.

On the other hand, the template blocks allow the replacement of their implementation, which reaches the *second level* of hierarchy. For instance, it is possible to use a classical matrix multiplication-addition PE (using a three-for-loop implementation) or a more

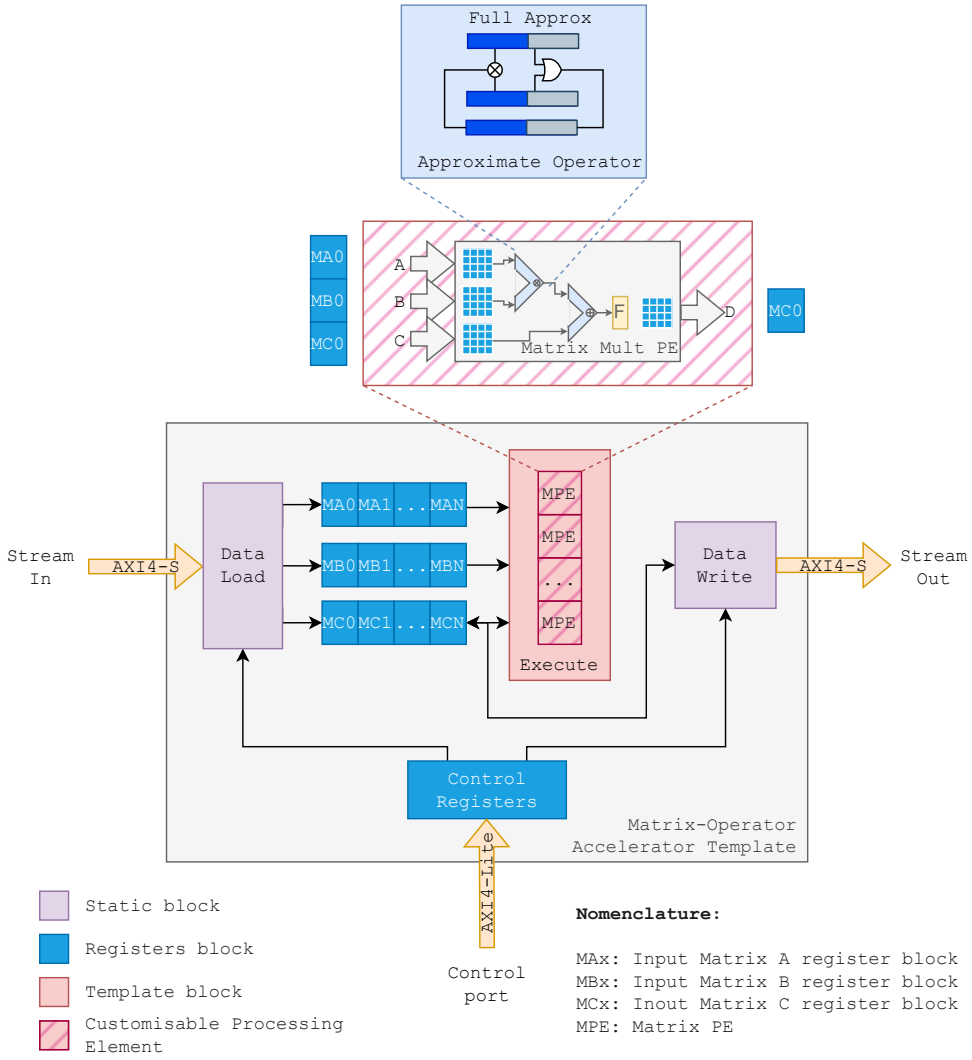


Figure 3.1: Block diagram of the proposed architecture for matrix operations, composed of two types of blocks: statics (including registers) and templates. The template blocks are adjustable in implementation, whereas the statics can only adjust the operand sizes and word lengths.

optimised PE, like a Strassen Matrix-Multiplication (explained later in this chapter), meeting the characteristic 6, allowing an easy algorithm switch according to the design requirements in delay, latency, and resources.

To enforce the other characteristics, it is crucial to define the requirements for the PE design, which are:

- **Easy data type swapping**: the PEs shall allow the support of multiple data types, i.e., floating-point, fixed-point, or integer. This support can be done through a C++ template parameter.

- **Configurable operand size**: varying the input/output size makes it possible to adjust the PE size and define how many PEs can fit into an accelerator.
- **Stackable blocks**: the accelerator shall support placing one or more PEs seamlessly. It means that adding more PEs shall not be problematic or require additional developer logic.
- **Open for approximations at the operator level**: approximate adders, multipliers and non-linear functions can be supported by the PE. Thus, by modifying a parameter, it will be possible to replace an exact adder with an approximate one.
- **Replaceable**: different implementations of the PE shall have a standard interface allowing changing the PE at the accelerator's level.

These requirements allow the accomplishment of the accelerator's characteristics at the second level of hierarchy. From the requirements mentioned before, the *Stackable blocks* and *Replaceable* permit the abstraction to the top level, particularly through the characteristics 5 and 6. The other requirements help to meet the other characteristics.

For the *third level* of hierarchy, the math operators can be abstracted to be replaceable and modifiable, allowing the characteristic 4, and inherently, the characteristics 1 and 2. At this level, it is possible to replace any math operation, allowing different implementations and approximations. For instance, it is possible to replace approximate versions of the exponential function, shifting from the HLS-builtin exponential function, or LUT or Taylor-based approximations, which are addressed in Chapter 4.

From another perspective, Figure 3.1 can be conceptualised in an oversimplified code through Algorithm 1. In lines 2 - 20), it defines how the developer can tune the implementation of the template part. The accelerator parameters, such as the data type (line 9), operator (line 10), data width (line 9), operand size (4) and PE implementation (13), are built into the accelerator; nevertheless, these parameters can be defined through C definitions for compile-time configurability. Apart from configuring the architecture, the accelerator provides the functionality to load data from a host and write the results back. Both functions, represented through `load_data` and `write_data` (lines 17 and 19), are implementation-fixed, but they can scale according to the data type `D` and operand sizes `M` and `N`. The overall implementation can be optimised through a directives file leveraging Vivado/Vitis HLS capabilities.

The template part, conceptualised as `execute` (lines 23-27), integrates the number `N` of PEs, and the PE implementation, `E` (as a `SimpleMatrixMultiplyAdd`, illustrating the configurability of the template block. Deepening into this function, `execute` of Algorithm 1 shows a possible implementation. In this case, `Vector` (line 26) is a wrapper that replicates the engine `E`, proposed later in this work.

Algorithm 1 Example of built-in accelerator definition

```
void accel(Stream &in, Stream &out) {
  /* M: matrix size of PE, N: number PEs */
  constexpr int M = 2;
  constexpr int N = 2;

  /* D: fixed-point data with width = 14
   * and int width = 6 */
  using D = FxP<14, 6>;
  using Op = MathExact;

  /* E: PE implementation */
  using E = SimpleMatrixMultiplyAdd<D, M, M, Op>;

  /* Other definitions ... */

  load_data<D, M * N, M>(in, A, B, ...);
  execute<E, N, D, M * N, M>(A, B, C, ...);
  write_data<D, M * N, M>(out, C, ...);
}

/* Example of definition of the template block */
template<class E, int N, typename D, int R, int C>
void execute(D A[R][C], D B[R][C], D C[R][C], ...) {
   using Vector = ama::hw::ParallelMatrixOperator;
   Vector<N, N, E>::Execute(A, B, C, D);
}
```

3.3 Flexible Accelerator Library

The generic accelerator idea was proposed in the previous section, and the final accelerator implementation is subject to various degrees of freedom: data types, data widths, number of PEs, math operators, and algorithms. Every combination leads to different results in terms of trade-offs, particularly in resource consumption, area and latency. Knowing the effects of every configuration is crucial for implementing a proper accelerator in terms of the restrictions and design goals.

This work proposes the Flexible Accelerator Library (FAL), which includes the accelerator templates, PEs and math cores, representing the three levels of hierarchy explained in the previous section. Moreover, it is equipped with a tool for Design Space Exploration, which allows the study of the impact of different configurations on the final accelerator implementation.

Figure 3.2 shows the proposed automatic accelerator generation workflow. The process

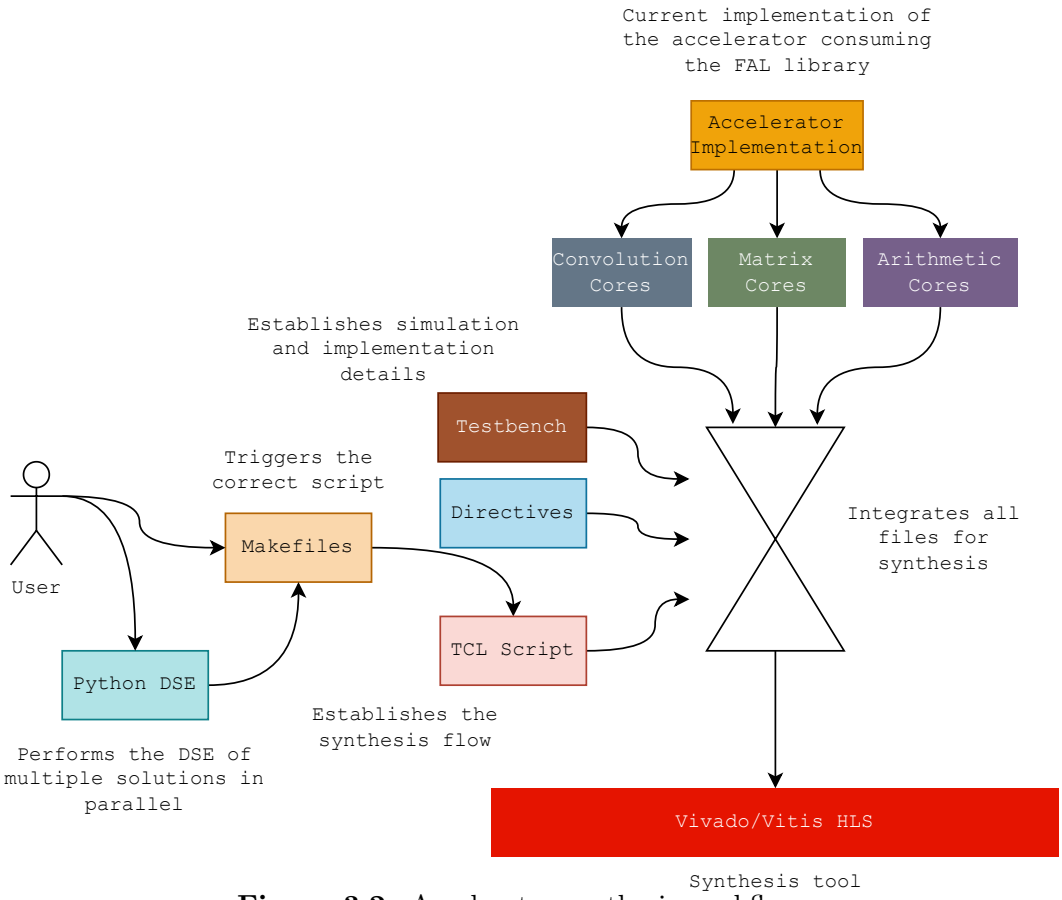


Figure 3.2: Accelerator synthesis workflow.

starts with either a design space exploration (DSE) process (through a parallel Python script) or introducing the specific configuration using a Makefile. In both cases, a generalisation of the Algorithm 1 is used, where the configurations are available at compile time through C pre-processor macros, allowing a Tcl script to introduce them in the generation process.

The workflow receives the accelerator template and the PE designs for matrix, convolution or arithmetic operations. Apart from the implementations, it also receives a testbench to verify and validate the current implementation (design plus parameters) and captures data on the associated errors. This allows the introduction of directives to optimise the target implementation. At the end of the process, the workflow produces an IP core and a report with the resource utilisation (look-up tables (LUT), flip-flops (FF), block RAM (BRAM), and digital signal processing (DSP) blocks), latency, and error of the implementation. For this work, the solutions have been optimised using a pipeline-like implementation.

With the generic accelerator proposal, this work proposes a series of generic or template PEs to perform different matrix operations (Matrix Cores) and convolutions (Convolution) and assess different implementations. The generic accelerator architecture and PEs also receive the arithmetic operators (Arithmetic Cores) addressed in Chapter 4.

The scope of this contribution will be limited to the matrix multiplication, emphasising different algorithm implementations and exact math. Other operations are addressed indirectly in the following chapters from a different perspective. More complex approximate computing approaches are addressed in Chapter 4. Likewise, the convolution accelerator had been previously tackled in previous research work [88].

The Standard Matrix Multiplication (a.k.a. naïve algorithm) is the most popular implementation to compute matrix multiplications on FPGAs. It has been adopted by state-of-the-art (SOTA) frameworks such as FINN [82] and HLS4ML [89]. It uses three loops: the two outer ones iterate over the output rows and columns, whereas the inner loop performs the vector dot-product of the selected row of the left-hand matrix and the selected column of the right-hand matrix, leading to a $\mathcal{O}csdv < f < (n^3)$ time complexity.

Algorithm 2 Standard Matrix Multiplication

```
1: for i = 1 to M do
2:     for j = 1 to N do
3:         c_ij ← 0
4:         for k = 1 to K do
5:             c_ij ← c_ij + a_ik × b_kj
6:         end for
7:     end for
8: end for
```

Algorithm 2 illustrates the Standard Matrix Multiplication to perform $\mathbf{C} = \mathbf{AB}$ and the construction of the three loops, where M is the number of rows of the output, N is the number of columns and K is the number of columns and rows of the first and second input matrices, such that $\mathbf{A}$ has a size of $M \times K$, $\mathbf{B}$ $K \times N$ and $\mathbf{C}$ $M \times N$. Considering the case of a 2×2 matrix-matrix multiplication, unrolling the operation leads to the operations expressed in (3.1). This particular case leads to *eight products and four additions*.

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} \tag{3.1}$$

Strassen is an alternative algorithm that reduces the number of multiplications in exchange for more additions, assuming that adders are faster than multipliers [90]. It reduces the computation complexity of the matrix multiplication to $\mathcal{O}(n^{2.8074})$ with a degradation of the numerical stability [91].

The Strassen Matrix Multiplication for 2×2 matrices starts with the computation of the following factors:

$$\begin{aligned} m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}); \\ m_2 &= (a_{21} + a_{22})b_{11}; \\ m_3 &= a_{11}(b_{12} - b_{22}); \end{aligned}$$

$$
\begin{aligned}
m_4 &= a_{22}(b_{21} - b_{11});\\
m_5 &= (a_{11} + a_{12})b_{22};\\
m_6 &= (a_{21} - a_{11})(b_{11} + b_{12});\\
m_7 &= (a_{12} - a_{22})(b_{21} + b_{22});
\end{aligned}
$$

Then, the algorithm proposes a series of additions and subtractions to find the output elements, as illustrated in (3.2). In this case, the Strassen Matrix Multiplication requires *seven multiplications (one less than the Standard) and eighteen additions (fourteen more than the Standard).*

$$
\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 - m_2 + m_3 + m_6 \end{bmatrix} \tag{3.2}
$$

The *Winograd form* of the Strassen algorithm is another alternative that, similar to Strassen, uses *seven multiplications* but reduces the additions to *fifteen*, resulting in possible resource savings compared to Strassen. The complexity is still the same as in Strassen [91].

Like Strassen, Winograd formulates the matrix multiplication as follows in (3.3). Within this formulation, it is possible to reuse the results from $a_{00}b_{00}$, $w + u$ or $w + v$, $a_{10} + a_{11}$, $b_{01} - b_{00}$ and the last one plus b_{11}. In this way, the number of multiplications is *seven* and the additions *fifteen*.

$$
\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & w + v + (a_{00} + a_{01} - a_{10} - a_{11})b_{11} \\ w + u + a_{11}(b_{10} + b_{01} - b_{00} - b_{11}) & w + u + v \end{bmatrix} \tag{3.3}
$$

with:

$$
\begin{aligned}
w &= a_{00}b_{00} + (a_{10} + a_{11} - a_{00})(b_{00} + b_{11} - b_{01});\\
u &= (a_{10} - a_{00})(b_{01} - b_{11});\\
v &= (a_{10} + a_{11})(b_{01} - b_{00});
\end{aligned}
$$

The most popular implementation for dealing with matrices of greater sizes is the divide-and-conquer, decomposing the matrices until getting 2×2 matrices and performing a submatrix multiplication.

This work studies three PE implementations: 1) standard matrix multiplication (using Equation (3.1)), 2) Strassen multiplication (using Equation (3.2)) and, 3) Winograd multiplication (using Equation (3.3)); for 2×2 matrix multiplication implemented in HLS-compliant C++, where each element from the input/output matrices can be accessed in parallel and are operated simultaneously, attached to the principles stated above regarding the arithmetic operators.

Regarding the accelerator, it is able to adapt its implementation to cope with the target arbitrary precision based on the width of the bus (in this case, fixed to 64-bit for illustration purposes and subject to convenience). For this case, the implementation splits the bus data width into pairs of matrix elements, given an arbitrary precision. The number of PEs will be determined by squaring the number of pairs that fit into a transmission packet (bus width).

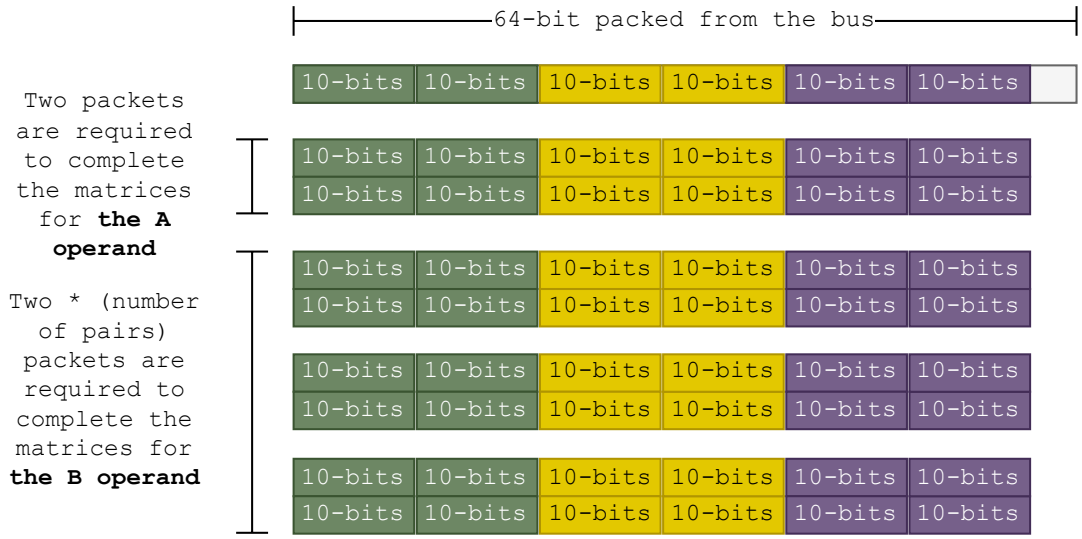


Figure 3.3: Operand transmission according to the bus width. Given a matrix multiplication $\mathbf{C} = \mathbf{AB}$, $\mathbf{A}$ takes two packets from the bus, whereas the $\mathbf{B}$ uses $2 \times n_{\text{pairs}}$ packets from the bus.

Figure 3.3 provides an example of 10-bit operands' transmission through a 64-bit bus. The first operand $\mathbf{A}$ takes two packets to represent the rows of the PE. The pairs of data over the bus consider the number of columns of the PE. Thus, every packet will feed a complete row of all PEs. In the case of the second operand $\mathbf{B}$, apart from considering filling a complete row, it also takes into account the number of columns of $\mathbf{A}$ for data reuse. It leads to a square matrix of PEs of the size $n_{\text{PE}} = \left(\frac{B_{\text{bus}}}{2B_{\text{data}}}\right)^2$, where B_{bus} is the width of the bus and B_{data} is the data arbitrary precision in bits. Lower B_{data} and bigger B_{bus} lead to more PE units and greater parallelism.

3.4 Experimental Results

The evaluation of the FAL involves a series of steps, which implies analysing the PEs, the accelerator and studying the impact on the accuracy of a DL Model. Moreover, given the context of the FPGAs, the metrics of interest are:

- Numerical Error and Instabilities
- Resource Utilisation

- Latency
- Deep Learning Performance

This section will address the evaluation of FAL in three major blocks: 1) the PE analysis, 2) the accelerator analysis, and 3) the DL benchmark.

3.4.1 Metrics and Figures of Merit

Resource Utilisation

When evaluating digital architectures on FPGAs, four primary resources are commonly analysed:

- **Look-Up Table Cells (LUTs)** are fundamental logic elements used to implement combinational logic; their count reflects the overall complexity and area of the logic design.
- **Flip-Flops (FFs)** are used for sequential logic, such as storing state information or synchronising data across clock domains, and contribute to pipeline depth and shortening timing behaviour.
- **Digital Signal Processing Cells (DSP)** blocks are specialised hardware units optimised for arithmetic operations like multiplication and accumulation, making them critical in architectures involving heavy numerical computation.
- **Block Random Access Memories (BRAMs)** provide dedicated on-chip memory and are essential for efficiently buffering data or implementing large memory arrays.

The utilisation of these resources directly impacts the **overall area, performance, and power consumption** of an FPGA implementation.

An equally important performance metric is **latency**, which refers to the number of clock cycles an architecture takes to produce an output from the time an input is applied. Latency depends on both the depth of the pipeline. The **delay**, typically measured as the critical path or maximum operating frequency, is influenced by how these resources are interconnected and the depth of the logic paths, making resource balance crucial for achieving optimal performance. It determines the maximum clock frequency at which a design can run.

Error evaluation

For error evaluation, it is possible to have a notion of the numerical stability of the architecture. By computing the operation in a single-precision floating point, it is possible

to get the golden data. Then, the matrix is quantised to the custom datatype and transferred to the PE. The resulting elements are compared against the gold data as

$$\epsilon_{ij}^{(p)} = \frac{||\hat{x}_{ij}^{(p)} - x'_{ij}||}{2\mathbf{sup}||\mathbf{X}'||} \tag{3.4}$$

where the error is computed as the L1-distance between the result $\hat{\mathbf{X}}$ from the p-th solution and the golden data scaled $\mathbf{X}'$, normalised against the *supremum* of the vector space X'. In this case, the value of the supremum is 2 given that the dynamic range is $]-1,1[$. Equation (3.4) will be used to compute the mean error $\mathbb{E}[\epsilon^{(p)}]$, the standard deviation $\sigma^{(p)}$, and the histograms for the error distribution illustration.

Deep Learning Metrics

Top-1 Accuracy (this work will treat this as simply *accuracy*) is defined as the percentage of input samples for which the classifier's most confident prediction (i.e., the class with the highest predicted probability) matches the true class label [92]. Formally, given a set of N input samples $\{(x_i,y_i)\}_{i=1}^{N}$, where x_i is the input and y_i is the true label, and a classifier that outputs a ranked list of class scores or probabilities $f(x_i)$, the top-1 accuracy is given by:

$$\text{Top-1 Accuracy} = \frac{1}{N}\sum_{i=1}^{N} 1(\hat{y}_i = y_i) \tag{3.5}$$

where $\hat{y}_i = \text{argmax} f(x_i)$ is the predicted class with the highest score, and $1(\cdot)$ is the indicator function, which returns one if the expression is true and zero otherwise.

On the other hand, the Receiver Operating Characteristic (ROC) curve is a graphical representation used to evaluate the performance of a binary classifier as its discrimination threshold is varied [93]. The curve plots the True-Positive Rate (TPR) against the False-Positive Rate (FPR) at different threshold settings. TPR is the ratio of correctly predicted positive instances to all actual positives, while FPR is the ratio of incorrectly predicted positives to all actual negatives.

The Area Under the ROC Curve (AUC) quantifies the overall ability of the model to discriminate between the positive and negative classes, independent of any specific threshold. An AUC of 1.0 indicates perfect classification, while an AUC of 0.5 implies performance equivalent to random guessing. The ROC-AUC is especially useful in imbalanced datasets where accuracy alone may be misleading.

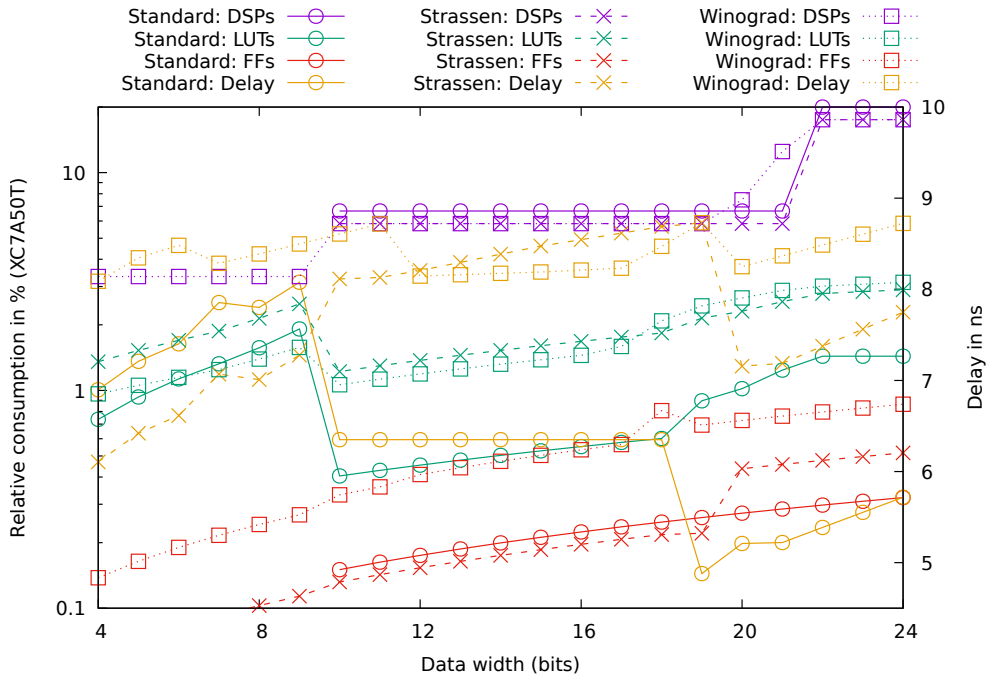
3.4.2 Processing Elements

Previously, the Standard, Strassen, and Winograd implementations of matrix multiplication were explained. These implementations are tied to the PE code, which represents the algorithm that is replaceable at the accelerator level. The following analysis will focus on comparing the trade-offs of the proposed algorithms, highlighting the differences in terms of resource utilisation, delay (minimum clock period) and error. This will help in the selection of the algorithms according to design requirements. Also, all resource analysis targets the XC7A50T, a low-end FPGA, running at two different clock frequencies.

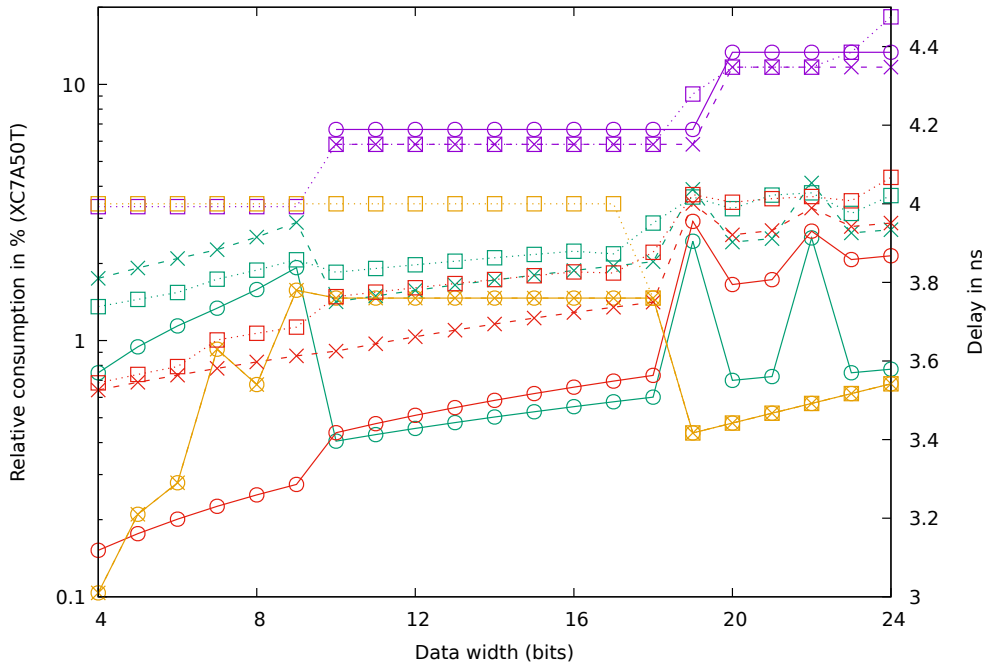
Figure 3.4 shows the resource consumption and the delay of the critical path of each PE when adjusting the numerical precision from 4 to 24 bits when using 100 and 250 MHz clock speeds. As the numerical precision (data width) increases, there are three consumption zones. The *first zone* starts from 4 to 9 bits, the *second* from 10 to 18 bits and the *third* one from 19 to 24 bits. Moreover, the three algorithms seem to have their strengths and weaknesses in every consumption zone, suggesting no clear winner in terms of resources across all numerical precisions.

At 100 MHz, the Standard Multiplication's *first consumption zone* wins in terms of Flip-Flop consumption, having a null consumption and partially wins in terms of LUT consumption, where Winograd consumes fewer resources from 7 to 9 bits. Winograd consumes at least 6.01% fewer LUTs compared to the Standard and 28.96% fewer than Strassen. On the other hand, Strassen is the clear winner in delay, having at least 9.90% and 14.36% less delay than the Standard and Winograd, respectively. In the *second consumption zone*, Strassen and Winograd are the best in DSP consumption, saving 12.5%. On the other hand, Strassen consumes at least 12.24% and 60.19% less Flip-Flops than the Standard and the Winograd PEs, respectively, leading to an FF-friendly implementation. In delay, the Standard multiplication shows superiority with a constant 6.35ns implementation, which is at least 27% better than the other two algorithms. The *third consumption zone*, the results are similar to the second zone in terms of DSP, with savings of 12.5%, and in delay, being 35.52% better than the other two algorithms. Thus, the Standard multiplication can be considered the best implementation if DSP is not a priority.

At 250 MHz, there are still three consumption zones. In the *first zone*, the delay between Standard and Strassen is approximately the same, suggesting no benefits in using either Strassen or Winograd, leaving the Standard as the best. In the *second zone*, Strassen and Winograd are superior in DSP consumption (12.5% less); however, Strassen keeps the same latency as the Standard, being better than Winograd, but in exchange for other resources, like 250% more LUTs and 108% more Flip-Flops. Therefore, if DSPs are a concern in the second zone, the best is Strassen; otherwise, the Standard Matrix Multiplication. For the *third zone*, the delay for the three implementations is the same. Nevertheless, the DSP consumption is superior in the Strassen and Winograd implementations: Winograd consumes 37.5% more DSPs than the Standard implementation at 19 and 24 bits, respectively. On the other hand, Strassen and Winograd consume around 250% and 56.35% more LUTs and Flip-Flops than the Standard implementation, respectively, making it



(a) Matrix Multiplication Implementations at 100 MHz



(b) Matrix Multiplication Implementations at 250 MHz

Figure 3.4: Resource consumption and signal delay of the Processing Elements under study. The numerical data precision varies from 4 to 24 bits. The delay is given in nanoseconds, and the consumption is relative to the XC7A50T FPGA running at 100 and 250 MHz. Each PE is implemented using exact arithmetic units and the same optimisations that achieve the best trade-off.

superior in LUT and Flip-Flop consumption. Between Strassen and Winograd, the first consumes fewer LUT and FF than the second, making it more resource-friendly.

Table 3.1: Application domains for the implementations studied during this work from the results presented in Figure 3.4. The contents of the table summarise the targets of each implementation. The target clock speed is 100 MHz

Bits	Standard	Strassen	Winograd
4-9	FF consumption	Delay FF consumption	LUT consumption
10-18	Delay LUT consumption	FF consumption DSP consumption	DSP consumption
19-24	Delay LUT consumption FF consumption	DSP consumption	DSP consumption

Table 3.2: Application domains for the implementations studied during this work from the results presented in Figure 3.4. The contents of the table summarise the targets of each implementation. The target clock speed is 250 MHz

Bits	Standard	Strassen	Winograd
4-9	Best		
10-18	Delay LUT consumption	Delay DSP consumption	DSP consumption
19-24	Delay LUT consumption FF consumption	Delay DSP consumption	DSP consumption

Tables 3.1 and 3.2 summarise the application domains of the three algorithms based on their strengths and weaknesses after analysing Figure 3.4. Based on consumption results, most of the consumption goes to the DSP cells with 11.67%. It suggests that, in low-end FPGAs, adding many PEs can deplete the resources available in the chip, which is the most critical resource in this type of FPGAs. Thus, Strassen and Winograd are attractive since they consume more from other resources to free DSPs, as expected when the number of multiplications is reduced. However, the scenarios may change in high-end FPGAs, given the proportion of resources, placement and routing results and optimisation techniques.

Moreover, the results from Winograd are interesting: there are fewer additions, but it can be considered the worst implementation in most scenarios regarding latency, FF consumption and DSPs. Winograd degradations are caused by the depth of the operations, given that, to reduce the number of additions, it is mandatory to reuse some of the computations, aggregating delay and enlarging the critical path. This requires the addition of flip-flops and extra clock cycles to stabilise the signal and meet the timing constraints. Table 3.3 shows the depths of each implementation and the clock cycles required for each consumption zone. Unlike the Standard multiplication, Strassen and Winograd add depth, but Winograd adds extra depth due to the reuse of results. This

leads to increased clocks, adding one clock cycle compared to the Standard and Strassen algorithms. Strassen, instead, keeps the depth relatively low enough to avoid increasing the clock cycles but the increase on the number of additions translates into an increased LUT consumption.

Table 3.3: Depth of the operations for computing every output and the clock cycles required per PE when targeting a 100 MHz synthesis. c_{ij} represents the i-th and j-th element of the output matrix and z_k is the consumption zone.

	Depth of the Operators				**Clock Cycles**		
Implementation	c_{00}	c_{01}	c_{10}	c_{11}	z_1	z_2	z_3
Standard	2	2	2	2	1	1	1
Strassen	4	3	3	4	1	1	2
Winograd	3	6	6	6	2	2	3

For error analysis, the error is determined by measuring the distance between the hardware and software results for every element and constructing the histogram of error distances. Thus, it is possible to profile the mean error rate of the PE for a given numerical precision.

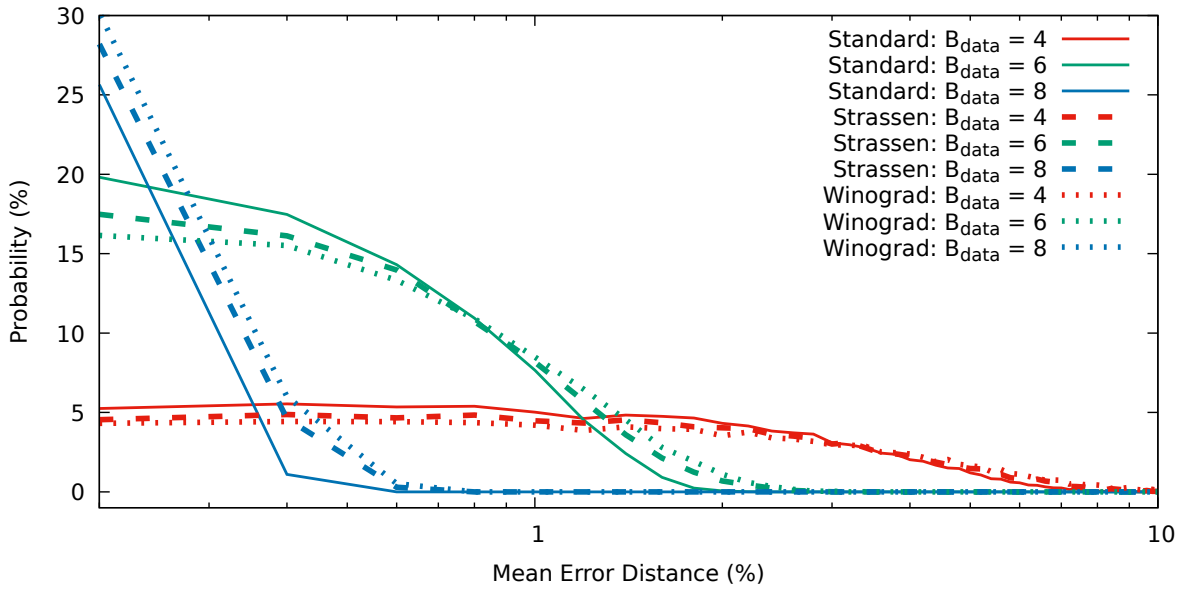


Figure 3.5: Probability distribution of the Mean Error Distance (MED) when varying the numerical data precision in bits (B_{data}).

Figure 3.5 shows the error analysis for the three matrix multiplication algorithms through an error histogram, using the lowest three data widths. The Y axis represents the probability that a mean error distance (using Equation 3.4) between the gold data and the experimental data occurs. In this case, a good sign is that the histogram (or probability distribution) has a lower variance (concentrated towards zero). In all cases, the non-standard matrix multiplications have a greater error variance, suggesting that the operators' refactoring degrades the quality of the results (QoR).

3.4.3 Accelerator

After analysing the matrix multiplication algorithm, the next step is to see the impact after plugging them into the accelerator template. During this process, the PE will be replicated automatically if the number is not fixed, according to the bus width of the stream port (AXI-4 ports) and the data width, to maximise the parallelism. The following analysis will unveil the effects of integrating Strassen and Standard Matrix Multiplication PE designs into the accelerator template (depicted in Figure 3.1) that allows swapping the implementations for a fair comparison. The behaviour of Winograd will be homologous to that of Strassen and will be excluded from the analysis.

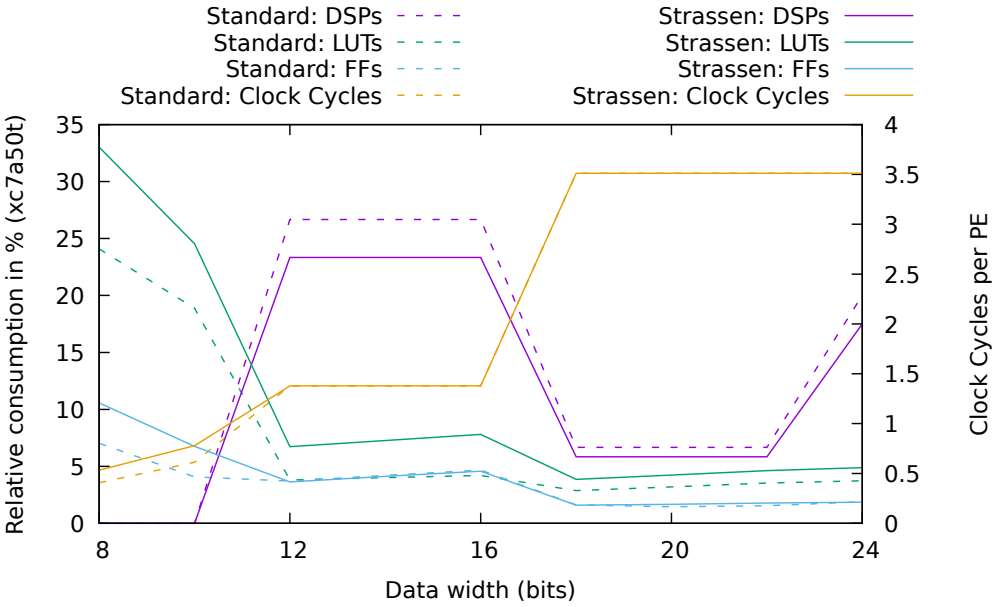


Figure 3.6: Resource consumption and mean clock cycles of an accelerator that integrates the PEs under study. The data precision varies from 8 to 24 bits. The mean clock cycles required to execute the PE (with communication overlap) are computed by dividing the total number of clock cycles of 10000 execute stage runs (Figure 3.1) by the number of runs. The resource consumption is relative to the XC7A50T FPGA, running at 100 MHz.

In this study, the accelerator's execute stage is completed by the square number of pairs that fit into a bus packet (for maximum utilisation), increasing the parallelism within the accelerator (Figure 3.3). After integrating the Standard and Strassen PE into the template accelerator, Figure 3.6 shows the resource consumption and mean clock cycles for a single run. Overall, the mean clock cycles for both configurations are the same, suggesting no speed benefit except when the data width is less than 12 bits, where the Standard Multiplication is better. Regarding resources, the behaviour observed when analysing the PE resource consumption is preserved (Figure 3.4) but with more noticeable changes. For $B_{\text{data}} \leq 10$, the Strassen-based accelerator occupies up to 8.93% of the whole FPGA more than its Standard counterpart. For $B_{\text{data}} > 10$, the Strassen accelerator has better resource footprint, with a total reduction of 3.33% for $12 \leq B_{\text{data}} < 18$,

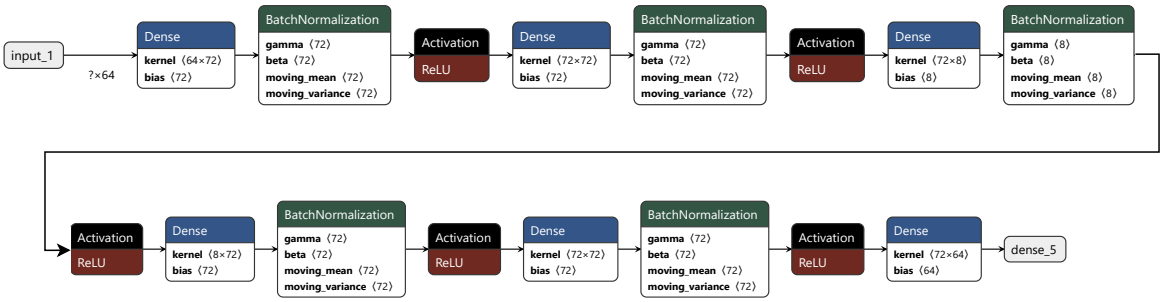


Figure 3.7: Anomaly Detection (AD) Model based on a Multi-Layer Perceptron Encoder.

0.83% for $18 \leq B_{\text{data}} < 24$ and 2.5% for $B_{\text{data}} = 24$. Contrasting with the DSP-specific reductions between the two implementations and ignoring the FPGA total resources, Strassen consumes 12.5% fewer resources than the Standard for $B_{\text{data}} > 12$. However, as also observed in Figure 3.4, the LUT consumption is still considerable. For $12 \leq B_{\text{data}} < 18$, the Strassen accelerator consumes 46.19% more LUTs with respect to the Standard accelerator, and for $18 \leq B_{\text{data}} < 24$, 24.57%. Therefore, Strassen-based accelerators are more suitable for numerical data precisions with more than 10 bits when the goal is to save DSP cells. Moreover, the LUT consumption gap is reduced after 18 bits, resulting in a more promising scenario. Once again, the compromise of LUTs and FFs is still present and might cause energy consumption issues.

Furthermore, more recent studies use Strassen matrix multiplication in high-end FPGAs, highlighting that it matches or exceeds baselines [94]. Hence, studying matrix multiplications in FPGA remains an open topic for future research.

3.4.4 Deep Learning

One of the most significant practical applications of matrix multiplication accelerators is Deep Learning Inference (DLI), as dense layers—where matrix multiplications dominate—are among the most frequently used operations in this domain. To evaluate the effectiveness of the proposed Processing Elements (PEs), experiments were conducted using a custom simulator, *AxC Executer* (introduced in Chapter 4), developed in C++. The evaluation followed a similar experimental setup to that used by HLS4ML and FINN [95], targeting the Anomaly Detection (AD) task from the MLPerf benchmark [96]. Specifically, the ToyADMOS dataset [97] was employed, using the same pre-processing and post-processing stages as in prior work.

Anomaly Detection is an unsupervised learning task to distinguish between normal and anomalous signals. For this study, the autoencoder model proposed in [95] was adopted, which is based on fully connected layers (FCL), incorporates Batch Normalisation, and uses the Rectified Linear Unit (ReLU) activation function. The architecture of this model is illustrated in Figure 3.7.

For this evaluation, the model from the MLPerf benchmark was synthesised for the Xilinx

XC7A50T FPGA, using various configurations of the reuse factor across all layers to minimise resource consumption and enable deployment on the entire FPGA fabric. In HLS4ML, the reuse factor defines the number of operations a single execution unit can perform sequentially. For example, a reuse factor 128 means that one unit handles 128 operations, promoting hardware reuse and reducing overall resource utilisation.

Table 3.4 presents the results obtained for three different reuse factor settings, highlighting their impact on both latency and hardware resource consumption. When the maximum reuse factor achievable for the model was applied, the latency reached 3696 clock cycles to produce a result, while the Look-Up Table (LUT) utilisation exceeded 100%. This makes the hardware solution from HLS4ML infeasible for this scenario.

Although this model is relatively small compared to others, such as LeNet-5 [98], SqueezeNet [99], and MobileNet [100], HLS4ML is unable to provide viable hardware implementations for low-end FPGAs, even after extensive optimisation and fine-tuning. This limitation also persists for larger models and high-end FPGAs, reinforcing the necessity for a generic and flexible accelerator architecture, as proposed in this work. A detailed discussion of this approach is provided in Chapter 6.

Table 3.4: HLS4ML statistics when implementing the model on the Xilinx XC7A50T FPGA. The resource consumption refers to the available resources of the FPGA and the reuse factor applies to all layers, where 512 is the maximum supported by the model.

Reuse Factor	Latency (clocks)	Maximum Frequency (MHz)	LUT	FF	DSP	BRAM
128	1148	158.73	340.64%	85.71%	127.50%	34.00%
256	1998	158.73	391.48%	81.20%	65.83%	18.00%
512	3696	158.73	363.16%	81.44%	35.00%	19.33%

Focusing on the greatest reuse factor and the dense layers, Table 3.5 shows the resource utilisation and latency of all the dense layer modules that are generated as part of the hardware. All these layers utilise the same quantisation but differ in layer size, meaning that HLS4ML allocates a number of dense accelerators equal to the number of dense layers. The main idea is to have granularity in terms of the configuration of each layer and increase the output by specialising each module. However, when looking to save resources, it is not possible to reuse the module as a whole, forcing the implementation to have multiple redundant units.

For testing the model in AxC Executer, in contrast, the model was optimised using Zero-Shot Quantisation [62]. This method converts the floating-point operands to fixed-point without model-level optimisation, like fine-tuning, by just scaling and assigning a number of bits with acceptable QoR. In this case, a 20-bit fixed-point with a 12-bit integer part was used for the input and output layers (external layers), whereas a 16-bit fixed-point with a 6-bit integer was used for the hidden layers. This configuration provided more

Table 3.5: Resource utilisation and latency of all the dense modules present in the HLS4ML implementation. The resource utilisation is with respect to the Xilinx XC7A50T FPGA. All of the dense modules use the same quantisation but different sizes of inputs/outputs. The aggregation of all the modules does not fit into the FPGA.

Layer	Latency	LUT	FF	DSP	BRAM
2	581	12.37%	5.58%	0.83%	1.33%
6	582	16.13%	5.57%	0.83%	1.33%
10	582	25.48%	8.71%	7.50%	4.67%
14	582	25.97%	8.38%	6.67%	4.67%
18	582	23.37%	6.83%	7.50%	2.67%
22	519	25.48%	8.71%	7.50%	4.67%
Total	N/A	**128.79%**	**43.79%**	**30.83%**	**19.33%**

than 0.8 AUC, similar to the Keras model results using zero-shot quantisation without fine-tuning [62]. Batch Normalisation is performed as an element-wise multiplication and addition (see equation (3.6)) to get rid of the divisions and the square roots at inference time. For evaluation purposes, the mean-squared error (MSE) was computed between the input and output and averaged over each window [95].

$$\mathbf{y} = \beta + \gamma \times \frac{\mathbf{x} - \mu}{\sqrt{\sigma + \epsilon}} = \mathbf{b} + \mathbf{A}\mathbf{x} \tag{3.6}$$

where

$$\mathbf{a} = \frac{\gamma}{\sqrt{\sigma + \epsilon}};$$
$$\mathbf{b} = \beta - \mathbf{a} \times \mu;$$

Therefore, it implies that the execution of the model requires: (a) accelerators for matrix-multiplications, (b) accelerators for element-wise addition and (c) multiplication. At least one of the former types is required for each quantisation; therefore, six accelerators are required to run the AD model.

Focusing on matrix multiplication, Table 3.6 summarises the implemented accelerators for the AD model, using the three algorithms examined in this work. At first glance, it is evident that Standard Multiplication outperforms the other methods in terms of resource usage—excluding DSP blocks—while the Winograd and Strassen algorithms achieve lower overall resource consumption.

According to the results in Table 3.2, Standard Multiplication is highly recommended for matrix operations involving operands with bit widths between 19 and 24 bits. In contrast, the Strassen algorithm is better suited for operand sizes between 10 and 18 bits. Similar

trends are observed at the accelerator level in Table 3.6, where the maximum operating frequency for both Standard and Strassen algorithms is comparable, but the Winograd implementation achieves a lower maximum frequency.

In terms of DSP resource utilisation, the non-standard algorithms (Strassen and Winograd) offer savings of up to 12.5% in configurations using 16-bit operands. Specifically, Standard Multiplication requires 32 DSP cells, while the Strassen and Winograd implementations require only 28. Regarding Flip-Flop consumption, the results show that Standard Multiplication provides a 12.3% reduction, using 6001 cells compared to the 7342 cells consumed by the Strassen implementation.

Table 3.6: Resource utilisation and latency of all the matrix-multiplication accelerators from FAL to accelerate the AD model with module reuse. The resource utilisation is with respect to the Xilinx XC7A50T. The modules show two different quantisations: fixed-point with 16 bits, 6-bit integer (FxP<16, 6>), and fixed-point with 20 bits, 12-bit integer (FxP<20, 12>).

Configuration	Latency (worst case)	LUT	FF	DSP	BRAM	Maximum Frequency (MHz)
FxP<16, 6>: Standard	**391**	**7.48%**	**9.20%**	26.67%	0.00%	**247**
FxP<20,12>: Standard	**782**	**3.43%**	**2.72%**	6.67%	0.00%	**247**
FxP<16, 6>: Strassen	**391**	11.15%	11.26%	**23.33%**	0.00%	**247**
FxP<20,12>: Strassen	**782**	10.12%	3.06%	**5.83%**	0.00%	**247**
FxP<16, 6>: Winograd	475	10.61%	12.15%	**23.33%**	0.00%	235
FxP<20,12>: Winograd	950	10.11%	3.23%	6.67%	0.00%	235

Following the recommendation to use Strassen Multiplication—primarily to reduce DSP cell usage, which is a scarce resource on this FPGA—the total hardware resources required for the dense layers would amount to 6934 LUTs, 9339 Flip-Flops (FFs), and 32 DSP cells. These values represent 21.3%, 14.3%, and 29.2% of the available FPGA resources, respectively. In comparison, the HLS4ML implementation consumes 128.79% of LUTs, 43.79% of FFs, and 30.83% of DSP cells (as shown in Table 3.5). This demonstrates that the accelerators proposed in this work achieve significant resource savings: LUT usage is reduced by 83.5%, FF usage by 67.3%, and DSP usage by 5.3%. If Standard Multiplication were used instead of Strassen, the consumption of LUTs and FFs would decrease even further. However, this would result in an 8% increase in DSP cell usage compared to the HLS4ML implementation.

According to the previous numerical results, the Strassen and Winograd algorithms exhibited lower numerical stability compared to Standard Multiplication, as illustrated in Figure 3.5. This instability could potentially affect the performance of the Anomaly Detection (AD) model. To assess this impact, Figure 3.8 presents the Receiver Operating Characteristic (ROC) curves for the AD model executed using accelerators based on the studied Matrix Multiplication PEs. These results are compared against the baseline Keras model implemented with single-precision floating-point (32-bit FP) arithmetic.

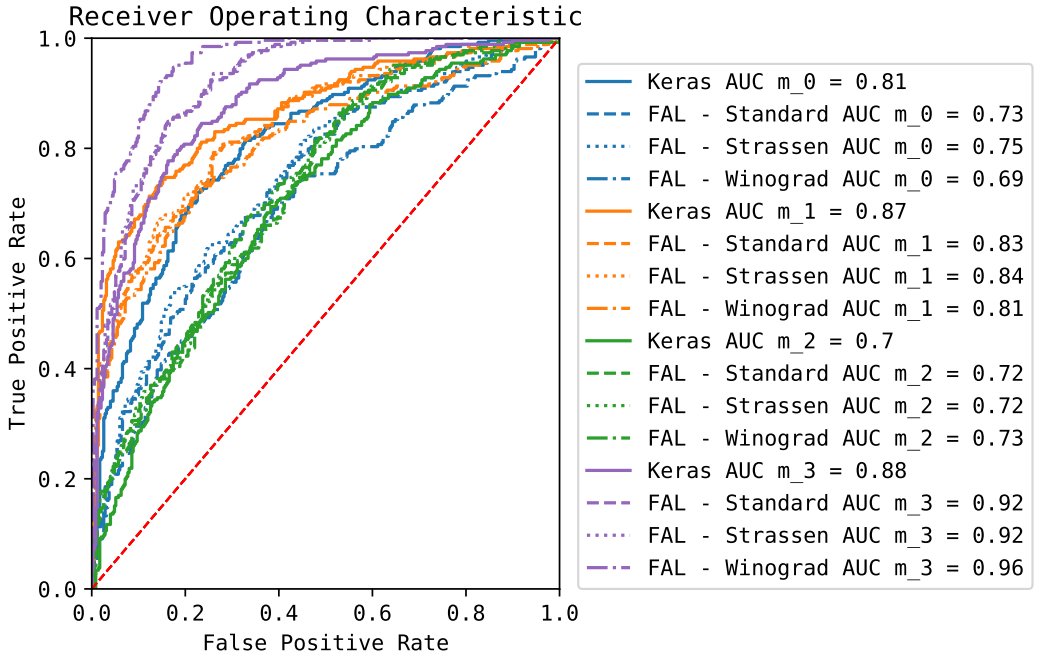


Figure 3.8: Receiver Operating Characteristic curves obtained from the test data set when evaluating the Keras model in 32-bit floating-point and the accelerators based on the Standard, Strassen and Winograd Multiplication. m_i refers to the number of the sample.

Interestingly, the numerical errors introduced by the Strassen algorithm unexpectedly improve the ROC curves and Area Under the Curve (AUC) scores across all evaluated samples when compared to the Standard Multiplication implementation. In some cases, such as samples m_2 and m_3, all three hardware implementations outperform the floating-point Keras baseline, with Winograd achieving the highest performance. The average AUC scores are as follows: Keras (32-bit FP) is 0.815, Standard Multiplication is 0.803, Strassen Multiplication is 0.808, and Winograd Multiplication is 0.796. Thus, the Strassen implementation achieves an average AUC that is only 0.69% lower than the Keras floating-point baseline, while Winograd is 1.9% lower. These results demonstrate that, despite minor reductions in numerical stability, the Strassen and Winograd implementations deliver performance levels that remain highly acceptable for many practical applications.

3.5 Final Remarks

This chapter introduced the Generic Accelerator architecture and the Flexible Accelerator Library (FAL), designed to overcome key limitations in FPGA-based deep learning inference. Current state-of-the-art frameworks often suffer from limited design granularity, difficulties handling large models during hardware mapping, and a lack of support for aggressive approximate computing optimisations. By addressing these issues, the proposed architecture not only improves resource utilisation but also extends the applicability of

FPGA accelerators to workloads beyond deep learning inference.

The architecture is structured across three hierarchical levels: the top level, which defines the overall accelerator; the intermediate level, responsible for algorithmic design through Processing Elements (PEs); and the bottom level, which focuses on the mathematical core. This layered design enables flexibility in algorithm selection, data type configuration, numerical precision, parallelism, and computational accuracy. To validate the proposed approach, matrix multiplication accelerators were implemented using three widely studied algorithms: Standard Multiplication, Strassen Multiplication, and Winograd Multiplication.

Experimental results revealed that each algorithm presents distinct trade-offs, particularly when varying operand bit-widths. These findings offer designers a range of options to balance resource constraints and performance objectives according to specific application requirements. While the use of non-standard multiplication algorithms introduced some numerical inaccuracies, this compromise proved beneficial in practical scenarios. In fact, the numerical imprecision led to improvements in the Area Under the Curve (AUC) metric for a real-world deep learning model, demonstrating the potential advantages of integrating approximate computing strategies. The proposed design also significantly reduced resource usage and latency for dense neural network layers compared to implementations generated by HLS4ML. However, this came at the cost of increased overall inference latency, a trade-off that was expected given the design's emphasis on hardware reuse and flexibility.

The open-source implementations resulting from this research are publicly available and can be accessed through the repository referenced below.

Related Publications:

- L. G. León-Vega, E. Salazar-Villalobos, and J. Castro-Godínez, "An Exploration of Accuracy Configurable Matrix Multiply-Addition Architectures using HLS," in *2022 IEEE 15th Dallas Circuit And System Conference (DCAS)*, 2022, pp. 1–6. DOI: 10.1109/DCAS53974.2022.9845501

- L. G. León-Vega, E. Salazar-Villalobos, A. Rodriguez-Figueroa, *et al.*, "Automatic Generation of Resource and Accuracy Configurable Processing Elements," *ACM Trans. Embed. Comput. Syst.*, Apr. 2023, ISSN: 1539-9087. DOI: 10.1145/3594540

- L. G. León-Vega and J. Castro-Godínez, "Generic Accuracy Configurable Matrix Multiplication-Addition Accelerator using HLS," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2023, pp. 171–174. DOI: 10.1109/DSN-W58399.2023.00048

- L. G. León-Vega, A. Chacón-Rodríguez, E. Salazar-Villalobos, *et al.*, "Acceleration of Fully Connected Layers on FPGA using the Strassen Matrix Multiplication," in

2023 IEEE 5th International Conference on BioInspired Processing (BIP), 2023, pp. 1–6. DOI: 10.1109/BIP60195.2023.10379257

Repository:

- E. Salazar-Villalobos, L. G. Leon-Vega, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Matrix Accelerator*, version v1.1.0, 2022. DOI: 10.5281/zenodo.6413238

Chapter 4

Introducing Approximate Computing on FPGA-based AI Accelerators

Approximate computing is a paradigm in which calculations produce results that are inaccurate but acceptable, as opposed to strictly accurate outputs. It provides low-power, small-size designs by reducing, for instance, hardware overhead [101], [102]. Applications using human senses (e.g., audio, video) can exploit the trade-off between accuracy and power savings of approximate computing [103]. One of the proposals is to modify the implementation logic of the arithmetic operators to reduce the circuit size and, thus, the power consumption and resource utilisation [101], [102], [104]–[108]. However, most of these solutions target Application-Specific Integrated Circuits (ASICs) for other kinds of devices, such as Field Programmable Gate Arrays (FPGAs).

On the other hand, HLS has helped to lower the entry barrier to FPGA development, and there are efforts in leveraging DL inference to FPGAs, in particular, with *HLS4ML* [80] and FINN [82]. They allow coarse optimisations to deal with the latency-resource consumption trade-off, allowing one to choose between two different implementations and hardware reuse. However, apart from quantisation, they do not allow other approximations like approximate mathematical units. In the previous chapter, new alternatives have been explored to allow the introduction of approximations within the accelerators through parameterised PEs for computing matrix multiplications, that leverage HLS and allow control over the size of the PE by modifying the quantisation and matrix size and admitting the introduction of custom arithmetic units for operator approximation facilitating more degrees of approximation.

This chapter studies introductory approximate arithmetic operators for HLS and complex functions like softmax (based on exponential functions). Thus, the contributions are:

- a framework to analyse approximations using a simulator implemented in C++ (called AxC Executer),
- a starting point with a basic set of accuracy configurable arithmetic units for addi-

tions and multiplications implemented using untimed C++ for HLS, parameterised in data type, data width, integer part, and the number of approximated bits, and

- the assessment of approximate computing techniques to implement the softmax function using Taylor series and interpolation methods with Look-Up Tables.

4.1 Related Work

Approximate computing is a computing paradigm that trades off accuracy for gains in performance, energy efficiency, and reduced resource usage. It is beneficial in applications where perfect accuracy is not critical, such as image and video processing, machine learning, and sensor data analysis. Instead of delivering exact results, approximate computing allows for small, controlled inaccuracies acceptable within the application context. Popular techniques include loop perforation, which skips iterations in loops to speed up execution; approximate storage, which stores data with lower precision; and arithmetic approximators, which replace parts of traditional code with approximate operators that produce similar outputs more efficiently [103].

This section covers the preliminary study for approximate arithmetic and approximate functions with Taylor Series and Interpolation.

4.1.1 Approximate Arithmetic

Approximate arithmetic has been approached from several fronts, including manual optimisation by replacing partial products in multiplications, approximating the least significant bits by forcing a value or using OR gates in adders, and including automatic generation of approximate units using evolutionary algorithms.

EvoApprox8B library [109] is one of the most popular research in approximate arithmetic synthesis, proposing the automatic generation of adders and multipliers using Catersian Genetic Programming (CGP), establishing one of the most relevant Pareto fronts in the field in terms of errors vs. area, delay, and power applied to TSMC 180 nm ASICs. SMApproxlib [110] proposes an open-source hand-crafted multipliers library for FPGAs, reaching better results for FPGAs against the use of EvoApprox8B library directly applied to FPGA logic synthesis with less than 1.5 dB of loss and 16% of Look-Up Table (LUT) savings for an 8×8 multiplier. Works based on SMApproxlib focused on LUT-aware generation of approximate partial products, improving error degradation and reducing by 12.5% the LUT consumption while also reducing latency, allowing the multipliers to run at higher frequencies [111], [112]. DeMAS is the complementary alternative to SMApproxlib for adders [113] based on LUT optimisation. The Fast and Error-Optimised Approximate Adder Unit (FAU) highlights additional resource consumption against the configurability of alternatives because of the lack of fixation [114]. Despite these solutions, they are not adjustable in numerical precision or are unavailable for C++ HLS.

Apart from the aforementioned approximate units, other research community proposals also target FPGAs. For instance, the Approximate Mirror Adder 5 (AMA5) proposes that the approximate LSBs equal the first operand LSBs [115]. InXA2 involves using the XOR gate between the two operands and the carry [116]. The median adder (MA), similar to the LSB dropping, fixes the LSB of the output to a value that reduces the Mean Error Distance (MED) in contrast to setting the lower part to zeros [117]. Within the former adders, the best power savings are the bit truncation and the MA. Nevertheless, the bit truncation is the worst-performing in terms of error. In contrast, the MA outperforms those mentioned above, having a slightly similar performance as an LSB OR-ing adder [117], [118].

Considering flexibility in terms of configuration and the exploration of solutions according to design requirements, alternatives like MA, LSB Dropping and LSB OR-ing are good candidates that reduce resource consumption while sacrificing the quality of the results. The configurability of these designs allows the control of the approximation degree, tuning the number of LSBs to sacrifice in exchange for resource consumption. In this case, LSB OR-ing and LSB Dropping are extremes that represent a certain degree of accuracy and the maximum resource consumption. In contrast, the LUT-based optimisations are the least configurable options. Despite their low resource consumption, the degree of approximation is fixed, forcing the designer to meet the accuracy of the operator instead of selecting it.

4.1.2 Softmax Implementations

Apart from the arithmetic, AI usually involves non-linear functions that add an extra complexity to the implementation. Usually, these functions are simple, like ReLU [119], but there are also complex cases like the softmax, which normalises the classification layers. Softmax implementations on FPGAs have been explored through both exact and approximate approaches. A fundamental strategy for reducing hardware complexity involves lowering numerical precision by using fixed-point arithmetic representations [120]. Among the exact designs, some works leverage the CORDIC algorithm to compute exponentials and divisions efficiently [120], [121]. On the other hand, approximate accelerators aim to reduce computational effort by simplifying multiplication and division operations [122], achieving, for instance, a 3% accuracy degradation on LeNet-5 using 16-bit fixed-point arithmetic, while consuming only 1354 FFs, 1604 LUTs, and 3 DSP slices, with a latency of 3.788 ns. A similar approach is found in [123], which uses a Taylor Series-based approximation, consuming 2229 LUTs and resulting in a 2% accuracy drop.

4.2 Introductory Approximate Arithmetic

This work proposes the implementation of adders and multipliers using HLS, allowing arbitrary-precision integer and fixed-point data types configured through C++ template parameters. The implementations can be classified into:

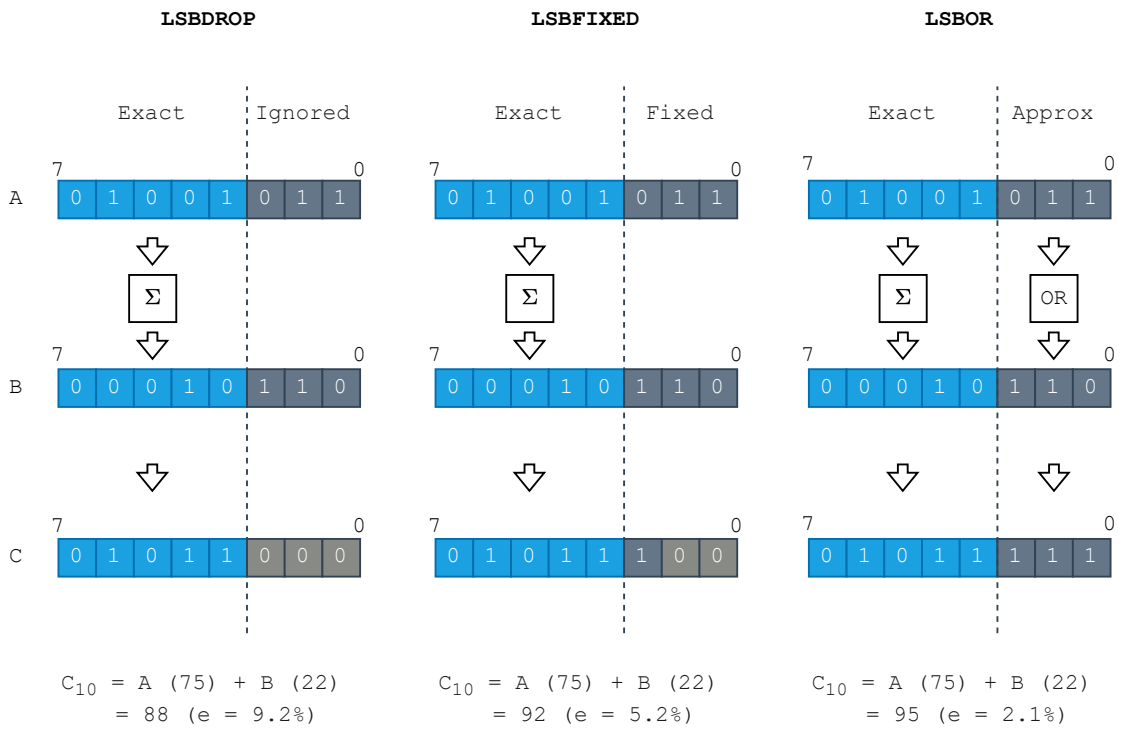


Figure 4.1: Proposed approximation techniques used, here representing the case of an addition between two 8-bit operands. The upper part of the operands is computed using exact logic, whereas the lower part uses an approximation technique.

- **Exact**: performs the exact computation using the built-in adders and multipliers.
- **Least Significant Bits (LSB) Drop**: computes addition and multiplication using bit truncation, meaning that a certain number of LSBs will be dropped and set to zero.
- **Least Significant Bits (LSB) Fixation**: computes addition and multiplication using bit truncation, meaning that a certain number of LSB will be set to a number that minimises the error distance.
- **Least Significant Bits (LSB) OR-ing**: approximately computes addition and multiplication using OR gates to compute a certain number of LSB.

Figure 4.1 illustrates the proposed approximations for the adders. It shows how the addition is performed by dropping, fixing and OR-ing the LSB. Dropping the LSBs and setting them to zero (`LSBDROP`) may lead to the biggest error distance compared to fixing the LSBs as performed in MA (`LSBFIXED`, using the central point) or OR-ing them (`LSBOR`). The multiplier follows an analogous approximation strategy to that used in the adder.

Besides the approximation, the arithmetic operators are parameterised in the data type, allowing the adoption of arbitrary precision integers and fixed-point numbers. The implementation is inspired by functors, widely used in the C++ Standard Template Li-

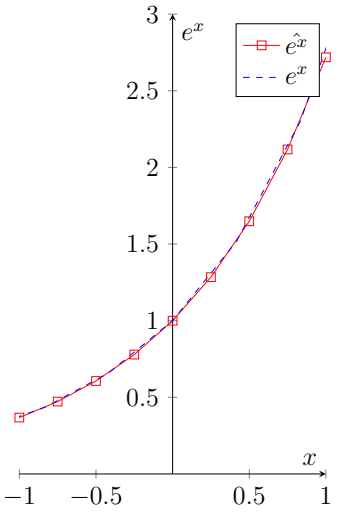


Figure 4.2: Piecewise representation by doing eight samples within the domain S and applying a linear interpolation. e^x corresponds to the exact function and $\hat{e^x}$ is the linear interpolation of e^x.

brary [124], facilitating the use of the operators in static polymorphism (as template parameters) in HLS-based PEs.

4.3 Approximate Non-Linear Functions

Approximating non-linear functions implies using domain-constraint constructs, where the function is limited to work in a specific numerical domain, with a mean relative error below a pre-defined threshold. This work studies two possible options: Taylor Series, for zero-centred domains, and LUT-based interpolation for domain-restricted applications.

A Taylor series describes a function with an infinite sum of elements expressed in terms of the target function's derivatives at a single point. For the exponential function, the Taylor series centred at $a = 0$ is

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \ldots, \forall x \in \mathbb{R} \tag{4.1}$$

where a is the point where the function's derivative is centred and it converges everywhere [125].

On the other hand, the LUT-based piecewise interpolation method consists of sampling the function at uniform, equidistant points and computing the best-fit polynomial between the points. For instance, a linear polynomial requires two points to compute, whereas a quadratic requires three points [126]. Figure 4.2 shows how a linear interpolation fits the e^x function by taking eight samples and performing linear interpolation.

The piecewise function segments can be calculated at computation time (runtime) or

recalculated at compile time. At runtime, the slope and intercept are computed as

$$m_p = \frac{y_{p_1} - y_{p_0}}{x_{p_1} - x_{p_0}}, b_p = y_{p_1} - m_p x_{p_1} \tag{4.2}$$

such that $f_p(x) = m_p x + b_p, x_{p_0} \leq x \leq x_{p_1}$, where $(x_{p_0}, y_{p_0}), (x_{p_1}, y_{p_1})$ are the points before and after the point of interest x_p, respectively. In this case, the computation of the point requires: (1) storing the points in an LUT, (2) computing the linear equations, and (3) evaluating the function at the target point. This work proposes computing the slope and the intercepts at synthesis time and storing them in LUTs to speed up the computation, shortening the path from (1) to (3).

Moreover, to avoid unwanted divisions while computing the indices of the slope-intercept pairs required for the computation, the number of points can be a power of two, such that the division becomes a bit-shift, in such a way that:

$$p = x' \gg P \implies m_p = M[p], b_p = B[p] \tag{4.3}$$

where P is the number of points (power of two), x' is the quantised value of x in fixed-point, M and B are the LUTs for the slope and intercept, respectively.

4.4 Experimental Results

During this work, a framework called *AxC Executer* has been constructed to emulate generic hardware accelerators (as presented in Chapter 3) with replaceable math operators, which is illustrated in Figure 4.3). AxC Executer is a templated C++-based inference backend for DL models. It can perform design space exploration (DSE) to choose the data and integer widths for quantisation, limiting the quantisation degradation below a given threshold and providing better room for approximation. After the quantisation, the proposed approximate operators are introduced into a C++ simulator from the AxC Executer, whose functions for computing the convolution and dense layers accept the operators through template classes, replacing all additions and multiplications with the provided ones coming from FAL. The simulation provides an overview of the accuracy of the model inference.

4.4.1 Approximate Operators Assessment

The approximate arithmetic operators evaluated in this section were synthesised using Vivado HLS 2018.2 due to stability in analysing their resource consumption in targeting an AMD 7Z020 FPGA 100 MHz.

Evaluating the performance of the operators implies the quantification of the resource consumption and the error associated with the approximations. Figure 4.4 shows the LUT

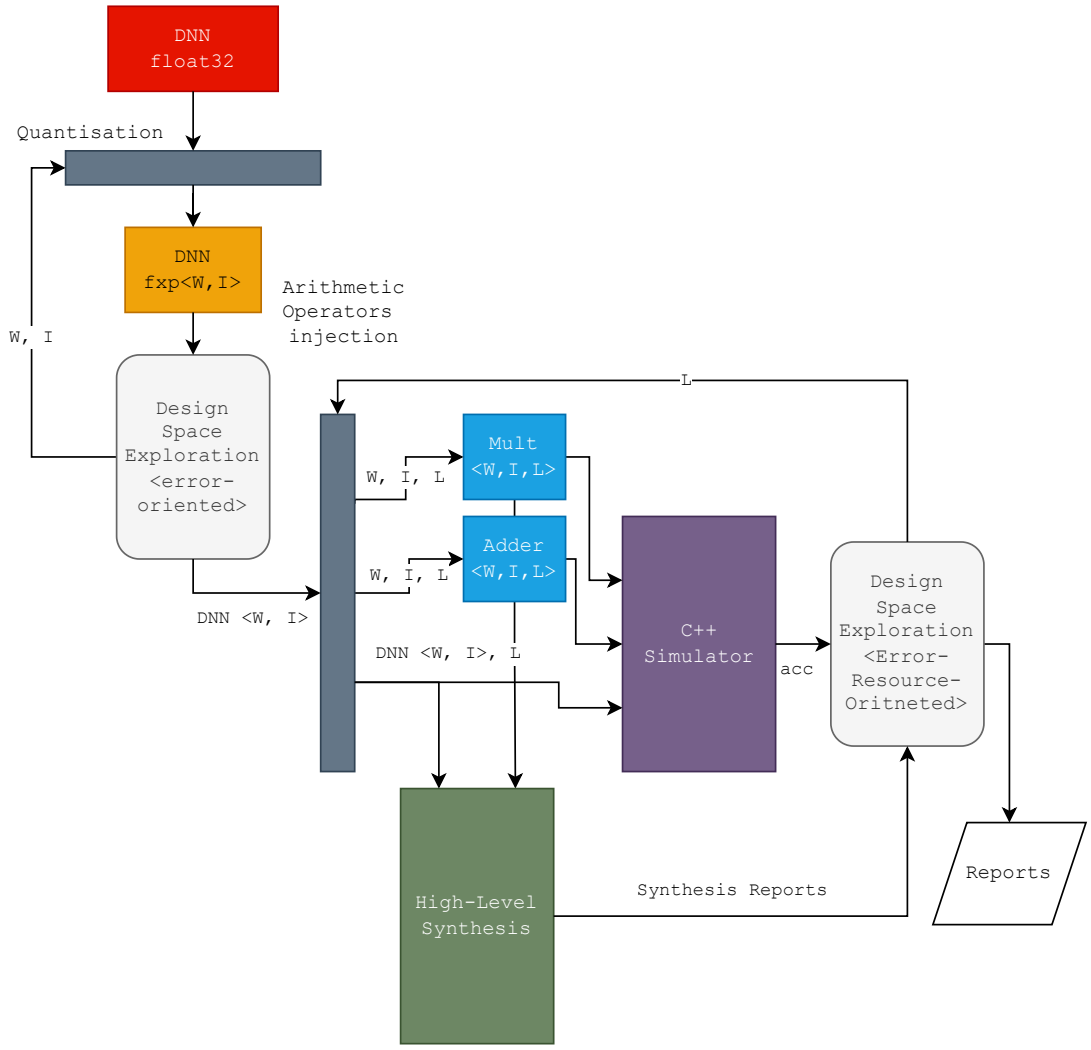
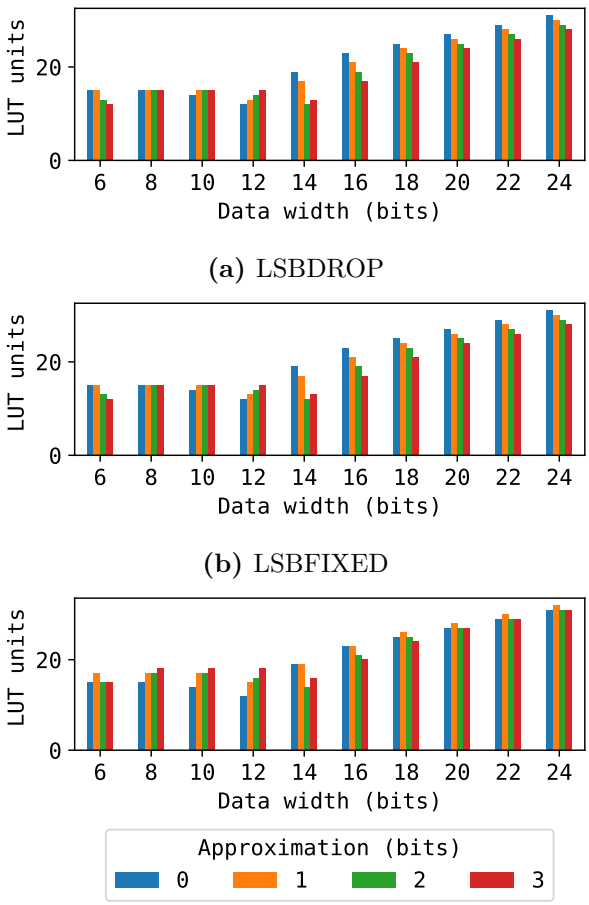


Figure 4.3: Experiment setup. The process involves quantisation of the neural network and an iterative process to determine the accuracy loss based on the configuration of the approximate adders and multipliers.

consumption of the adder with `LSBDROP`, `LSBFIXED`, and `LSBOR` approximations when varying the numerical precision and the number of approximated bits. Overall, the `LSBDROP` and `LSBFIXED` present the same behaviour, explained by the fact that the LSBs are hard-wired to a particular value. In both configurations, the approximations tend to favour resource savings except for 12-bit numerical precision, where the exact version outperforms. `LSBOR`, instead, demonstrates benefits only when having numerical precisions of more than 14 bits and more than 2-bit approximations. All configurations show a minimum consumption when having a 14-bit data width in 2-bit and 3-bit configurations. In contrast, for a 1-bit approximation, the minimum is achieved at a 12-bit data width, but it spends more resources than the exact version. In general, the most favourable



(a) LSBDROP

(b) LSBFIXED

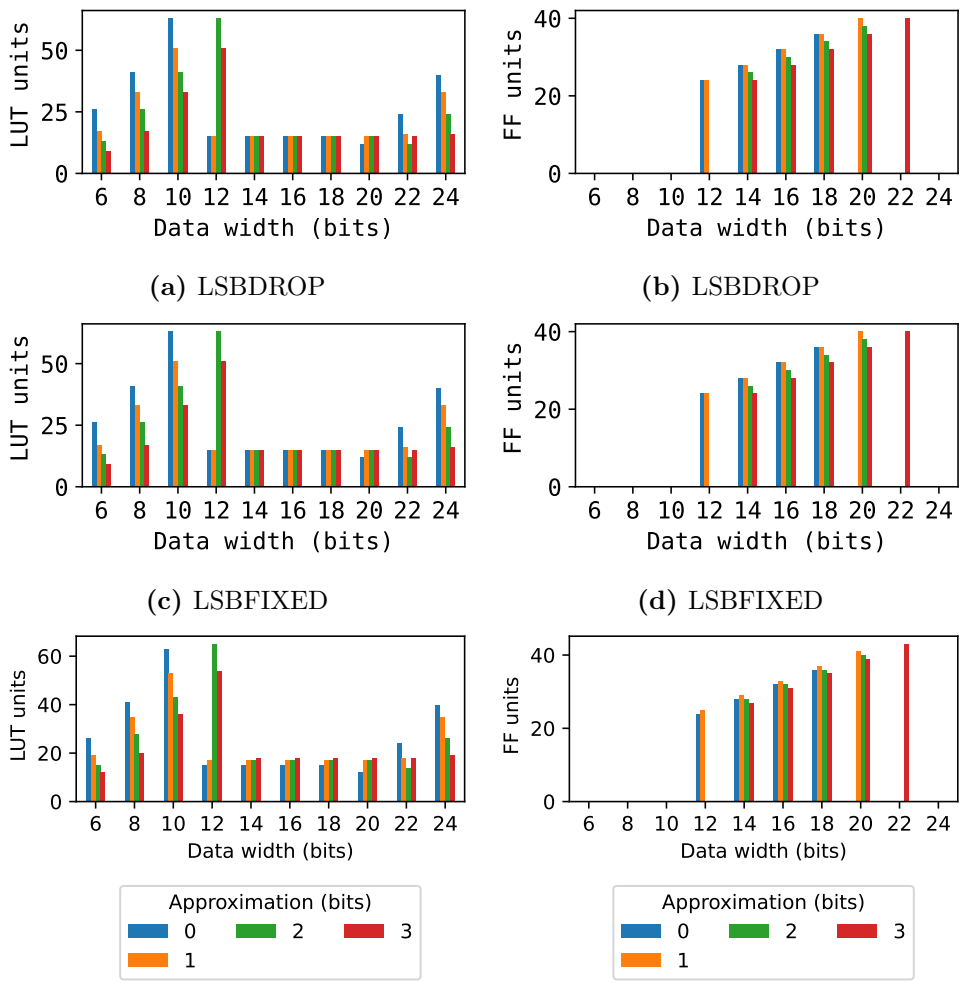
(c) LSBOR

Figure 4.4: Resource consumption of the proposed adders in terms of LUTs when increasing the data width under three different approximations.

configuration is a 14-bit data width with 2-bit approximation, with savings of 36.8% in `LSBDROP` and `LSBFIXED`, and 26.3% in `LSBOR`. Other resources like FF, DSP and BRAM are not used in the proposed adders.

Figure 4.5, similar to Figure 4.4, shows the LUT and FF consumption of the multiplier. 12-bit multipliers start consuming DSP and FF units, while the LUT consumption decreases. One of the most relevant aspects is the DSP and FF usage. Assuming that a possible goal is to reduce their consumption, the most recommended configuration is to keep the exact multiplier of the upper part (most significant bits) below 12 bits before the transition to DSPs. Another aspect is LUT consumption. A 12-bit multiplier with a 2-bit approximation equals the consumption of an exact 10-bit multiplier. In contrast to the adder, the multiplier consistently benefits from approximation, leading to quasi-linear resource consumption.

Evaluating the operators requires the computation of the mean error, which is evaluated as



(a) LSBDROP (b) LSBDROP

(c) LSBFIXED (d) LSBFIXED

(e) LSBOR (f) LSBOR

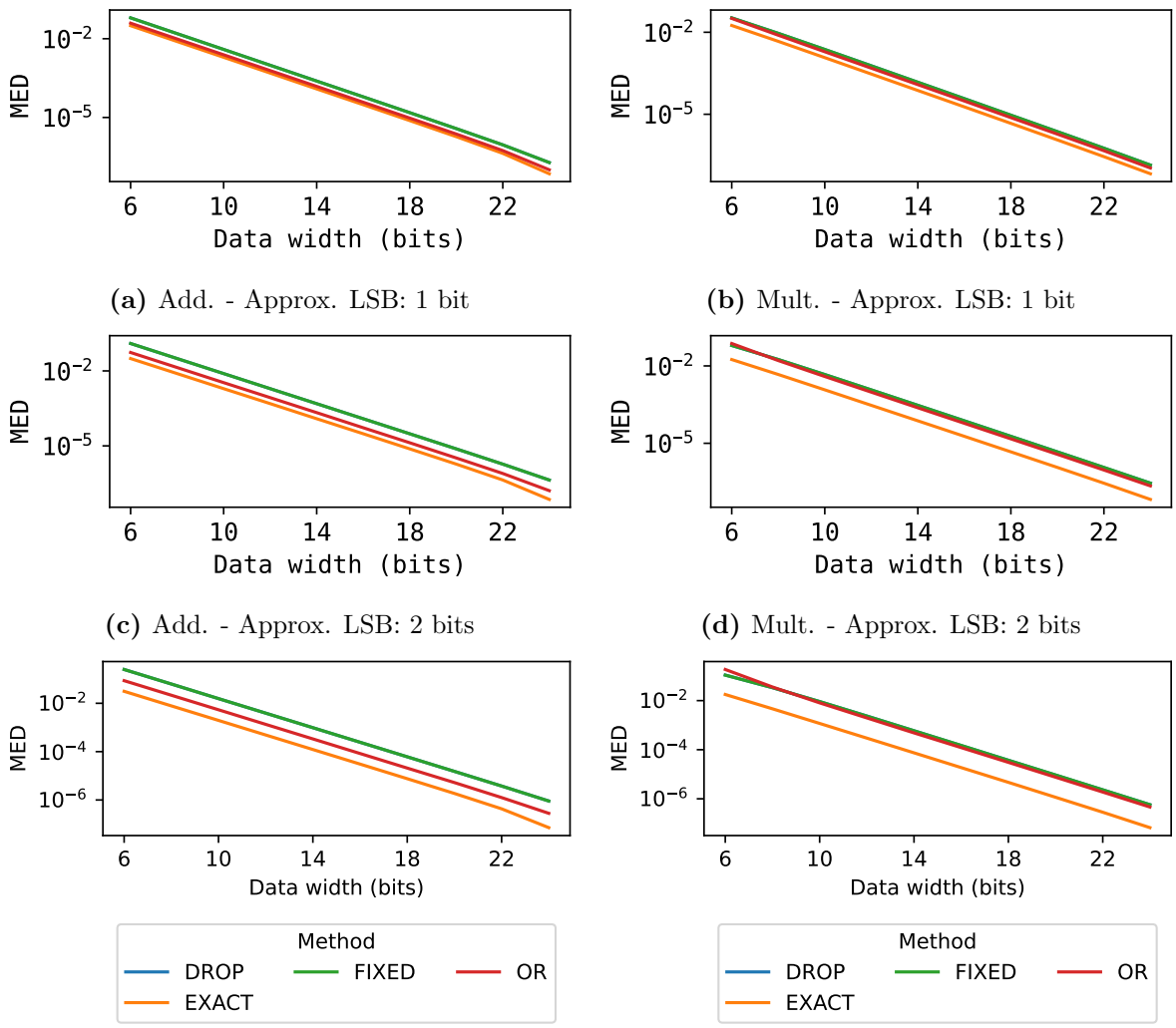
Figure 4.5: Resource consumption of the proposed multipliers in terms of LUTs when increasing the data width under three different approximations. On the left LUT consumption, and on the right, FF consumption.

$$\bar{e} = \frac{\sum_{i=1}^{N} e_i}{N} \tag{4.4}$$

where e_i is the error at the i-th sample and N is the number of samples.

The mean error is used as a derivation of the Mean Error Distance (MED), widely used in approximate adder evaluation [127]. However, given the number of bits, the number of samples may exceed 100 million to compute the MED. For this reason, we propose an evaluation using a random sample population based on the central limit theorem [128].

Figure 4.6 illustrates the errors of the adders and multipliers as the data width is varied according to the number of approximated LSBs. Within the proposed adders, `LSBDROP` has the most significant errors, followed by `LSBFIXED` (with slightly more degradation) and the `LSBOR`. In most cases, the `LSBDROP` with 1-bit approximation equals the error of the `LSBOR` with 2-bit approximation. It suggests that the `LSBOR` is superior in terms of ap-



(a) Add. - Approx. LSB: 1 bit

(b) Mult. - Approx. LSB: 1 bit

(c) Add. - Approx. LSB: 2 bits

(d) Mult. - Approx. LSB: 2 bits

(e) Add. - Approx. LSB: 3 bits

(f) Mult. - Approx. LSB: 3 bits

Figure 4.6: Mean Error Distance (MED) with respect to the distance between the minimum and maximum number representable within the notation. In this case, between -1 to 1.

proximations. Regarding resource consumption, `LSBOR` approximations are only promising with a 14-bit data width. The trade-off between error and resource consumption reduction in other cases provides fewer benefits. Moreover, the `LSBFIXED` resulted in being as error-prone as the `LSBDROP`, suggesting that there are almost no benefits in using this approximation

In contrast to the adder, the numerical errors between the proposed approximations are closer in the multiplier operator. For the 14-bit data width with a 2-bit approximation, the `LSBDROP` adder has a mean error of 0.049% and the `LSBOR` of 0.022%. However, the `LSBDROP` multiplier has an error of 0.118% and the `LSBOR` 0.110%. Therefore, operating on the LSBs of a multiplier has less impact on the error than in the adder. There is more benefit in using bit truncation regarding resource gain (7.14% in contrast to -11.8% of losses that pose the `LSBOR` multiplier) in exchange for numerical accuracy.

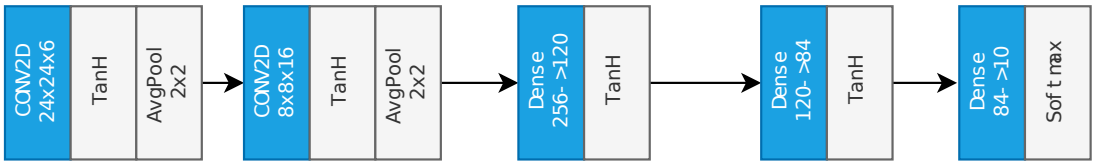
Compared to the state-of-the-art (SOTA), for a 16-bit adder, DeMAS has two configurations: 1) 12 LUTs with 0.9% of Mean Relative Error (MRE), and 2) 8 LUTs with 2.2% MRE [113]. The proposals from this work consume 17 LUT with 0.022% of MRE for the same configuration while using three approximated bits. In the multipliers, SMApproxLib proposes 8-bit configurations with 54 LUTs with 27% of MRE. In comparison, the proposed, in 3-bit approximation, achieved a 3.4% error while using 17 LUTs only without consuming FF and DSP cells, outperforming the proposed by SMApproxLib. In terms of usability, the proposed operators are parametric in terms of configuring the number of approximated bits and the numerical precision, in contrast to the SOTA proposals that are manually designed and fixed in numerical precision. Another relevant aspect of the characterisation is that when reaching more than 3 bits of approximation, the resource consumption oscillates. For this reason, this work limits the analysis to three approximated bits.

Apart from evaluating the operators isolated, this work uses the AxC Executer to integrate the approximate operators into a LeNet-5 [98] model trained on the MNIST [129] dataset for digit classification with Zero-Shot Quantisation (ZSQ), which implies quantising the weights to fixed-point numbers as they come, for energy saving. The evaluation seeks to study the impact of the operators on the overall model accuracy when varying the total data width (keeping a 6-bit integer part) and the number of approximate bits while using `LSBDROP` and `LSBOR` on the dense and convolution operations. `LSBFIXED` is partially covered since the benefits are insignificant, as demonstrated in the previous section.

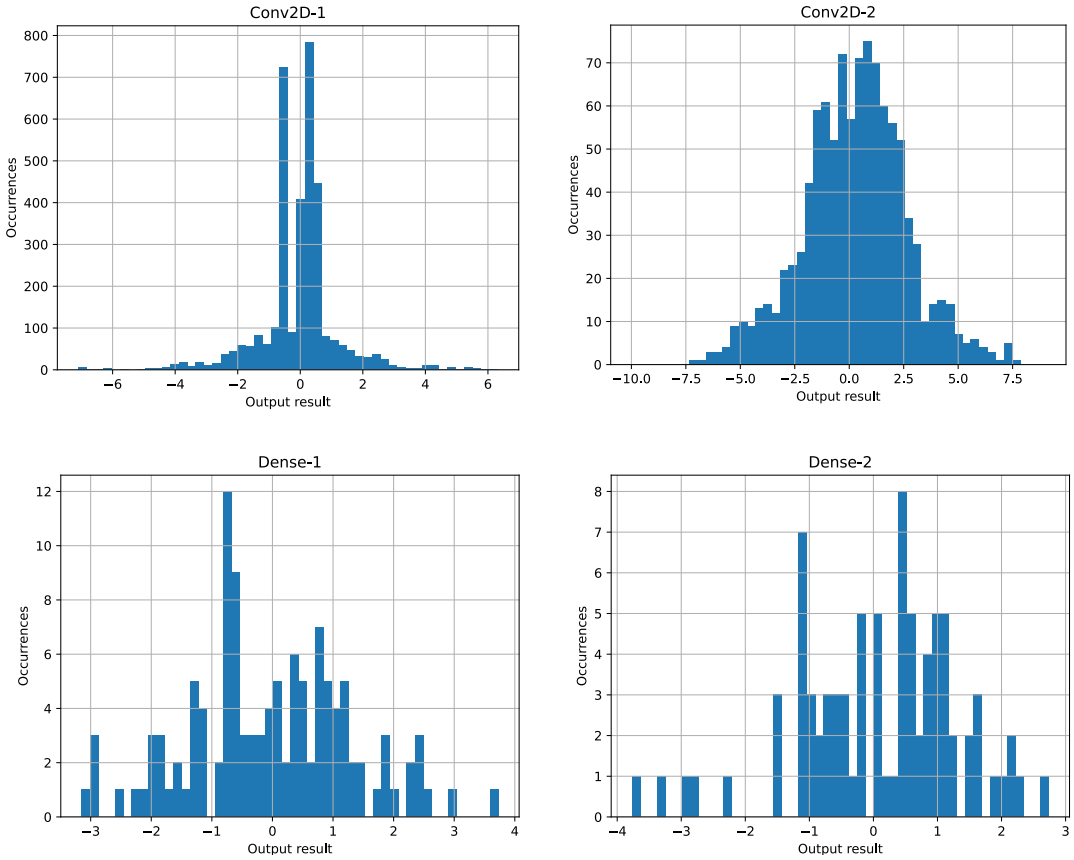
The first step for the DL model evaluation is to determine the accuracy loss when varying the data width (with a fixed integer part width) and the approximations, regardless of the PE implementation, which are set to be the spatial convolution and the standard matrix multiplication. The integer part can be found from the model histograms, taken when executing the model using floating-point. Figure 4.7 shows the LeNet-5 histograms when inferring the dataset. The dynamic range oscillates between -8 and 8, suggesting that five bits are, at least, required. So, using an even number of bits, a 6-bit integer part was chosen.

Then, the AxC Executer is run to evaluate all possible candidates using the DSE. Figure 4.8 shows the results of the inference accuracy and how it is affected by the several solutions. The starting point is the single-precision floating-point inference, showing an accuracy of 0.9878. After quantising, the inference with 12-bit fixed-point numbers has accuracy losses greater than 0.35%, making them unsuitable in most inference applications. The 14-bit-quantised inference reports more promising results, achieving 0.9724 (1.5% of accuracy loss), leading to a model compression of 2.29×. For 16-bit quantisation, the inference reports an accuracy of 0.986, whereas most approximation techniques have an accuracy loss of less than 15%. However, the goal is to minimise the data width until a reasonable minimum while maximising resource savings.

Regarding the implementation, the evaluation aims to utilise four accelerators to run a LeNet-5 to perform the convolutions (covered in previous research [88]), matrix multi-



(a) LeNet-5 Model Oversimplified



(b) LeNet-5 Output Layer Histograms

Figure 4.7: LeNet-5 model and its output layer histograms. These histograms represent the data involved in the outputs of the layers. The minimums and maximums represent the dynamic range required to fit into the fixed-point's integer part.

plications, and two additions based on the FAL accelerators, proposed in the previous chapter. This is only possible when using a single quantisation across the layers, minimising the granularity of the acceleration and reusing the units as much as possible. To analyse the effects of the approximation in these operations, a DSE that varies the number of approximated bits, the type of approximations and the accelerators affected by these approximations is performed.

Figure 4.9 shows the DSE plot of the weighted area consumption vs. the LeNet-5 Top-1 accuracy. On the left side of the front (less area consumption), the `LSBDROP` and `LSBFIXED` are the best-performing, whereas for the upper side (more accuracy), it is possible to have Pareto's optimal in all approximations. `LSBOR` has a point in the Pareto's front but tends to consume more than similar points in `LSBDROP`. Taking into account a threshold of 20%

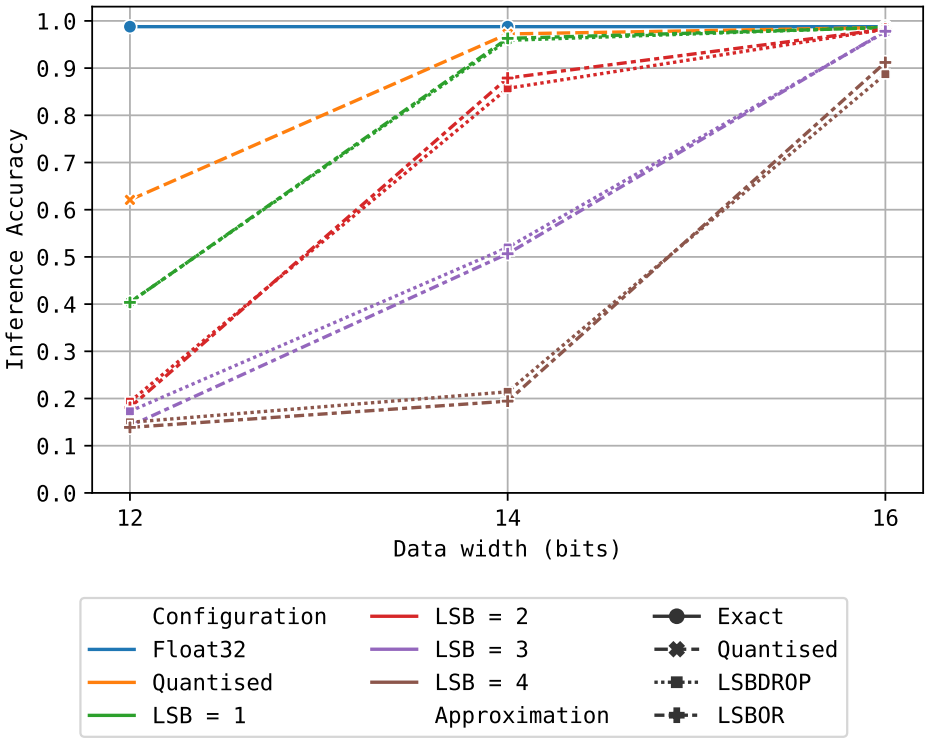


Figure 4.8: Inference accuracy of the LeNet-5 model after quantisation and approximations without quantisation-aware training. The fixed-point representation uses *data width bits*, where 6 bits represent the signed integer part. This accuracy measures the results of the first 5000 samples of the MNIST dataset.

Table 4.1: DSE Pareto Front Candidates. The points have been filtered taking 20% degradation as the threshold.

Configuration	Convolution	Adder 1	Dense	Adder 2	Area	Accuracy
Pareto 1	LSBOR	LSBOR	LSBOR	LSBOR	0.8107	0.904
Pareto 2	LSBOR	LSBOR	LSBOR	LSBOR	0.7860	0.831
Pareto 3	LSBOR	LSBOR	LSBFIXED	LSBFIXED	0.6764	0.801
Pareto 4	LSBOR	LSBOR	LSBDROP	LSBDROP	0.6764	0.801
Pareto 5	LSBFIXED	LSBFIXED	LSBFIXED	LSBFIXED	0.6094	0.790

of maximum accuracy loss, Table 4.1 shows the area consumption gains under different approximations and the associated accuracy loss. The best candidate shows a degradation of 9.6% while saving 18.93% resources. By softening the constraints in terms of accuracy, it is possible to gain up to 39.04% by still obtaining 79% accuracy.

4.4.2 Softmax Function Assessment

This section covers the assessment of different softmax accelerator configurations implemented on HLS to observe the difference between them and the exact version provided

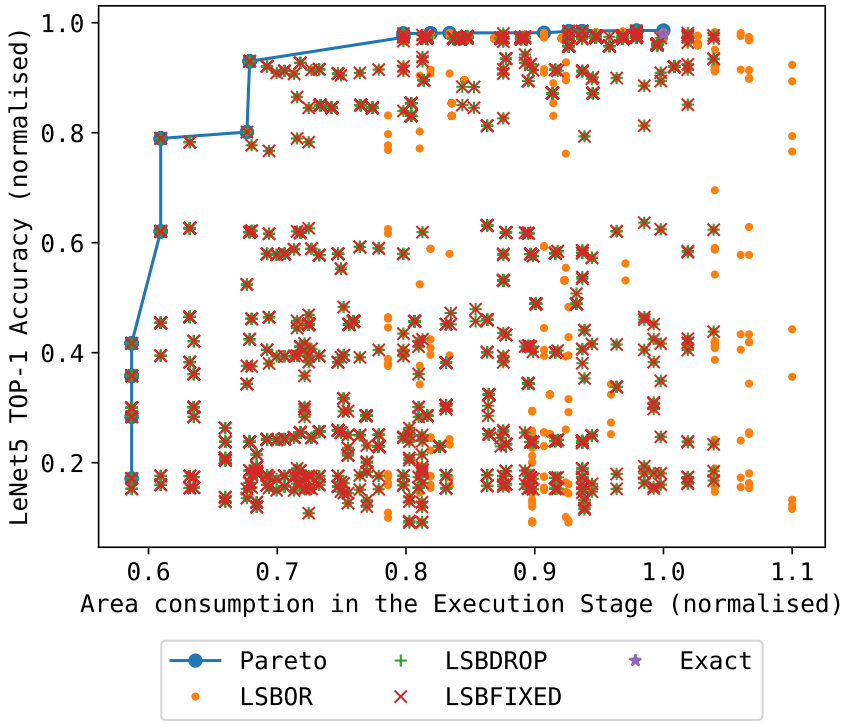


Figure 4.9: DSE of the implementation of LeNet-5 accelerators by using several configurations of the approximations within the layers.

Table 4.2: Error metrics for the Taylor-Softmax approximation

Type	**Error** (RMSE)	**Variance**	**Standard Deviation**
Order 1	$3{,}13 \times 10^{-3}$	$2{,}48 \times 10^{-6}$	$1{,}57 \times 10^{-3}$
Order 2	$2{,}97 \times 10^{-3}$	$2{,}45 \times 10^{-6}$	$1{,}56 \times 10^{-3}$
Order 3	$4{,}18 \times 10^{-5}$	$6{,}84 \times 10^{-10}$	$2{,}62 \times 10^{-5}$

by Vitis HLS (`hls::exp`), targeting an AMD Kria KV260 running at 250 MHz.

Tables 4.2 and 4.3 show the error metrics gathered for each softmax approximation type. The results were captured using a test vector with 1000 random values within the softmax domain $S =]-1{,}1[$ in a 16-bit fixed-point representation. The approach that generated the lowest error value from all the solutions presented was quadratic interpolation using LUTs with 64 samples, reaching RMSE $= 2{,}31 \times 10^{-7}$. In the case of the Taylor approach, the third-order approximation was the one that obtained the best error result with RMSE $= 4{,}18 \times 10^{-5}$.

Regarding the evolution of resource consumption vs complexity, Figure 4.10 shows how the resource consumption and latency scale as the data width changes in softmax accelerators based on both approximation methods. The *Taylor Approximation* uses a third-order Taylor approximation, and the *Linear Interpolation* uses a 64-sample LUT for the exponential function. Both cases use a 16-bit fixed-point data type and a 1024-element vector.

Table 4.3: Error metrics for the LUT interpolation softmax with 64 samples

Type	Error (RMSE)	Variance	Standard Deviation
Linear	$3{,}22 \times 10^{-6}$	$4{,}28 \times 10^{-12}$	$2{,}07 \times 10^{-6}$
Quadratic	$2{,}31 \times 10^{-7}$	$2{,}60 \times 10^{-14}$	$1{,}61 \times 10^{-7}$

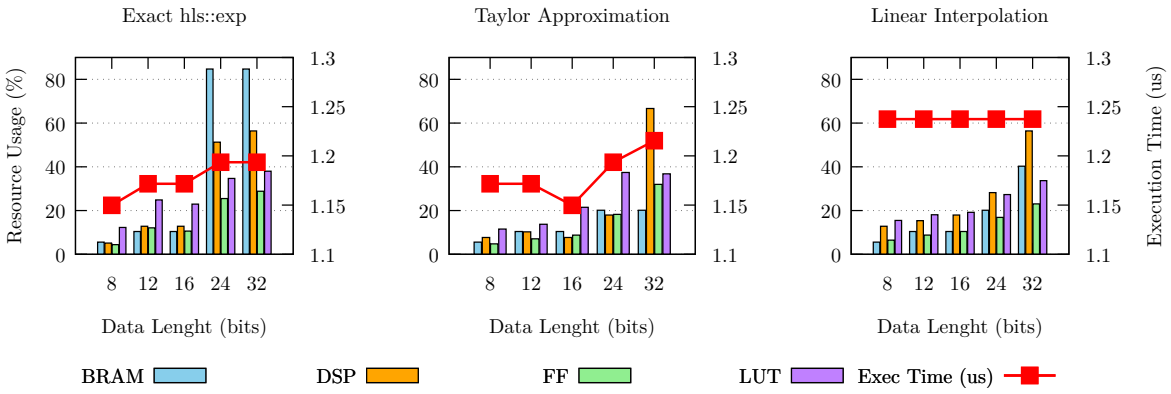


Figure 4.10: Resource usage and execution time of the softmax accelerators based on 3rd-order Taylor and 64-sample Linear Interpolation processing a 1024 16-bit fixed-point vector. The resource consumption is relative to an AMD Kria KV260.

The *Taylor Approximation* shows a latency oscillating between the 1.14 μs and 1.22 μs, resulting in a faster execution time compared to the *Linear Interpolation*, with a constant execution time of nearly 1.24 μs along the different data lengths. The exact version, in contrast, has configurations that are faster than the Taylor approximation, particularly at 8-bit and 32-bit data widths. In resources, *Taylor Approximation* consumes fewer resources than *Linear Interpolation* up to 16 bits, where the former starts to have less overall consumption than *Taylor Approximation*. In the case of the exact version, the overall consumption is always the greatest. Nevertheless, in both approximate accelerators, the consumption of DSP cells starts to grow exponentially as the data length increases due to the arithmetic complexity involved in the computations.

This highlights a trade-off between the data length, resource consumption, and numerical error. In scenarios where the error resilience is high, the Taylor-based approximation offers a lightweight solution to the computations. Otherwise, the interpolation-based approximation provides a more robust, low-error solution effective for high-resolution data (16-24-bit fixed-point). Likewise, there are scenarios like the 8-bit configuration, where using the exact version has more benefits than the approximate versions.

To evaluate the effects on actual DL models, this work considers different configurations of the softmax approximated functions at the error level and execution time, in two models: LeNet 5 [98] (evaluated with 10000 samples) and MobileNet v2 [100] (evaluated with 250 samples), using the AxC Executer.

Tables 4.4 and 4.5 show the results after accelerating the softmax layer, corresponding

Table 4.4: LeNet 5 Synthesis Results with 12-bit Fixed-Point (6-bit integer part) and a shift of 3 bits for an AMD Kria KV260

Configuration	Top-1 Accuracy	Layer Time (us)	LUT Cells	FF Cells	DSP Cells
Exact	0.9768	0.87	7940	5362	52
Interpolation 32 samples	0.9763	0.88	8050	5825	47
Interpolation 16 samples	0.9763	0.88	8010	5820	47
Interpolation 8 samples	0.9765	0.88	7997	5415	47
3rd-order Taylor	0.9763	0.89	7308	5879	52
2nd-order Taylor	0.9752	0.87	6684	4739	42
1st-order Taylor	**0.9751**	**0.84**	**6544**	**4615**	**37**

Table 4.5: MobileNet v2 Synthesis Results with 20-bit Fixed-Point (10-bit integer part) and a shift of 1 bit for an AMD Kria KV260

Configuration	Top-1 Accuracy	Layer Time (us)	LUT Cells	FF Cells	DSP Cells
Exact	0.748	1.17	32203	33043	160
Interpolation 64 samples	0.74	1.19	28975	26912	128
Interpolation 32 samples	0.688	1.19	28847	26816	128
Interpolation 16 samples	0.556	1.19	28559	26752	128
3rd-order Taylor	**0.872**	1.17	37223	37904	224
2nd-order Taylor	**0.872**	**1.15**	**21575**	**22653**	**128**
1st-order Taylor	0.0	1.12	18183	20400	64

to the last layer in both models. In the case of LeNet 5 (Table 4.4), the vector size is 10 elements of 12-bit fixed-point with a 6-bit integer part. It shows that the best configuration is the first-order Taylor, with 0.2% of Top-1 accuracy degradation, keeping the resources low compared to the exact implementation, saving 14% of resources (with the least saving in FF) with respect to the exact version. For the MobileNet v2, the softmax accelerator processes a 1000-element vector of 20-bit fixed-point elements, with a 10-bit integer part. The Taylor approximation improves the Top-1 accuracy compared to the exact version. This happens because the approximation introduces healthy numerical disturbances within the model, which are not generalisable, as shown in the LeNet-5. Second-order Taylor is the best configuration, improving the Top-1 accuracy by 16.6%, while saving 20% of overall resources (with the least saving in DSP) with respect to the exact version. The linear interpolation was used to evaluate the exponential function, resulting in a more expensive solution than the Taylor approximation by 1.2× (comparing

second-order Taylor and Interpolation of 16 samples).

After evaluating actual DL models with the approximation, it is possible to observe the benefits of approximating the exponential function in the softmax layers. The error resilience of this type of layer is robust enough to support the Taylor approximation, resulting in an opportunity to reduce the computation complexity to a polynomial-like computation. Evaluating more cases, such as LLMs, may yield interesting results given the amount of softmax computations (the eighth most intensive computation in Llama 2 [9], as explained in Section 5).

With respect to the related work, the design proposed in this work operates with arbitrary precision on a LeNet-5 (12-bit fixed-point precision). It achieves a significantly lower accuracy degradation—no more than 0.2%—with a slightly improved delay of 3.65 ns, albeit at the cost of approximately four times the area. However, the proposed solution offers a key advantage: it is highly configurable in data precision, order, and number of samples, and it can be tailored to different models and deployment scenarios, providing greater flexibility during development compared to previous works, easily integrable within popular frameworks like HLS4ML [89].

4.5 Final Remarks

In this chapter, this work proposed configurable approximate arithmetic operators for addition and multiplication based on LSB OR-ing and bit truncation, parameterised in the data width and the number of bits to approximate. After synthesising the operators using HLS, the effects of modifying these parameters on the mean numerical error and resource consumption have been analysed. In the case of the adders, the Pareto optimal point is in the bit-truncation configuration with 14-bit data width and 2-bit approximation, with 36.84% savings in resources and 0.05% of mean error. In contrast, the LSB OR-ing multipliers have a similar error to the LSB truncation multipliers. Multipliers can benefit more significantly when using less than 12 bits in the exact part (MSB), avoiding DSPs and Flip-Flops. Hence, future exploration shall be driven by using different levels of approximations in adders and multipliers. Additionally, it outperforms similar work on multipliers for FPGA.

On the other hand, running a LeNet-5 model trained on the MNIST dataset has shown promising results. Introducing approximations in both operators (convolution and dense operators) results in an accuracy degradation of 9.6% relative to single-precision floating-point inference by using 18.93% fewer resources compared to the quantisation without approximated operands.

This chapter also explored various approximate implementations of the softmax function on FPGAs, focusing on Taylor series and linear interpolation with Look-Up Tables (LUTs). The results indicate that quadratic interpolation offers the lowest numerical error within the softmax domain. However, this method—and linear interpolation more

broadly—incurs increased execution latency due to LUT access and interpolation overhead and interpolation steps. In contrast, Taylor-based approximations deliver better performance, attributed to their simpler arithmetic structure for approximating the exponential function.

When deployed in real-world deep learning models such as LeNet-5 and MobileNet v2, Taylor approximations proved to be a practical trade-off, introducing minimal accuracy degradation while significantly reducing resource usage on FPGAs. For future work, these approximation techniques show promising potential for accelerating inference in large language models (LLMs), which are increasingly dominant in state-of-the-art AI applications and heavily rely on softmax computations.

Finally, as a side contribution, this work also created AxC Executer, a new framework to perform design space exploration and C++ evaluations for the most popular networks, allowing the introduction of new accelerator designs, approximation and arbitrary data types, emulating the platforms using C++.

Related Publications:

- D. Cordero-Chavarría, L. G. León-Vega, and J. Castro-Godínez, "Configurable High-Level Synthesis Approximate Arithmetic Units for Deep Learning Accelerators," in *2024 IEEE 42nd Central America and Panama Convention (CONCAPAN XLII)*, 2024, pp. 1–6. DOI: 10.1109/CONCAPAN63470.2024.10933846
- A. Leiva-Valverde, F. Elizondo-Fernández, L. G. León-Vega, *et al.*, "A Quantitative Evaluation of Approximate Softmax Functions for Deep Neural Networks," *Accepted in 10th Workshop on Approximate Computing (AxC 2025)*, 2025. DOI: 10.48550/arXiv.2501.13379

Repositories:

- D. Cordero-Chavarria, L. G. Leon-Vega, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Math Operators*, version v0.1.1, Feb. 2023. DOI: 10.5281/zenodo.7708216
- L. G. Leon-Vega, D. Cordero-Chavarria, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Computing Executer (AxC Executer)*, version v0.1.0, Mar. 2023. DOI: 10.5281/zenodo.7712042
- León-Vega, L., Leiva-Valverde, A., Prieto-Sibaja, L., Blanco, G., Castro-Godínez, J., *HLS FPGA Accelerators*, 2024. [Online]. Available: https://github.com/ECASLab/hls-fpga-accelerators/

Chapter 5

FPGAs and LLM-based Inference

Large Language Models (LLMs) are gaining prominence due to their versatility, robustness, and broad applicability. They are being integrated into a wide range of applications, including chatbots, productivity tools, and robotics, setting the stage for a modern Internet era. Due to their complexity, optimisations such as non-uniform quantisation have proven to be effective, achieving up to 1-bit compression [131].

In the previous chapters, this work has addressed the acceleration of Deep Learning models through generic accelerators for matrix multiplication and optimisations based on approximate computing. It is possible to continue pushing the efforts to accelerate LLMs based on the foundations covered in this manuscript so far.

This section examines various architectures for DL acceleration, including those from HLS4ML and FINN, as well as layer acceleration using generic accelerators, their applicability, and limitations for LLM inference on FPGAs. It compares various implementations and pioneers an architectural feasibility study based on resource consumption and latency, providing insights to guide the future development of LLM hardware accelerators on FPGAs and other kinds of reconfigurable hardware.

5.1 Related Work

Accelerating DL inference is an emerging field focused on optimising models and accelerators to reduce model size and enhance throughput and energy efficiency. GPGPUs have improved their architectures by integrating specialised units for matrix multiplication, a critical operation in AI. These enhancements, such as the tensor multipliers, have significantly improved time-to-solution and energy consumption, covered in Chapter 2. Additionally, specialised units, such as the Neural Processing Units (NPU), optimise convolution operations using algorithms like Winograd, common in deep learning-based computer vision [132].

FPGAs have also contributed to the emergence of various architectures. The first archi-

tecture consists of *mapping accelerators* that map the entire neural network model into the FPGA, converting it to a hardware implementation. This approach has shown superiority in inference throughput, achieving ultra-low latencies suitable for scientific research, telecommunications, and other critical applications. FINN-R [133] and HLS4ML [89] are the most popular frameworks that use this architecture, supported by AMD and CERN, respectively.

The second architecture consists of *layer accelerators*. In this case, the layer is executed in a specialised unit. This method allows hardware resource reusability and the creation of multi-purpose hardware not limited to model execution. Although this approach results in lower inference throughput due to increased communication overhead between the host processor and the FPGA, it reduces resource consumption, making it ideal for smaller FPGAs and accelerating larger models. This architecture has been explored previously in this manuscript. Other recent research has accelerated transformer layers on FPGAs [134], [135], showing promising potential, including optimisations to adapt the transformer blocks to fit the FPGA.

The third architecture consists of *specialised co-processors* implemented in the FPGA as soft accelerators, which often face difficulties in accelerating deep learning workloads due to a higher communication overhead. However, they are smaller than the mapping and layer accelerators, making them suitable for small FPGAs in applications not constrained by latency or throughput. The AMD Deep Learning Processor Unit (DPU) is one example of this architecture [68].

Additional architectures combine the above. For instance, *weight-fixed layer accelerators* choose the weights according to the executing layer, minimising host-FPGA communication and increasing the throughput while keeping the resource utilisation low. The *operation accelerators* accelerate overall operations of the model and additional algorithms by scheduling work and transmitting data inside and outside the FPGA, enabling resource reusability, keeping the resource utilisation low, but at the expense of latency. The *spatial accelerators* are a hybrid architecture that accelerates groups of layers, rather than each layer, thereby accelerating and covering more parts of the network. An example is the spatial transformer acceleration [136], which spends resources but improves the latency.

LLM acceleration involves enormous trainable weights, ranging from billions to tens of billions of parameters. Given that these models require tens of gigabytes of memory, this complicates their implementation on standard personal computers, which are limited by the available primary memory. It does not exclude modern GPGPUs or FPGAs. Solutions to address this issue and speed up processing include using non-uniform quantisation [131], which reduces the required bits to around 2 bits, and model pruning. Nevertheless, it is not enough for implementations based on *mapping accelerators* and certain *spatial accelerators*, depending on the size and complexity of the model.

5.2 Architectures for Accelerating LLMs

Accelerating LLMs is an active research topic, presenting challenges due to resource constraints, high design complexity, and the size of the models. Frameworks that implement *mapping accelerators* roughly support transformer-based models, and this architecture becomes less suitable as the model size increases. Lightweight approaches such as the *operation accelerators* and *specialised co-processors* yield suboptimal results. The solutions available in the market (such as DPUs) lack support for LLMs and are not optimised for inference throughput, potentially increasing energy consumption. This limits the scope of our exploration to *layer-based architectures*, such as *layer*, *weight-fixed layer* and *spatial accelerators*. This work focuses mainly on assessing the latter architectures, highlighting their resource consumption, latency and energy consumption from a quantitative approach based on the Llama 2-7B [137].

5.2.1 LLM Architecture

LLMs are built on top of the transformer architecture introduced in 2017 [138]. Figure 5.1 illustrates the structure of a transformer, showcasing both the encoding and decoding process. The model comprises linear layers, which handle linear algebra operations, and non-linear layers, which employ activation functions to introduce non-linearity and prevent model collapse.

Key operations within transformers include matrix multiplication (*MatMul*), feed-forward network (*FFN*) and *Projection*. The non-linear operands often include the softmax activation, Gaussian Error Linear Unit (GeLU) or Sigmoid Linear Unit (SiLU), rotary positional embedding (ROPE) and Root Mean Square (RMS) normalisation.

Figure 5.2 shows a histogram of operations for the Llama 2-7B model with 32 transformer layers with 4096, 5120 and 8192 hidden embedding sizes [137]. The most frequent operation is matrix multiplication, followed by other linear operations, such as matrix addition and multiplication. Other operations, such as RMS normalisation, softmax, activations (unary), and ROPE, also require intervention, implying that an integral accelerator must consider all the aforementioned operations.

5.2.2 Implementation of Accelerators

There are various architectural choices for implementing DL and LLM accelerators. For illustration purposes, the analysis will be centred around an arbitrary deep learning model with two groups formed by fully connected layers, batch normalisation, and activation layers ① (as illustrated in Figure 5.3). Nevertheless, extrapolating the discussion to bigger models like LLM is homologous.

1. **Layer Accelerators**: accelerate the layer operations in their multiple configura-

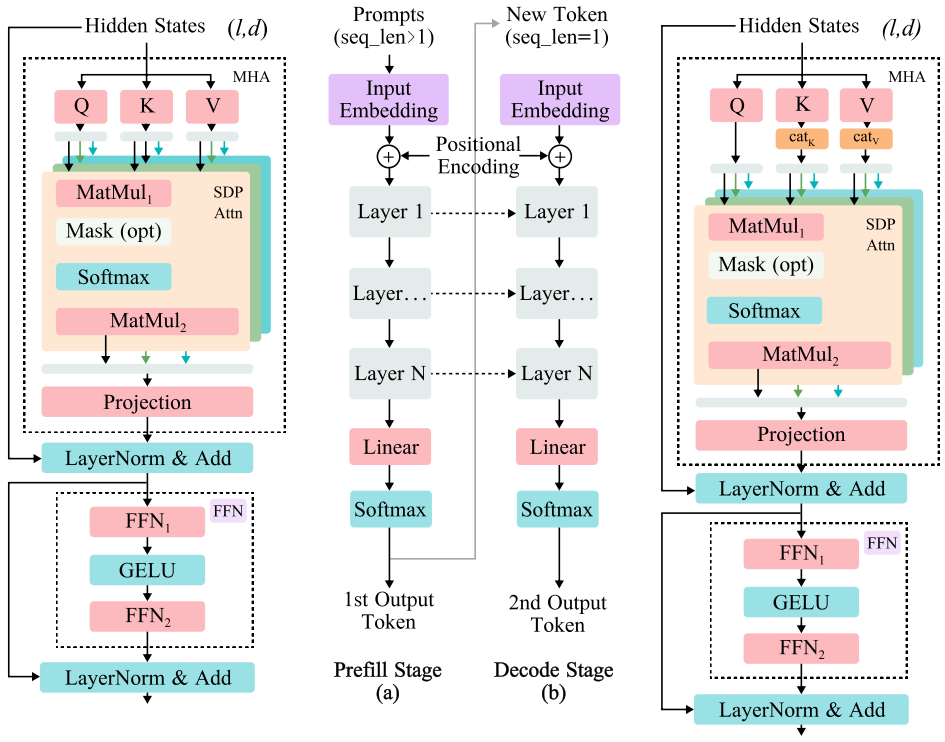


Figure 5.1: Scaled Dot Product (SDP) Transformer Architecture: composed of an encoder (left) and a decoder (right) with temporal dependency of one previous output. The turquoise boxes represent non-linear layers, whereas the red ones are linear.

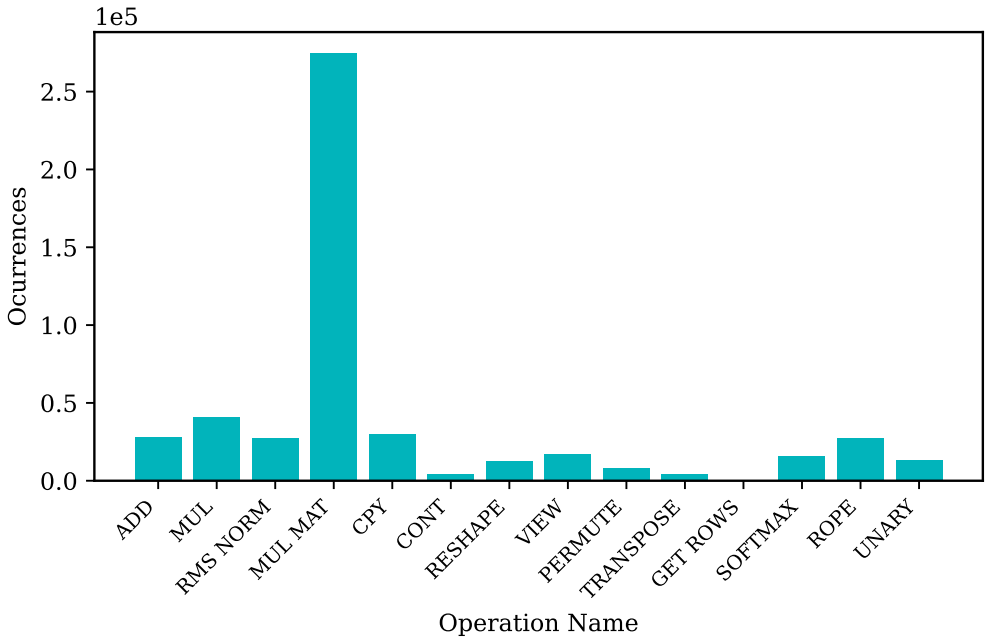


Figure 5.2: Operation count: histogram of operations executing a Llama 2-7B model using GGML framework.

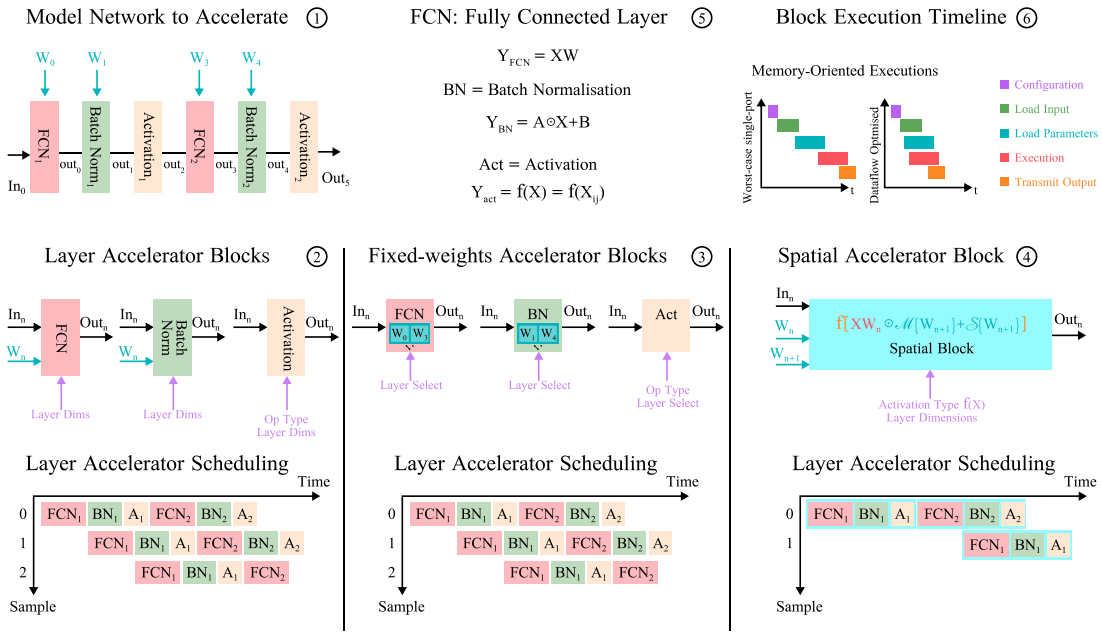


Figure 5.3: Architectures for Deep Learning Accelerators. The model to accelerate consists of two-layer blocks of fully connected layers, followed by batch normalisation and an activation layer. There are three possible implementations: 1) *layer*, 2) *weight-fixed layer* and 3) *spatial accelerators*. It is assumed that each accelerator follows the dataflow design pattern, and the memory access limits the number of ports.

tions ②. They generalise the number of inputs, outputs and parameters, specialising only in the mathematical operations. Some examples are matrix multipliers, convolution engines, and element-wise unary and binary operations. The simplest configuration corresponds to a single operation acceleration per unit, where all the operating layers of the same kind use the same accelerator as observed in ②, allowing a certain degree of parallelism and a low resource footprint.

2. **Weight-fixed Layer Accelerators** are similar to the *Layer Accelerators* with the key difference that all the network parameters are hard-coded to the accelerator, specialising in the mathematical operations and the network, generalising only the inputs and outputs ③. Weights are stored within Read-Only Memories (ROM), and the layers are selected using multiplexers, reducing resource usage and execution latency by removing the transfer of the parameters.

3. **Spatial Accelerators** is a variant of the *Layer Accelerators*, which fuse multiple operations into a single one, optimising the execution and avoiding unnecessary data transmission ④. They generalise the inputs, outputs and parameters but specialise in a set of operations. These accelerators perform a model hardware optimisation, combining the computation of more than a layer per execution cycle. In the case of the model in ①, layer operations presented in ⑤ are compressed into a single operation, and a spatial block is created as in ④. The impact of this optimisation on LLMs will be studied in the following section.

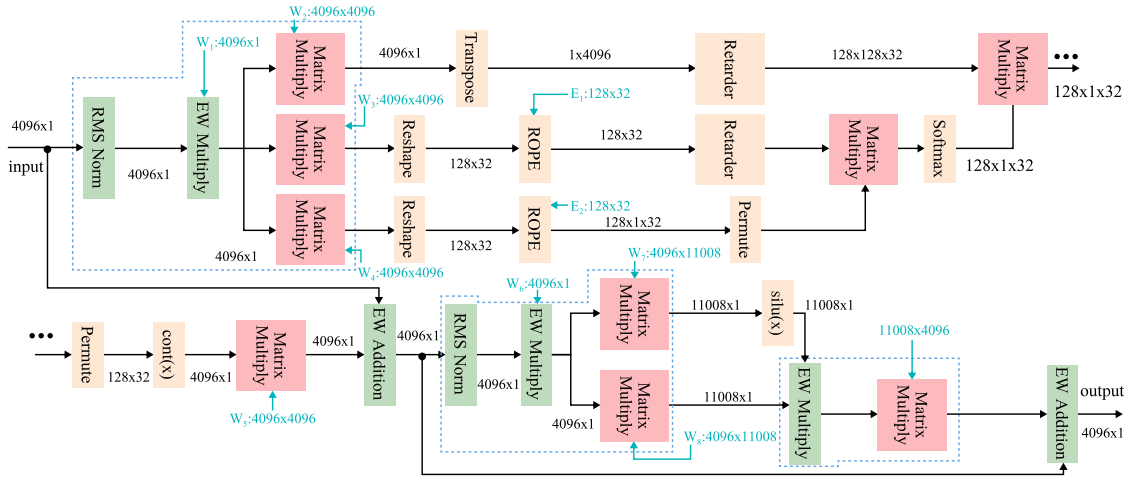


Figure 5.4: Execution graph of Llama 2-7B LLM when executing on GGML [139]. The execution involves matrix multiplications, element-wise vector/matrix operations and some auxiliary or special functions. There is a pattern along the graph highlighted in dashed blue lines, suggesting a possible optimisation.

5.3 Architecture Assessment from a Quantitative Approach

This section analyses the different architectures for accelerating the Llama 2-7B model implemented in the GGML framework [140]. The trade-offs, resource consumption and latency of various FPGA implementations are evaluated, providing insights into the feasibility of LLM acceleration on FPGAs. The reference hardware includes an AMD Alveo U250 running at 300 MHz, two AMD EPYC 7H12 with 64 cores each and an NVIDIA Tesla Volta V100 from the ORFEO cluster at AREA Science Park, Italy.

5.3.1 LLM Execution Graph

Figure 5.4 shows the execution graph of a single transformer layer from the Llama 2-7B LLM decomposed into vector and matrix operations. The most frequent operations include element-wise, norms (in green) and matrix multiplications (in red), which align with Figure 5.2. In this case, they can be accelerated using *layer accelerators*. Furthermore, there is a pattern (dashed blue lines) where multiple matrix multiplications receive the same output from an element-wise multiplication, enlightening an optimisation based on *spatial accelerators*, combining an RMS norm, an element-wise multiplication and up to three matrix multiplications.

Regarding the *weight-fixed layer accelerators*, the model contains more than 30 transformer layers, which implies 270 matrix multiplications and 90 matrix element-wise multiplications, where this technique becomes unsuitable due to the number of parameters that need to be stored in the FPGA look-up tables.

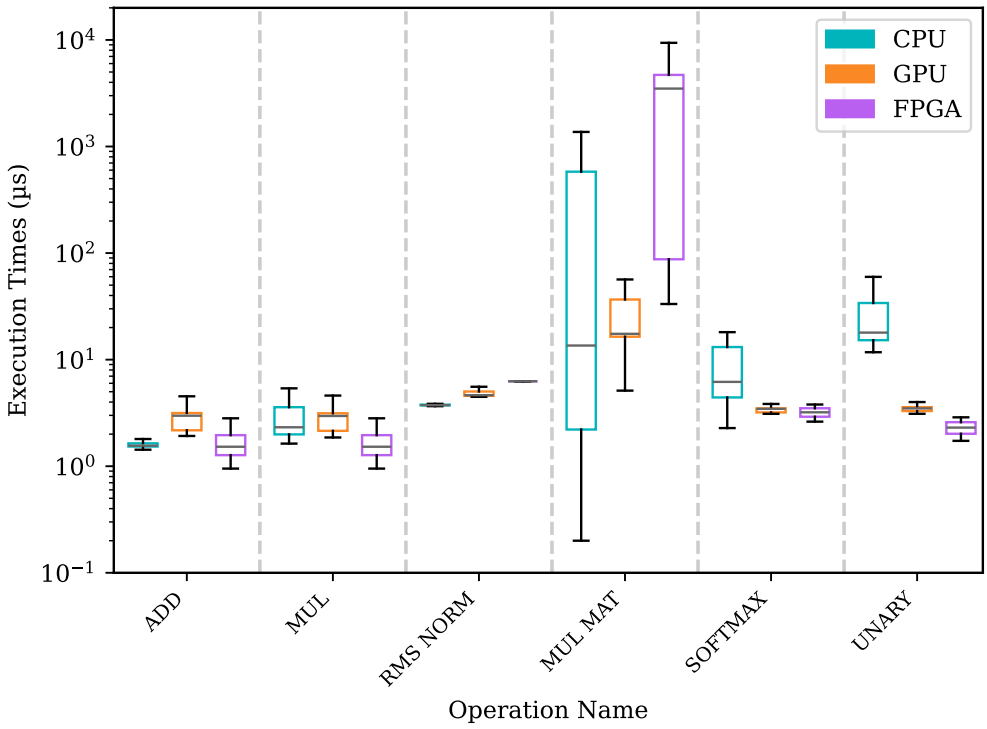


Figure 5.5: Execution times of LLM operations executed by CPU, GPU and FPGA. The GPU and FPGA times are the kernel execution times, neglecting any communication. The FPGA implementation is based on the *layer accelerator architecture*.

5.3.2 Performance Assessment

The initial assessment will consider the layer accelerator inspired by the generic architecture presented in Chapter 2, and the CPU and GPU implementations already available in GGML in version pre-b3620.

Figure 5.5 presents the execution times of various LLM operations on the CPU, GPU, and FPGA. The results indicate that the matrix multiplication (*MUL MAT*) is the most time-consuming operation, followed by *SOFTMAX*, *ROPE*, and unary and binary element-wise operations (*ADD, MUL, UNARY*). In the CPU, the matrix multiplication execution time ranges over five orders of magnitude. In contrast, the GPU ranges between two orders of magnitude, where the maximum execution time of the kernel of the GPU is about 40× lower than the CPU implementation, leading to a considerable speedup for the worst-case scenario. The GPU also provides speedups in other operations, except for addition and RMS normalisation, where the CPU dominates the worst-case measurement. According to Figure 5.2 and Figure 5.5, accelerating the matrix multiplication will impact the execution time more significantly than other operations, given its frequency and median execution time.

Besides, Figure 5.5 also displays the execution times of the kernels executed by the FPGA, implemented in C++ using High-Level Synthesis and IEEE 754 half-precision floating-

point. The execution times are competitive with those of GPU implementations in unary and binary matrix operations, including the softmax. However, the RMS normalisation and the matrix multiplication are the weakest operations on the FPGA side. At this point, the *layer accelerators* for matrix multiplication are a limitation to get a better performance on the FPGA.

5.3.3 Spatial Acceleration Assessment

Previously, it was unveiled that the matrix multiplication and the RMS normalisation were the slowest operations in the FPGA, requiring an evaluation of the *layer accelerator* architecture and opening room for a new architecture experiment. For the scope of this work, the focus will be on evaluating matrix multiplication, given its relevance to LLM execution at the operational level. The impact on the overall LLM inference time will be left for future work.

Section 5.3.1 highlighted the possibility of using *spatial acceleration* to fuse the execution of the RMS normalisation, an element-wise multiplication and matrix multiplications, as illustrated through the dashed line regions in Figure 5.4. Unlike GPUs, FPGAs offer an advantage by implementing pipeline data flows, which increases performance when combining operations as a single module. This work proposes the implementation of a single spatial accelerator that combines element-wise and matrix multiplication, leaving the RMS normalisation for future work. This accelerator receives a single input ranging from 4096×1 and 11008×1, and produces up to three outputs of 4096×1 elements. The matrix multiplications are conditionally executed on demand, not altering the adequate computation time, thanks to the parallelism offered by the FPGA. Therefore, the ideal case is the execution of three matrix multiplications as in the first block of Figure 5.4. Additionally, it is assumed that the weights and data are transmitted in separate ports to maximise the parallelism, as illustrated in the diagram ⑥ (right) in Figure 5.3, where the weights are transmitted in parallel to the input data.

Figure 5.6 compares the performance of matrix multiplication across different architectures: 1) using *layer acceleration* (FPGA Layer), 2) using *spatial acceleration* (FPGA Fused), and 3) with four replicas of *spatial acceleration* that can execute independently. The FPGA with *layer acceleration* is slower than the CPU in computation, mainly influenced by clocking and data communication from the host to the card, given that multiple kernels are launched. Although the communication overlaps with the calculation, each kernel still requires data transmission, which plays a significant role in performance degradation. Executing three matrix multiplications implies the triplication of data transmission times and computation, following the timeline expressed on the right of the ⑥ (Figure 5.3). The execution time also plays a relevant role in the energy consumption. The CPU consumes, on average, 80 W per socket, the GPU consumes 35 W and the FPGA 27 W, during execution. Due to the execution times, the FPGA outperforms the CPU in terms of energy consumption, offering similar performance while consuming a

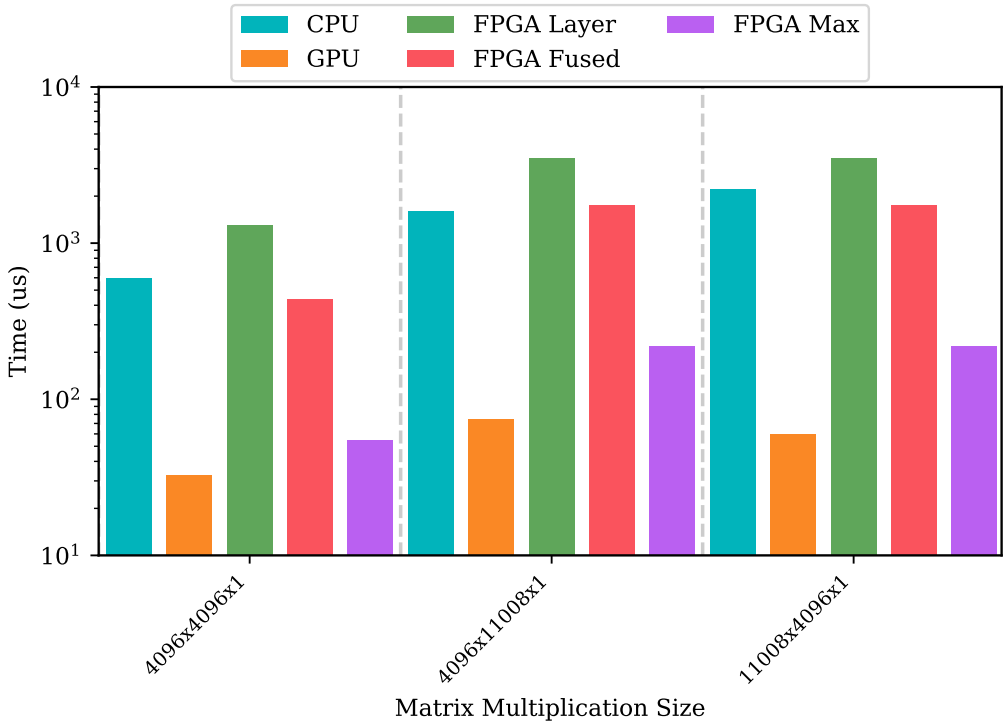


Figure 5.6: Median execution times of the matrix multiplication under different matrix sizes ($M \times K \times N$) on different architectures. The FPGA uses the *layer acceleration* (FPGA Layer), the *spatial acceleration* (FPGA Fused) and the latter with four replicas in a single FPGA (FPGA Max). Times include data communication overhead.

fraction of the power. On the other hand, the GPU is the winner due to its fastest execution. However, the FPGA can be more suitable than the GPU if the design runs faster and has lower power consumption.

The *spatial acceleration* aims to avoid data transmission replication by assigning multiple ports to feed the accelerator with the weights in parallel and leverage up to three simultaneous matrix multiplications, overlapping the data load. It implies a speed up of 2.99× in $4096 \times 4096 \times 1$ multiplications and 1.99× for other sizes. It makes the FPGA execution faster than the CPU execution by 1.37×. When placing four spatial acceleration units to parallelise the processing of multiple layers, the computation times are lower by 12× for $4096 \times 4096 \times 1$ and 8× for other sizes, with respect to the *layer acceleration*. It leads to a speed-up of 10.98× concerning the CPU in $4096 \times 4096 \times 1$, only overcome by the GPU, which is 1.66× faster than this optimised architecture.

The resource utilisation varies across the architectures, as shown in Table 5.1. The *layer acceleration* of all the operations combined fills 33% of the FPGA, whereas the *spatial acceleration* fills 10% of the FPGA capacity. This optimisation suggests that scaling the spatial accelerator is more resource-efficient, given that some hardware is reused, particularly the AXI-protocol interfaces and FIFOs. Until this point, the exploration of numerical and approximate computing optimisations is left for future work, leaving room

Table 5.1: Architectures Resource Consumption for single-precision floating-point

Configuration/ Resource	Layer Accelerator	Spatial Accelerator
BRAM	90	150
DSP	4036	1277
Flip-Flops	1143511	238519
Look-Up Tables	411098	99290
Overall Occupancy	33%	10%

for better architectures to reach and overcome the GPU execution times, emphasising the performance per Watt.

5.4 Final Remarks

This chapter studied various architectures for accelerating Large Language Models (LLMs) on FPGAs, highlighting the challenges and limitations associated with each approach, including network mapping, fixed-weight acceleration, layer acceleration, and spatial accelerators. A key finding emphasizes the unsuitability of network mapping and fixed-weight accelerators for LLMs due to the enormous number of parameters involved.

The assessment of individual layer accelerators identified the matrix multiplication as the slowest operation required for LLM inference. At the execution graph level, it is possible to determine that creating a spatial accelerator that combines element-wise multiplication and matrix multiplications significantly enhances performance. This approach resulted in speed-ups ranging from 1.37× to 10.98× over two AMD EPYC 7H12 CPUs with 64 cores each. The spatial accelerator was only slightly outperformed by an NVIDIA Tesla V100 GPU, which was 1.66× faster than the best FPGA implementation achieved in this work.

This demonstrates that FPGAs offer a feasible and efficient alternative for LLM inference, especially when optimised with spatial accelerators. Furthermore, it suggests that with further refinements, FPGAs could surpass GPUs in performance for this application. In future work, it is possible to introduce more optimisation, such as changing data types from half-precision floating-point to arbitrary-precision numerical representations tailored to the specific needs of LLMs and the exploration of more comprehensive spatial accelerators to achieve and potentially beat GPU execution times. All the hardware implementations for the accelerated layers are available as an open-source contribution in [130].

Related Publications:

- L. D. Prieto-Sibaja *et al.*, "LLM Acceleration on FPGAs: A Comparative Study of Layer and Spatial Accelerators," in *2024 IEEE 42nd Central America and Panama Convention (CONCAPAN XLII)*, 2024, pp. 1–6. DOI: 10.1109/CONCAPAN63470.

2024.10933896

Repositories:

- L. Prieto-Sibaja and G. Gerganov, *GGML - Tensor library for machine learning*, Jun. 2024. [Online]. Available: https://github.com/ecaslab/ggml

Chapter 6

Democratisation of the FPGA

In previous chapters, this manuscript has emphasised that Field Programmable Gate Arrays (FPGAs) are promising devices for hardware-based acceleration, given their reconfigurability and flexibility in representing new hardware architectures at a lower power consumption. Nowadays, getting started with FPGAs is becoming increasingly accessible thanks to the use of untimed C++ to represent hardware designs using High-Level Synthesis (HLS) tools. However, it remains challenging due to the diverse workflows, frameworks, tooling, HLS dialects, and the lack of libraries and familiarity with hardware concepts, which can overwhelm scientists and developers. Moreover, current software tools for AMD FPGAs are designed to be platform-agnostic, facilitating seamless transitions between edge and cloud environments by supporting legacy OpenCL. However, this agnosticism has not yet been analysed regarding the possible implications of centralising the entire support into a single runtime library instance.

In contrast, GPGPUs are more flexible and standardised. NVIDIA, the most popular GPGPU vendor, offers CUDA, an ecosystem comprising device drivers, daemons, libraries, and a framework that enables the development of accelerated applications in a single environment, with API agnosticism among GPU families within the same vendor [19].

This chapter addresses the difficulty of using FPGAs by proposing an open-source runtime daemon that allows the use of a set of selected generic accelerators that can be accessed through an API without requiring knowledge of hardware, carrying out:

- Analysis of the XRT and PYNQ regarding latency when scaling in workload size.
- Development of a simplistic Runtime Library with a PYNQ-like API to develop C/C++ applications.
- Resource scheduling that allows the use of a single accelerator by multiple processes.
- Memory allocation and management with zero-copy mechanisms to avoid additional overhead and the possibility of arbitrary precision data types.

- Acceleration by using a pre-built set of out-of-the-box reconfigurable accelerators for common matrix and vector operations, useful for AI and Digital Signal Processing (DSP) to prevent users from coding their custom kernels unless optimisation is required.

6.1 Related Work

For GPU development, NVIDIA propose the CUDA framework, which consists of an ecosystem with device drivers, a scheduling daemon, runtime libraries and compilers installed altogether that makes the programming experience seamless across the architecture or platform, imitating pure software development [19], being more friendly to a vast majority of developers and users. On the other hand, FPGA vendors, particularly AMD (formerly Xilinx), have tried to follow a similar approach, easing the learning curve of using FPGA for hardware acceleration through multiple methods, which include the Xilinx Runtime Library (XRT) [141], the unification of the workflow with Vitis [142], prototyping with Python using FINN [133] and PYNQ [143], compatibility with OpenCL and C++ for High-Level Synthesis and libraries included in Vitis [144].

All these solutions simplify the usage of the FPGA to *implement custom kernels* (accelerated code) but still require hardware knowledge during the process to lead to efficient designs. The Vitis Libraries [144] include a set of algorithms for DSP, AI and Linear Algebra that do not require intervention from the user in the kernel creation; however, some of the algorithms are closed-source without a chance of using different data types or introduce more *aggressive optimisations needed for AI*. On the other hand, XRT and PYNQ do not offer the possibility of using arbitrary precision data types, which are helpful for FPGAs in reducing communication bottlenecks and design resource consumption.

In the community, the FPGA Operating System (FOS) offers a solution for using FPGA through multiple processes. It is capable of scheduling resources and managing data, with support for Python and C++ in Linux applications. Users can develop their custom kernels using partial reconfiguration, which is launched during initialisation process of an application [145]. FOS is a promising approach for managing the FPGA as a multi-accelerator resource, requiring users to develop custom kernels. Moreover, it only targets edge FPGAs, such as the AMD ZYNQ family, leaving aside other FPGA variants.

A common approach to ease the user in developing custom kernels is using either HLS or OpenCL with Vitis [142]. From the community, multiple techniques have been used to improve the use of OpenCL, proposing hot plug-and-play custom kernel usage and partial reconfiguration under the hood [146]. FINN [133] and HLS4ML [89] are frameworks that particularly ease the development of DL-based inference applications accelerated by FPGAs. They receive a model and additional hyper-parameters, such as the reuse factor (related to hardware replication) and quantisation, and map the entire fitted model into an accelerator for that specific model. Between them, the best-maintained is FINN, with

the support of AMD. However, these solutions have the inconvenience of mapping the entire model, depleting FPGA resources aggressively without the opportunity to reuse the design for another type of computation, which is inconvenient in embedded FPGAs. Nevertheless, they perform best when they work, thanks to the FPGA data flow optimisations.

During this work, multiple alternatives to the model mapping architecture have been explored and proposed, such as generic accelerators, which are parameterised in data type, data width, number of operands, math operators and algorithm implementation (Chapters 3 and 4), and the spatial accelerators for LLMs (Chapter 5). These implementations are open-source to facilitate development and research on top of them, and some have been tested against kernels generated by HLS4ML, yielding interesting results, even when using zero-shot quantisation [62].

When transitioning to the software domain, PYNQ and XRT provide runtime libraries that utilise the FPGA binary configuration file to configure the FPGA, manage memory, and interact with the hardware accelerators implemented in the design [141], [143]. They can be installed in the newer Ubuntu distributions offered by Canonical-AMD for the FPGA-based SoCs, which eases the host development and avoids unnecessary cross-compilation, as is the case with the homologous GPGPU-embedded platform, the NVIDIA Jetson [31], allowing more complex development environments in terms of the software stack than the traditional cross-compilation approach, which uses Petalinux for creating a custom Linux image [147], [148].

This highlights a fragmentation in contributions and a lack of solutions that facilitate the development of FPGA-accelerated software applications without expertise in hardware architecture. This is due to the disconnection of vendor libraries from user needs and the lack of integration that does not occur in the CUDA environment of NVIDIA GPUs [19]. Additionally, the runtime libraries and workflows have not been analysed in terms of their overhead and starting latency. This emphasises the need for a compute-centric API, an analysis of FPGA-host communication, an accelerator manager, and a set of existing libraries.

6.2 Latency of Host Applications for FPGA Deployments

This section examines the commonly used host APIs utilised for FPGA-host communication, highlighting their functionality, strengths, and improvement opportunities, ultimately proposing a framework that combines the strengths of these APIs.

6.2.1 Current Issues with PYNQ and XRT

PYNQ is a Python framework that enables FPGA configuration from a bitstream (in AMD: .bit) or a reconfiguration file (in AMD: .xclbin), making PYNQ compatible with both legacy and the new Vitis workflows. Moreover, it presents wrappers for NumPy library interoperability and uses XRT for memory management. If PYNQ is configured with a bitstream, it utilises the Linux FPGA manager for configuration and Memory-Mapped Input/Output (MMIO) to handle IP transactions; otherwise, it uses XRT [143]. PYNQ suffers from overhead due to Python interpretation, which may significantly impact low-latency and light workload applications.

XRT, in contrast, is a C++ runtime library compatible with cloud and edge FPGAs that only takes the reconfiguration file generated by the new Vitis workflow. It introduces the concept of compute units and provides interoperability with OpenCL. The IPs are managed by registering transactions using mailboxing. For each register I/O, XRT performs an argument check, computes the offset addresses and performs *safe mailboxing* (`xrt::mailbox`), which polls for every register transaction [141]. The kernel (accelerator or IP) is also abstracted into compute units, implemented through multiple inherited classes (`xrt::run`, `run_impl`, `kernel_command`, `xrt_core::command`, `command_manager`) [141]. During the process, XRT adds various pollings, synchronisation, command messages, and function calls, which are unnecessary for simple designs but necessary for complex multi-functional designs.

Comparing the differences between the two frameworks regarding IP control, the MMIO approach performed by PYNQ is superior to the compute unit approach by XRT. When executing the same PYNQ MMIO implemented in C, XRT is 2.79× slower than the C MMIO when configuring the IP and 2.82× slower, on average, when performing synchronisation for a deep learning model inference. This implies that launching the IP core execution will result in greater startup latency in XRT.

In summary, PYNQ suffers from Python interpretation overhead, adding more time to the transaction startup. In contrast, XRT suffers from overhead in kernel launching due to the mailbox, which implements a series of polls, synchronisation, and several function calls.

6.2.2 CYNQ RT: the best from PYNQ and XRT

This work presents the CYNQ runtime library (CYNQ RT), an open-source library based on the interface-adapter architecture that makes the API agnostic. The foundation of the CYNQ Runtime Library (CYNQ RT) is to be as simple as using PYNQ to develop the host application while having the performance of a C++-based application without adding unnecessary machinery. CYNQ RT is more flexible than XRT, given that it does not impose the use of the new Vitis workflow to generate the configuration files.

Figure 6.1 shows the overall CYNQ RT architecture. It consists of four interfaces acces-

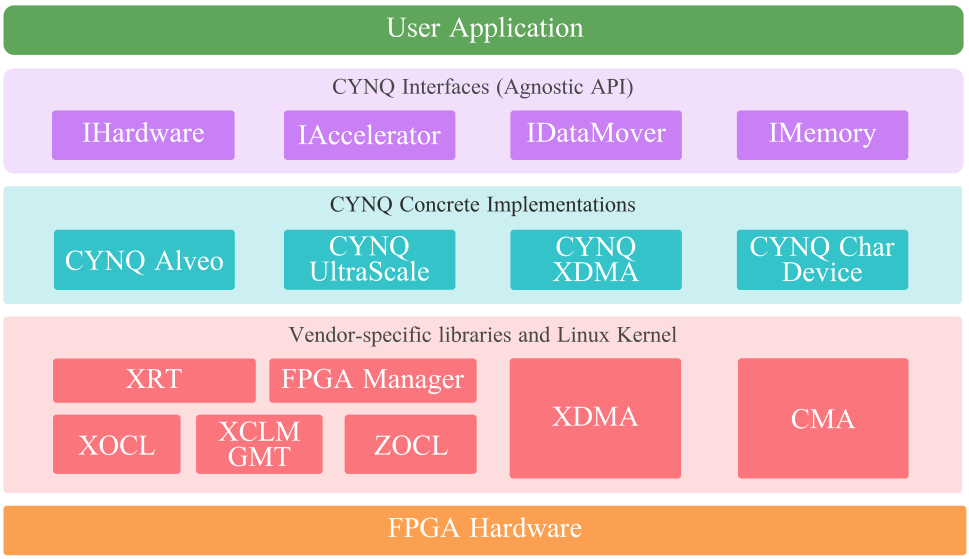


Figure 6.1: CYNQ RT Architecture. The user interacts with the interfaces, implemented through adapters that meet the hardware-specific details to connect with the FPGA.

sible to the user, keeping the API agnostic without falling into implementation details. Every interface has multiple implementations that communicate with vendor-specific libraries and/or Linux device drivers, depending on the FPGA model.

- **IHardware** abstracts the platform into a class that initialises the FPGA interfaces, registers, and configurations through either a bitstream (.bit) by using the FPGA Manager or reconfiguration binary file (.xclbin) by using XRT. It also works as a factory for the other interface instances according to implementation details.
- **IDataMover** provides an interface for allocating, uploading and downloading memory (whose host and device addresses are wrapped into **IMemory**). The implementation of ZYNQ-based platforms is based on DMA and XRT Buffer Objects.
- **IAccelerator** facilitates access to the IP core registers. It provides an interface to start/stop the accelerator, read/write the IP core registers and synchronise execution. For ZYNQ-based platforms, it is based on MMIO instead of the XRT kernel.

The process begins with the IHardware, which performs the FPGA's configuration according to its variant (ZYNQ, PCIe XDMA, or Alveo, where the last two are not part of the scope of this document) to set up data management, bus communication, clock speed, and other configurations. Once the FPGA is configured, IP core instantiation (IAccelerator) is performed through MMIO, memory management by XRT, enabling accelerator execution.

The CYNQ RT achieves three main goals:

1. provides an agnostic API, making it portable,
2. keeps simplicity, and
3. offers compliance with XRT and PYNQ.

The application can be easily transferred from one FPGA-based system to another with minimal changes. The interface's implementations (or adapters) provide this capability without intruding on the final user application. The final goal includes support for non-XRT-capable platforms, like low-end and non-AMD FPGAs.

6.3 CYNQ Framework: User-Friendly Ecosystem for AI FPGA-based Accelerators

To mitigate the fragmentation of the tools, this work proposes an integral solution to the existing issues regarding the implementation of custom acceleration kernels and the difficulties of using FPGAs in hardware acceleration by integrating an *agnostic API* that focuses on actual computations, a set of *pre-built libraries* that can be added to an acceleration stack through a linker, and a *runtime daemon* that manages the FPGA, memory allocation and schedules the execution of the workloads coming from different user applications. This aims to allow users access to the accelerators rather than managing the FPGA. Similar proposals have been successfully implemented in HPC CPU and GPU libraries, such as BLAS, cuBLAS, FFTW, cuFFT, LAPACK, and others [59], [149]–[151]. In these cases, the user configures the execution through a series of parameters passed as arguments to functions that perform all the configuration required and launch the computing kernels.

Figure 6.2 shows the proposed architecture as a stack. The *FPGA hardware* comprises xfOpenCV [152], Vitis Libraries [144] for Computer Vision, and the Flexible Accelerators Library (FAL, proposed along this work). With these libraries, the FPGA is equipped with accelerators for matrix multiplication (GEMM), convolutions (1D and 2D), and element-wise operations, which are the most common in AI. Unlike HLS4ML, FINN, and similar work, which map the entire unrolled model to the FPGA, we propose using reusable designs to accelerate AI simultaneously with other data pre-processing and post-processing applications, such as image filtering, demosaicing, warping and resizing. This allows a broader acceleration scope that accelerates more parts of the code rather than only the inference model, and the use of smaller FPGAs. However, this introduces a trade-off between a lower inference throughput due to data flow partitioning and a more granular acceleration throughout the entire code. Beyond that, the hardware layer is expandable to support additional operations by integrating new execution engines into the daemon and hardware kernels within the FPGA hardware implementation.

The next step is the *FPGA manager*, which handles the FPGA hardware at the Operating System (OS) level, which can be based on CYNQ RT (from the previous section) or

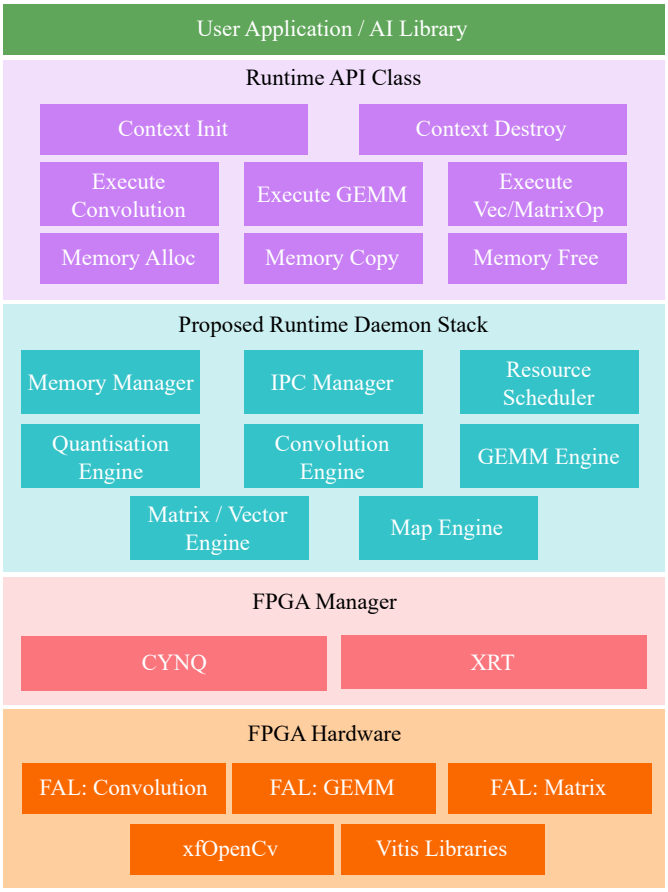


Figure 6.2: Proposed architecture stack. Includes one existing FPGA manager, a daemon and an API to perform the most common AI operations

XRT. It provides the device drivers, configuration, synchronisation and memory allocation mechanisms for FPGA-specific transactions, including support for the AXI protocol. This layer can load the FPGA configuration file with the implemented design, allowing reconfiguration on demand.

One of the contributions of this work is a *Runtime Daemon Stack* that sits on top of the FPGA manager, with modules that administer memory at the user-space level and facilitate inter-process communication (IPC). It also features a resource scheduler to manage and distribute resources when multiple applications attempt to access the same accelerator. The daemon concludes with the operation engines for the most common AI operations: convolution, matrix multiplication (GEMM), element-wise vector/matrix operations, and a map engine that performs a unary function on each vector/matrix element. These engines connect to pre-configured hardware on the FPGA, primarily based on the FAL. For the IPC and Memory Management, POSIX Semaphores and Shared Memory mechanisms are used to control the memory access and avoid race conditions amongst processes. The resource scheduler (not considered in this work during experimentation) is based on the Completely Fair Scheduler (CFS) proposed in the Linux kernel [153], which allows scheduling resources based on priorities. This can be particularly important when dealing with multiple processes that have different priority levels.

The C/C++ *Runtime API class* is parallel to the daemon, which offers the user's application methods to perform memory transactions, execute the engines and initialise or finalise the context (connections to the IPC and memory mechanisms). Figure 6.3 shows the transactions between the user-space application and the daemon. In this case, a delegate acts as a library for an AI framework or as a standalone application that performs initialisation, memory allocation/deallocation, memory writes/reads, and orders the execution of the accelerators. This workflow is inspired by other similar runtime libraries, such as CUDA and Vitis Libraries, which require memory allocation with specific details, including dimensions, data types, and memory types. It then performs operations in these memory regions from the host code for later execution of the accelerated kernel.

Unlike other work, the proposed memory allocation also performs arbitrary-precision fixed-point quantisation/de-quantisation according to the pre-built accelerator configured in the FPGA, allowing optimisation in communication, device-memory storage and implementation design resource consumption. This is achieved through a quantisation/de-quantisation engine, which receives the target precision and is attached to the memory transactions that perform the operations as the data flow, utilising auto-vectorisation (SIMD). In other approaches, data conversions occur inside the accelerator, consuming bandwidth, or are performed through the Vitis HLS arbitrary-precision library, which is not optimised for large data volumes.

At the hardware implementation, in the Vitis Libraries, the data type is fixed to 8-bit integers for convolutions and floating-point (16 and 32-bit precision) for matrix operations, without having room for optimisation in terms of memory and design size. In contrast, the approach followed in this work allows synthesising accelerators for arbitrary-precision data types, suitable for adequate design sizes and data volumes, according to the user's numerical precision requirements. Moreover, it can implement heterogeneous precision accelerators that support different fixed-point precisions, allowing support for non-uniform quantisations across the code.

Altogether, the proposal yields an integral solution that integrates the most common acceleration kernels used in DL inference, allowing for reuse in other applications while leveraging FPGA resource management and ease of use. This enables users to achieve acceleration in AI computations without concern for hardware implementation details. This marks a significant contribution to the democratisation of FPGA use in AI and HPC, allowing non-FPGA experts to utilise this hardware for offloading computations.

6.4 Experimental Results

The key impact of the framework lies in measuring the execution times when running the most common algorithms in AI and the impact of the different parts of the code that may cause degradation compared to standalone applications with managed FPGA, where the FPGA transactions are fully controlled internally. This work uses AMD Vivado and

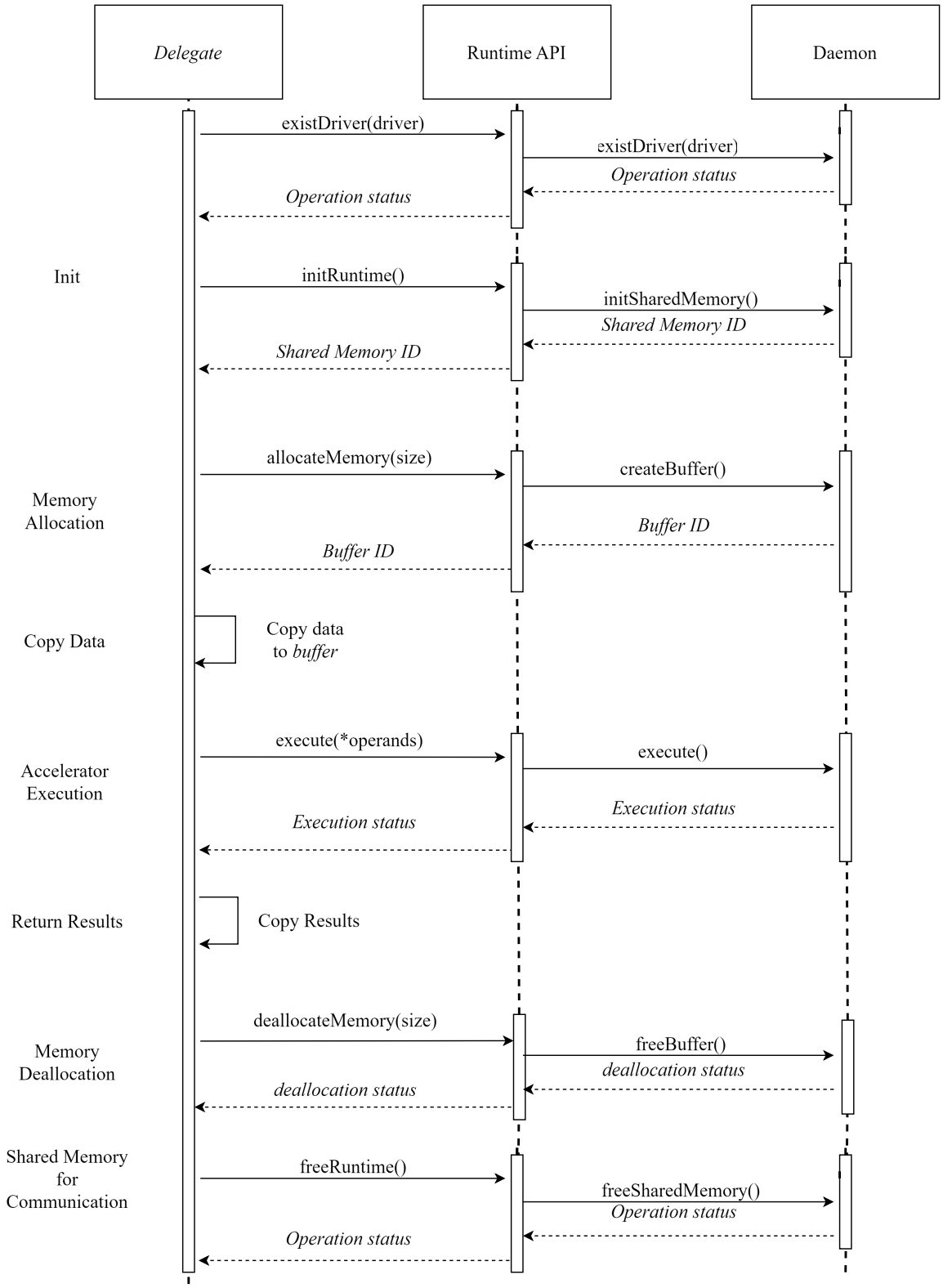


Figure 6.3: Transactions between a user-space application or library (a.k.a. delegate), the Runtime API Class and the Runtime Daemon.

Vitis 2022.1 to synthesise the FPGA hardware designs, a certified Ubuntu 22.04 on an AMD Kria KV260 development kit for evaluation purposes. For the execution times, the following experiments running at 200 MHz clock speed are performed:

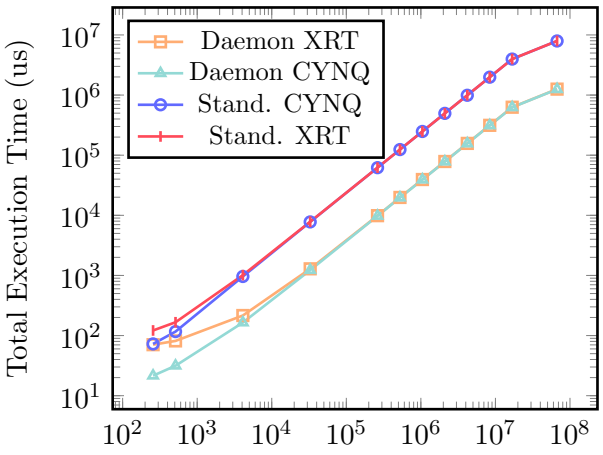
- **Convolution 2D**: performs a 3×3 convolution on a single-channel 8-bit image, useful for measuring the impact of 8-bit quantised networks and image processing applications (from xfOpenCV).
- **Matrix Multiplication**: performs a 16-bit matrix multiplication of two matrices, assuming the second matrix is transposed, using the standard matrix multiplication algorithm (from FAL).
- **Matrix Element-Wise Addition/Multiplication**: performs a 16-bit matrix element-wise addition and multiplication, using a signed 16-bit fixed-point with a 6-bit integer part from (FAL).
- **Multi-Layer Perceptron Auto-Encoder**: integrates the matrix operations aforementioned to quantify the total inference running time (from AxC Executer).

These experiments help to contrast the performance impact of each runtime backend (CYNQ RT and XRT), the quantisation/de-quantisation overhead (standalone vs our framework), and the quantisation/de-quantisation method (Vitis HLS/our implementation). For all the experiments, the standalone applications use the Vitis HLS quantisation/de-quantisation method, whereas the applications based on our framework will use our quantisation/de-quantisation engine.

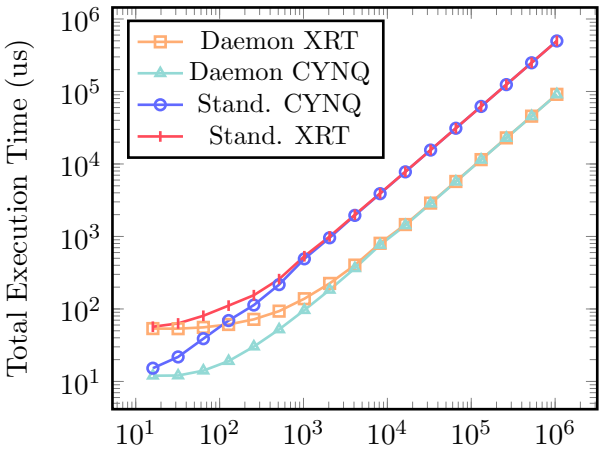
6.4.1 Convolution 2D

This experiment performs image filtering with a 3×3 kernel on 8-bit single-channel images using the spatial convolution algorithm. This is relevant for 8-bit quantised networks and image processing applications. In this case, the kernel selection does not affect the overall execution time or the resources and is fixed to an edge detection filter. Moreover, no quantisation/de-quantisation is performed during the computations.

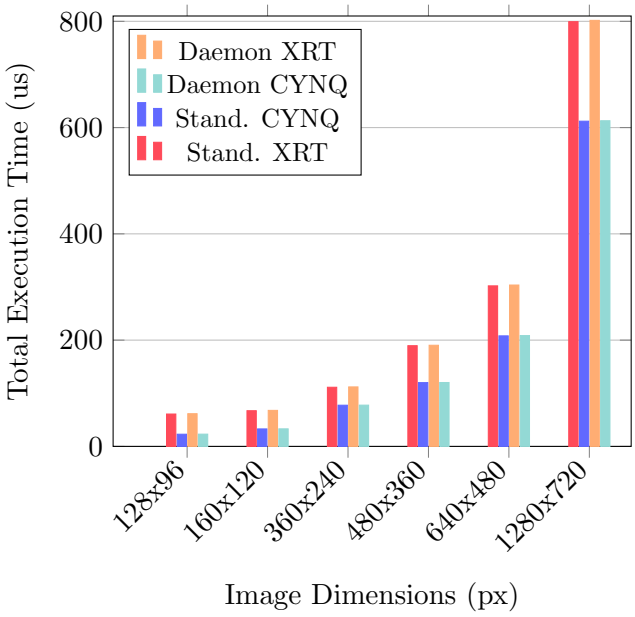
Figure 6.4c shows the execution times when performing the convolution on standalone applications with the FPGA manager included (Stand. CYNQ and Stand. XRT) and on apps based on the CYNQ framework (Daemon XRT and Daemon CYNQ) for 128×96 to 1280×720 image dimensions. For all cases, those applications that use CYNQ RT show superiority over those based on XRT, highlighting that CYNQ RT is a good choice for the platform used for experimentation due to its implementation simplicity. Moreover, it is possible to observe that, in the absence of quantisation, the performance of the daemon is quite close to a standalone application, suggesting that the choice for shared-memory and inter-process communication has a negligible impact on performance.



(a) Matrix Multiplication



(b) Matrix Element-Wise



(c) Convolution 2D

Figure 6.4: Execution times of the experiments when using standalone apps on top of the FPGA managers (Stand. XRT and Stand. CYNQ) and applications running based on the Runtime API Class with the daemon on top of XRT or CYNQ (Daemon XRT and Daemon CYNQ). The execution times are the average after 1000 runs.

6.4.2 Matrix Operations

These experiments perform multiplications and element-wise operations on matrices. In contrast to the Convolution 2D, these operations involve quantisation from single-precision floating-point (32-bit float) to 16-bit fixed-point with 6-bit integers. In this case, the standalone applications use the Vitis Library for quantisation, and those based on the

Table 6.1: Multi-Layer Perceptron Execution Times while processing 100 samples: XRT (X) and CYNQ RT (C) applications use (de)quantisation using the Vitis HLS library. Daemon applications use the our (de)quantisation implementation. All times are in milliseconds and taking the worst-case scenario.

Configuration	Standalone (X))	Standalone (C)	Daemon (X)	Daemon (C)
Mem. Alloc	0.75 ± 0.18	0.75 ± 0.34	0.75 ± 0.18	0.75 ± 0.34
Quant.	4.1 ± 0.4	4.1 ± 0.5	0.8 ± 0.1	0.8 ± 0.1
Exec. Dense	18.32 ± 0.03	14.8 ± 0.06	18.33 ± 0.03	14.9 ± 0.06
Exec. Add.	11.09 ± 1.2	4.78 ± 0.44	11.09 ± 1.2	4.78 ± 0.44
Exec. Mult.	11.03 ± 1.16	4.77 ± 0.40	11.02 ± 1.16	4.78 ± 0.40
Dequant.	1.29 ± 0.04	1.29 ± 0.03	0.097 ± 0.2	0.097 ± 0.2
Critical Path	**24.46**	**20.94**	**19.977**	**16.547**

CYNQ framework use the custom quantisation engine. Figure 6.4b and 6.4a illustrate the results obtained for each use case for multiple sizes of input operands. In the case of Figure 6.4b, the element-wise operation is a matrix addition, with a total number of elements that corresponds to the product of the matrix dimensions times 2, whereas Figure 6.4a, the matrix multiplication has a number of input elements which is the sum of the elements from the two input matrices. For the element-wise multiplication, the runtime is equivalent to the element-wise addition.

In both experiments, the results converge for standalone and framework-based applications when the number of operands tends to be large. This is due to the convergence between XRT and CYNQ RT in runtime when dealing with large volumes of data. Moreover, a gap exists between the standalone and daemon-based applications, making the framework-based application approximately 6× faster. This is due to the quantisation method, as our engine is faster than the quantisation based on the Vitis HLS library used by the standalone applications, thanks to SIMD auto-vectorisation.

6.4.3 Multi-Layer Perceptron Auto-Encoder

The last experiment is a Multi-Layer Perceptron Auto-Encoder. It is based on the AxC Executer framework running a pre-trained model for audio anomaly detection used in the MLPerf Tiny Benchmark [95], with six dense layers, batch normalisation decomposed as matrix element-wise multiplications and additions. It employs a 16-bit fixed-point Zero-Shot quantisation with a 6-bit integer part [62], resulting in minimal impact on the final model performance [4]. In this case, it utilises the same quantisation techniques and accelerators as those in Section 6.4.2 experiments. Table 6.1 shows the results for different parts of the code regarding the memory allocation (Mem. Alloc), quantisation (Quant.), the execution of the dense, element-wise addition and multiplication layers (Exec. Dense, Exec. Add., and Exec. Mult., respectively), which run on the same engines used in Section 6.4.2, and the de-quantisation (Dequant.).

According to the results, the experiments that run on top of the CYNQ framework (Daemon (X) and Daemon (C)) demonstrate overall superiority against those that run with the FPGA Manager directly and use Vitis HLS for quantisation (Standalone (X) and Standalone (C)). The execution times of the operations and memory allocations for a single layer are similar between implementations in the worst-case scenario, with a slight increase of around 100 microseconds. However, the quantisation and de-quantisation operations significantly impact the final overall results. The CYNQ framework is 5.12× faster quantising and 13.30× faster de-quantising for the worst-case scenario in the whole network, impacting the overall critical path time (allocation, quantisation, dense execution and de-quantisation), where the CYNQ framework integrated into AxC Executer is 1.22× faster than using the AxC Executer with Vitis HLS-based quantisation and the FPGA Manager integrated into it. These results align with the findings from Section 6.4.2, where a performance gap is observed between the standalone and framework-based applications. Likewise, it is worth noting that the model's numerical performance remains unaffected.

6.5 Final Remarks

This chapter briefly analysed the XRT and PYNQ runtime libraries, enumerating their weak points and how they affect the overall performance. It has also proposed CYNQ RT, an agnostic and friendly runtime library for interfacing software applications with the FPGA, which has demonstrated superiority in all cases studied for image processing and AI, easing software development with an agnostic and friendly C++ API similar to the one from PYNQ. CYNQ RT removes the Python interpretation overhead. Moreover, the performance is better than XRT's, thanks to its hybrid implementation, which utilises the Linux FPGA Manager and Linux Device Drivers (MMIO) in conjunction with XRT's memory management. It enables low-latency designs to unlock more performance and reduce accelerator startup latency, eliminating unnecessary complexity and keeping the design simple and direct.

This work also contributed with the democratisation of the FPGA, presenting a user-friendly ecosystem for FPGA-based AI acceleration that combines a stack of *pre-built hardware accelerators* for FPGAs, a *daemon to manage the FPGA resources and memory allocation*, and a *compute-centric runtime library for user application development*. This is part of a proposal to ease the development of FPGA-accelerated applications, which isolate application development from hardware implementation, facilitating a focus on the application rather than coding hardware accelerators using a computation-oriented approach.

The CYNQ framework integrates quantisation and de-quantisation engines that demonstrate superiority over the tools used by Vitis HLS, yielding gains of 5.12× faster quantisation and 13.30× faster de-quantisation in an actual MLP-based autoencoder model inference. Moreover, the daemon module benefits from FPGA Managers like XRT and CYNQ RT, keeping execution and allocation times close to standalone applications with

managed FPGA. It also integrates shared memory, scheduling and IPC mechanisms that allow user applications to have zero-copy memory transfers to the daemon, keeping the impact of the inter-process data communication low.

As a work leftover, CYNQ RT can potentially support other platforms not supported by XRT and PYNQ, such as XDMA-capable PCIe FPGA cards like some variants of AMD Artix-7, which are usually considered low-end FPGAs due to their resource limitations, and FPGAs from other vendors. Also, the CYNQ framework resource scheduler and inter-process communication can be improved. Since the CYNQ framework is modular, it is possible to integrate additional accelerators and perform benchmarks, making the ecosystem a promising alternative and well-suited for AI inference acceleration with FPGAs, as reflected in future work. The CYNQ repo is available and open-source.

Related Publications:

- Luis G. León-Vega and Erick Obregón-Fonseca and Jorge Castro-Godínez, "A User-Friendly Ecosystem for AI FPGA-based Accelerators," in *2024 IEEE International Conference on Omni Layer Intelligent Systems (COINS)*, 2024
- Luis G. León-Vega and Diego Avila-Torres and Jorge Castro-Godínez, "CYNQ: Speeding Up FPGA Applications with Simplicity," in *2024 31st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2024

Repositories:

- León-Vega, L., Ávila-Torres, D., Castro-Godínez, J., *CYNQ (v0.2)*, 2024. [Online]. Available: https://github.com/ECASLab/cynq
- E. Obregón-Fonseca, L. León-Vega, and J. Castro-Godínez, *AxC Kernel Delegate Interface*, 2022. [Online]. Available: https://gitlab.com/ecas-lab-tec/approximate-flexible-acceleration-ml/axc-kernel-delegate-interface

Chapter 7

Insights for the Next Generation of AI Computers

Previous chapters of this thesis explored techniques to account for energy based on executed instructions and FPGA-based AI inference acceleration. Chapter 5 finished with measurements of the best architecture achieved for LLM inference on a high-end FPGA, where the GPU is still faster. However, there is still room for optimisation, particularly with approximate computing, arbitrary precision, and sparse matrix operations.

Chapter 6 addressed some of the most common issues in computing using FPGAs, particularly due to the closed-sourceness of the tools, generalisation, and difficulties in FPGA development. This work has also proposed an ecosystem of low-latency runtime libraries (CYNQ RT), a framework with an FPGA administrator (CYNQ framework), and a compute-centric approach to address these issues.

Still, the possibility of reaching more efficiency and productivity than GPUs and FPGAs is under discussion. Hence, this chapter focuses on insights for the next generation of AI computers, which are AI-centric and will improve AI training and inference with better energy efficiency and performance than the architectures currently used for these tasks.

7.1 Architectures for AI

The design, optimisation and implementation of architectures to accelerate AI development and inference is a hot topic in industry and academia. Research work oscillates between designing and optimising architectures to speed up AI tasks while maintaining lower power consumption. Within existing architectures, CPUs, GPUs, and ASICs are the most commonly used, while FPGAs are still emerging with promising results. There is also research on the use of cooperative heterogeneous computing with Processing in Memory (PIM) and Near-Data-Processing (NDP), where memory becomes an active processing component that helps perform data transformations before and after they arrive

from other larger processing components, such as CPUs and GPUs [26].

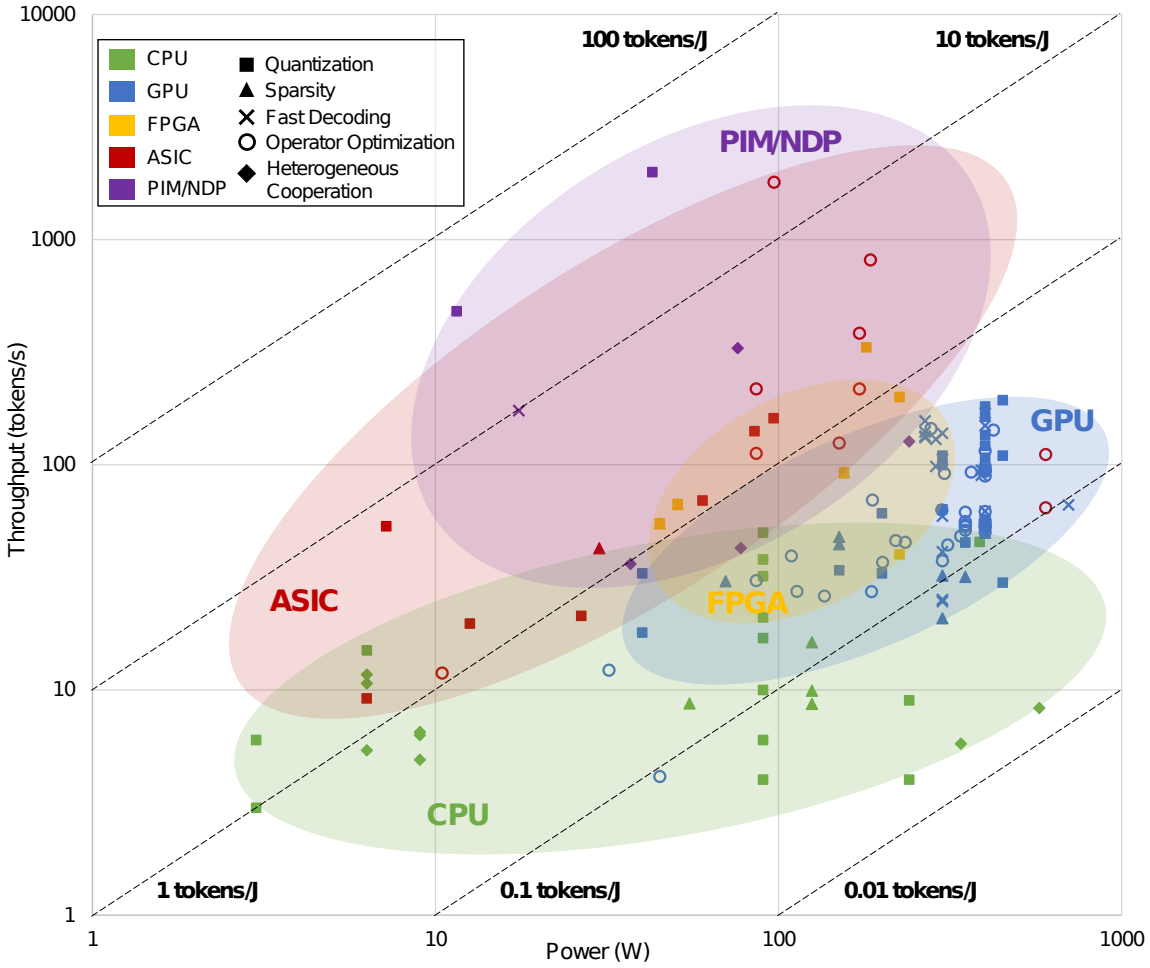


Figure 7.1: LLM (7 billion parameters) decode stage throughput (batch size 1) vs power on different platforms with different optimisation methods. Retrieved from [26].

Figure 7.1 shows a plot from [26], illustrating the throughput and power consumption for an LLM of 7 billion parameters of multiple architectures from academia and industry. In architecture optimisation, it is possible to observe that CPUs reach a similar throughput as GPUs and FPGAs at the same energy budget. FPGAs have higher throughput than GPUs with lower power consumption, thanks to quantisation, which firmly supports the research from Chapter 5. This finding suggests that FPGAs can achieve GPU performance with more optimisations. Beyond FPGAs, ASICs excel in energy efficiency, offering higher throughput at the same power consumption. However, the issue with ASICs is that they are not reconfigurable and cannot be upgraded as AI evolves, making them obsolete over time.

PIM/NDP offers a more promising approach: equipping the memory with processing when the operands are either loaded or stored. This allows for online quantisation techniques, mappings (such as activations) and other operations depending on the implementation [155]. PIM/NDP devices are often insufficient for an entire LLM computation and require cooperation with other, larger processing architectures, resulting in cooperative

heterogeneous computing. According to Figure 7.1, they are the most efficient architecture so far, suggesting a possible path for the next generation of AI computers, where all the processing components can work together to enhance the energy efficiency of AI processing.

7.2 Cooperative Heterogeneous Computing Paradigms

Cooperative Heterogeneous Computing (CHC) is a paradigm that involves utilising, orchestrating, and optimising multiple architectures to collaborate in processing complex workloads [156]. Some of the most popular combinations of CHC involve CPU + GPU, CPU + GPU + FPGA or CPU + GPU + ASIC [155], as illustrated in Figure 7.2. CHC presents several challenges when dealing with multiple architectures, some of which are mentioned below:

- **Workload Partitioning**: Selecting *where* a piece of code should execute is crucial to optimise runtime, energy consumption, and system occupancy. Poor partitioning can overload one architecture while leaving others underutilised, or cause unbalanced energy usage.
- **Workload Scheduling**: Deciding *when* and *on which* device a workload fragment should be executed is critical to maximising timeline overlap and minimising idle periods. Improper scheduling can prevent adequate overlap, leading to increased overall execution time.
- **Architecture Consumption Balancing**: CPUs are often less energy-efficient than other architectures when processing workloads. Although GPUs typically complete tasks faster, balancing execution across architectures can reduce energy consumption. For instance, adjusting the power profile of the GPUs allows for longer execution at lower power, which can be more energy-efficient. A similar strategy can be applied in the opposite direction, depending on the characteristics of the workload.
- **Memory and I/O Communication**: Multi-architecture execution often requires transferring data between fragments residing on different architectures. If I/O communication is slow and cannot overlap with computation, or if data transfer times significantly exceed compute times (I/O-bound scenarios), the overall execution time will increase.
- **Software Complexity**: Software frameworks and libraries introduce additional challenges when managing multiple architectures, particularly regarding programming models, specialised APIs, and support for hardware-specific optimisations.

Mitigating the challenges above accelerates workloads, balances energy consumption, and maximises the utilisation of system components. The extent of these improvements in

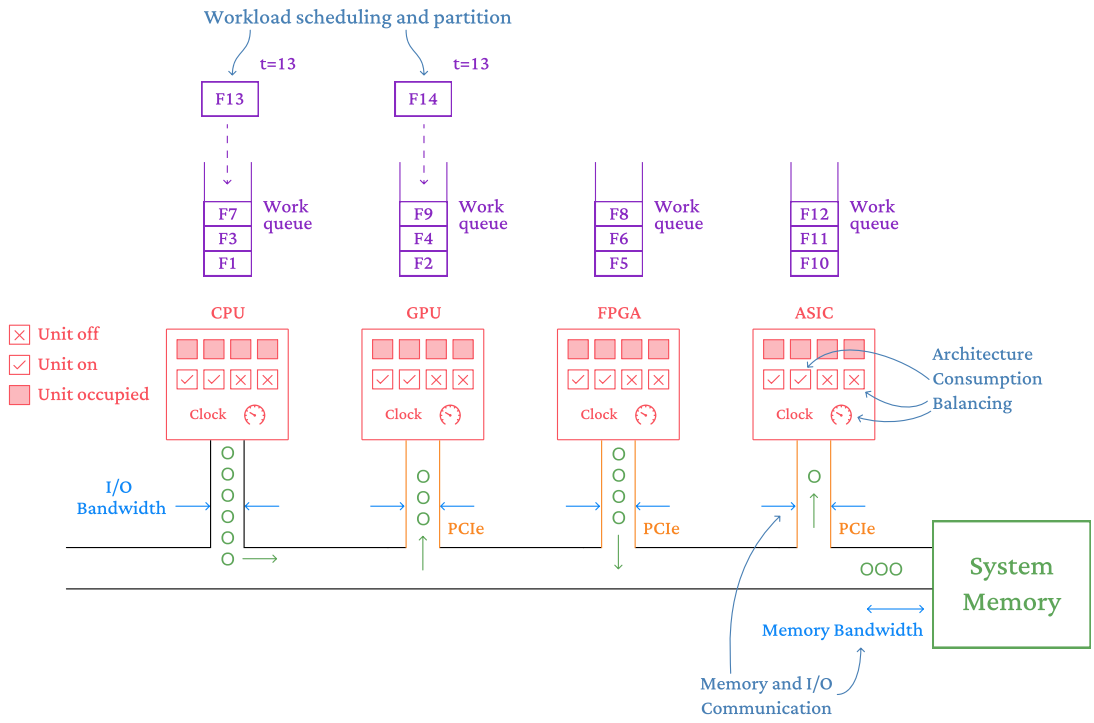


Figure 7.2: Heterogeneous computing architecture with multiple processing systems, such as CPU, GPU, FPGA and ASIC. It also illustrates multiple critical points for design, such as the Workload Scheduling and Partition, Consumption Balancing and Communication.

accelerated workloads depends on the nature of enhancements within the system where CHC is applied. These enhancements can be classified into two categories: *compute-oriented* and *memory-oriented improvements* [26].

Compute-oriented improvements aim to boost pure computational performance, benefiting compute-bound workloads characterised by high arithmetic intensity, where each datum is used multiple times in compute operations. Conversely, memory-oriented improvements address bottlenecks arising from low arithmetic intensity, which typically lead to memory-bound workloads. Within the improvements, several fronts can be distinguished, ranging from local architectural enhancements to broader CHC strategies. Examples of local architecture improvements include the use of AMX extensions in modern Intel CPUs [21] and DeepSeek's DeepGEMM optimisations for NVIDIA Hopper GPUs [27]. At the CHC level, techniques have been developed to schedule layer offloading dynamically across CPUs and GPUs [157]. Furthermore, PIM offers an alternative by absorbing low-arithmetic-intensity operations near the memory, thereby reducing data movement overhead. Recent research shows that PIM and NDP architectures can achieve over 3× speedups compared to GPU-only implementations in GPT workloads [158]–[160].

Memory-oriented improvements, on the other hand, focus on mitigating memory bot-

tlenecks by either computing closer to memory or increasing memory access bandwidth. PIM/NDP approaches accelerate operations with low arithmetic intensity, such as activations, matrix element-wise operations, and quantization [155]. Increasing memory bandwidth involves two closely linked strategies: (1) expanding the memory subsystem bandwidth and (2) enhancing interconnect bandwidth. High-Bandwidth Memory (HBM) technologies have revolutionised memory design by introducing 3D-stacked memory, widening memory buses, and inspiring new PIM/NDP techniques [161]. However, increasing memory bandwidth must be complemented by improvements in interconnect bandwidth to prevent data exchange bottlenecks. NVIDIA NVLink [162] provides a high-speed interconnect fabric that enables faster GPU-to-GPU communication, outperforming traditional PCIe. Similarly, Compute Express Link (CXL) builds upon PCIe to reduce communication overhead and increase bandwidth between CPUs and peripherals, with early applications notably involving FPGAs [163].

Another promising architecture within the CHC landscape involves RISC-V-based accelerators. Research in this area includes designing accelerators with RISC-V clusters augmented by vector instruction set extensions [164], [165] and integrated DNN-specific hardware accelerators [164], [166]. More recent work leverages the Parallel Ultra-Low Power (PULP) platform, an open-source template for developing energy-efficient RISC-V-based accelerators.

7.3 Reconfigurable Computing

Throughout this paper, FPGAs have been widely discussed as reconfigurable devices capable of accelerating workloads due to their ability to adapt the hardware to behave like any architecture, making them an example of *reconfigurable computing* devices.

In general, *Reconfigurable Computing* can adapt the implementation to become more efficient given a particular workload. They can modify internal connections, math operators, and data types to increase throughput under workload requirements. Often, they tend to have a better trade-off between energy consumption and computational performance [167].

Within the current state-of-the-art (SOTA) in High-Performance Reconfigurable Computing (HPRC), the two primary implementations are based on FPGAs and Coarse-Grained Reconfigurable Arrays (CGRAs) [155]. CGRAs offer a lower degree of reconfigurability compared to FPGAs. Still, they achieve higher efficiency by organising multi-dimensional arrays of PEs, each equipped with Arithmetic Logic Units (ALUs) and other specialised execution units. By operating at a higher hardware abstraction level, CGRAs mitigate the hardware complexity and programming overheads typically associated with fine-grained alternatives, such as FPGAs [168]. Furthermore, CGRAs have demonstrated superior energy efficiency, achieving lower power consumption per unit of throughput compared to NVIDIA's V100 GPUs at the time of the study [169].

RipTide represents an innovative, co-designed CGRA architecture paired with a compiler,

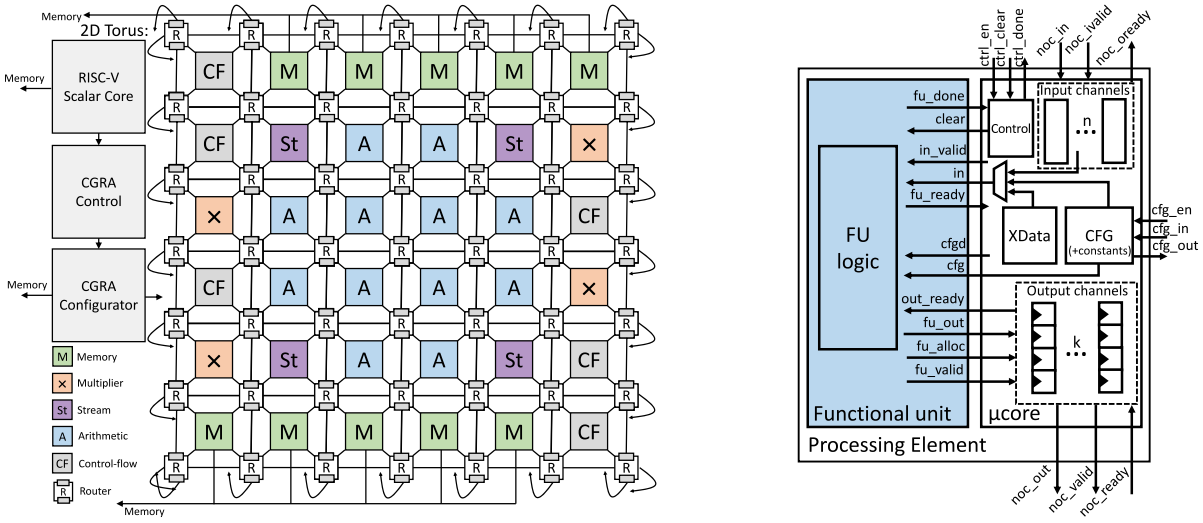


Figure 7.3: RipTide CGRA Fabric and components. This CGRA interconnects several PEs using a reconfigurable 2D torus interconnection, where each PE is specialised in either memory, arithmetic, control flow or data exchange. Adapted from [170].

achieving high programmability and energy efficiency. Unlike traditional CGRAs, which often require hand-crafted assembly code and are restricted to specific code patterns, RipTide supports arbitrary control flow and memory access patterns without depending on costly hardware mechanisms. This flexibility enables the efficient execution of a broad range of applications with complex control structures and irregular memory access patterns, such as sparse linear algebra operations. RipTide delivers substantial energy savings, executing code with an order-of-magnitude lower energy consumption compared to traditional von Neumann cores. It achieves a 25% performance improvement over contemporary CGRAs at the time of its release, and up to 490× greater performance-per-watt compared to the Cortex-M3 (0.13 MOPS/mW) [170]. Other notable CGRA implementations built on similar foundations include REVAMP [171] and SNAFU [172]. Meanwhile, X-CGRA introduces an innovative approach by combining CGRA reconfigurability with Approximate Computing techniques to further boost performance [173].

Beyond fully reconfigurable architectures, several approaches explore partial reconfigurability by extending traditional processing architectures. The Framework for a Dynamically Reconfigurable Accelerator (FDRA) is an open-source platform designed to facilitate the exploration and development of heterogeneous systems-on-chip (SoCs), integrating a RISC-V processor with a dynamically reconfigurable accelerator (DRA) based on CGRA principles [174]. FDRA supports multi-level parallelism (loop-level, instruction-level, and task-level) to improve computational efficiency and application flexibility, achieving modest performance gains compared to SNAFU [172]. Another notable architecture, FlexBex, incorporates a small embedded FPGA (eFPGA) within the Ibex RISC-V core, enabling dynamic reconfiguration of custom instructions at runtime [175]. This design empowers developers to implement specialised operations directly into the processor pipeline,

enhancing performance for targeted tasks without relying on external accelerators. Additional research explores the use of the MOLEN paradigm, employing micro-codes to configure reconfigurable regions [176], as well as the integration of Approximate Computing techniques into reconfigurable RISC-V platforms [177].

7.4 Route for Energy-Driven AI Processing Architectures

Hardware acceleration for AI-related workloads entails addressing several challenges, particularly those arising from compute- and memory-bound operations. Compute-bound workloads are commonly tackled using massive GPU parallelism or FPGAs. However, memory-bound workloads, which involve relatively few operations per data element and rely heavily on memory bandwidth or I/O communication, often render traditional compute-bound solutions underutilised. Therefore, it is necessary to develop solutions capable of handling both fronts: workloads requiring high computational throughput and those demanding high memory bandwidth.

Modern CPUs, GPUs, and ASICs have integrated *low-precision arithmetic* units to accelerate compute-bound tasks and compress AI models. Nonetheless, low-precision arithmetic alone can represent a *suboptimal solution*, as AI models continuously evolve, pushing the boundaries of quantisation, sparsity, and precision. Alternative numerical representations, such as the POSIT format [62], require different arithmetic unit designs. However, due to hardware constraints, their adoption often necessitates complex software optimisations, forcing traditional architectures to support new formats inefficiently.

Memory-bound limitations have been partially addressed through the development of high-speed dynamic memories, such as HBM [161], and fast interconnect technologies like NVLink [162]. Nevertheless, memory bandwidth remains a bottleneck relative to computational capabilities.

FPGAs remain a popular option among non-traditional paradigms, although their adoption is often hindered by the complex design expertise required, as discussed in Chapter 6.

7.4.1 Compute-Bound Track

Addressing compute-bound workloads requires adequate numerical support to maintain result quality while tolerating acceptable computational error rates. Although current platforms support low-precision computations, the hardware is often underutilised. A promising approach is *execution-level reconfigurability*, which enables execution units to adapt to different numerical precisions and allows optimisations such as approximate computing. Such reconfigurability has been explored in RISC-V- and CGRA-based architectures [174], [175], [177].

Furthermore, various architectures exhibit natural affinities to specific workload types: GPUs excel at dense, coalesced memory access patterns, while CPUs handle sparse, irregular access patterns more effectively. ASICs, including TPUs and NPUs, offer superior performance-per-watt for matrix multiplication through silicon specialisation. FPGAs also present a compelling alternative for low-latency tasks [155]. *Cooperative Heterogeneous Computing* (CHC) combined with dynamic power profiling presents a promising strategy for optimising the performance-per-watt trade-offs across different architectures.

Consequently, the path to address the computational demands of AI workloads lies not in a single architecture, but rather in *systems combining multiple specialised architectures*, capable of adapting numerical precision to model requirements through built-in reconfigurability. Figure 7.4 illustrates the introduction of a reconfigurable device, composed of multiple processing elements with Reconfigurable Execution Blocks (REBS) that allow implementing custom execution micro-architectures to fit specific workload requirements. In the example, it can extend the RISC-V architecture to adopt a new instruction set (ISA) to process approximate vector addition of an N/H-element vector with H-bit precision operators.

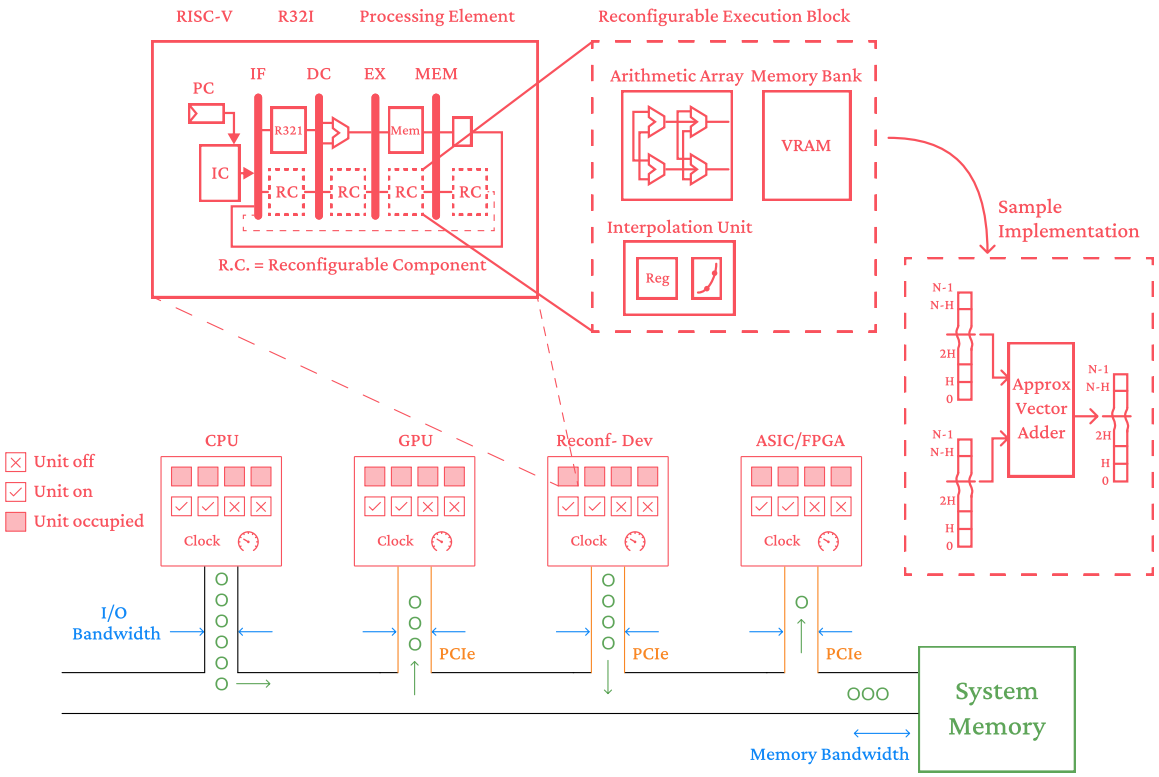


Figure 7.4: Introduction of reconfigurability in the CHC architecture. The reconfigurable devices (Reconf-Dev) are equipped with Reconfigurable Execution Blocks inspired by the CGRA architecture. They enable the CHC to be adaptable and implement optimised micro-architectures for a specific workload.

7.4.2 Memory-Bound Track

Memory-bound computations frequently occur in low-arithmetic-intensity tasks, such as activations, matrix additions, and layer-wise operations. In these cases, communication times dominate and cannot be effectively overlapped with computation, resulting in significant performance penalties.

As discussed in Chapter 5, implementing *dataflow-based layer fusion with FPGAs* within CHC systems effectively improves performance and reduces computation time. Moreover, PIM and NDC technologies embedded into memory modules (e.g., HBM) provide an effective strategy to offload low-arithmetic-intensity workloads, bringing computation closer to the data. Adding reconfigurability to PIM/NDC architectures enhances support for variable or arbitrary precision computations, enabling further optimisation of memory-bound and hybrid workloads. Figure 7.5 illustrates the integration of reconfigurable PIM into the memory module, allowing adaptability for arbitrary-precision computations.

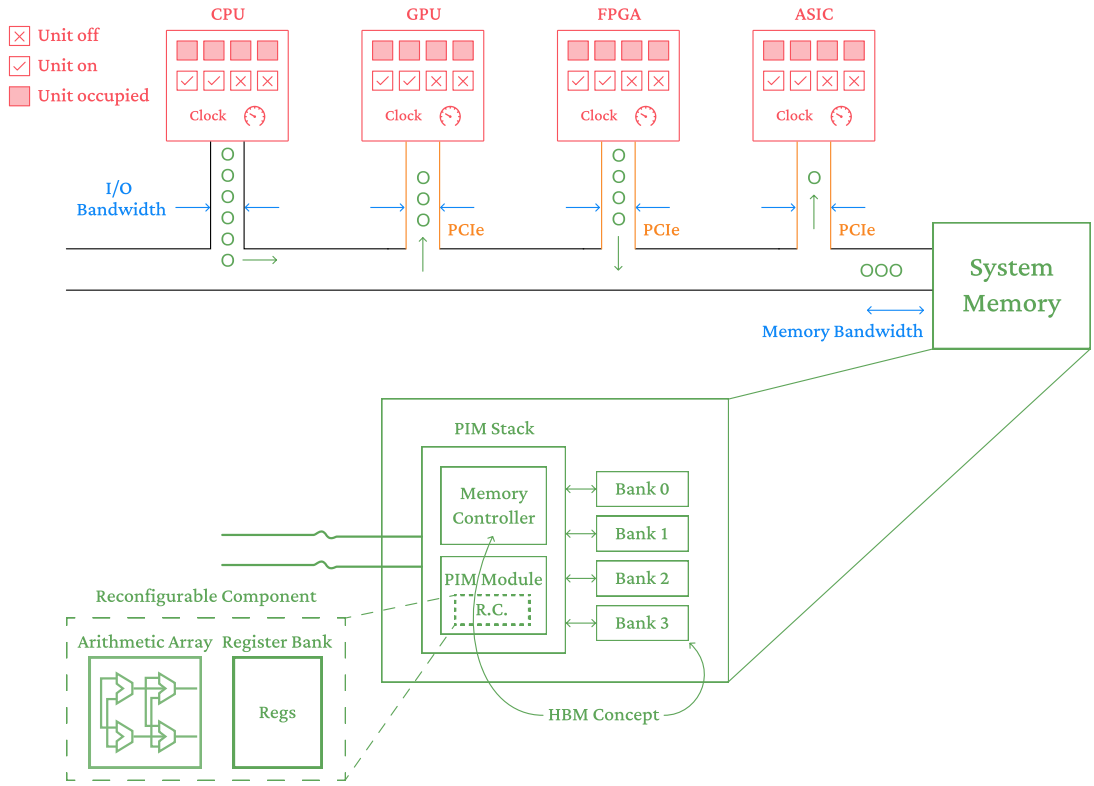


Figure 7.5: Proposed system memory module to tackle memory-bound workloads on CHC. The memory module is equipped with multiple banks, following the idea of HBM. In addition, it is equipped with a reconfigurable PIM module to adapt the memory modules to perform computation at any numerical precision, allowing arbitrary precision quantisation, element-wise operations, and element mapping.

7.4.3 Reconfigurability on Cooperative Heterogeneous Computing

Heterogeneous Computing for AI demands awareness of workload characteristics, energy consumption, and numerical precision requirements. Effective CHC systems must combine specialised computational units with scheduling mechanisms that optimise workload distribution while regulating energy use.

Leveraging reconfigurability to CHC involves multiple levels, from execution units to memory, and shows potential for AI workload demands. It also facilitates the incorporation of approximate computing techniques, enabling a flexible adjustment of the Quality-of-Result (QoR) versus energy consumption trade-off and alternative numerical formats, such as block floating-point computation [178].

Within the key advantages, it is possible to visualise:

- Approaches both compute-bound and memory-bound workload challenges.
- Reduces power consumption for low-arithmetic-intensity operations by leveraging in-memory processing.
- Optimises energy consumption through energy-aware workload scheduling.
- Offers fine-grained control over numerical precision via reconfigurable execution units.
- Enhances adaptability to AI model evolution, reducing hardware obsolescence.
- Increases optimisation opportunities through approximate computing and architectural flexibility.

Nevertheless, it also involves a series of challenges and requirements:

- Development of reconfigurable architectures, particularly CGRA-like structures with coarse granularity.
- Integration of traditional, reconfigurable, and PIM/NDC-based computing techniques.
- Implement energy-aware workload scheduling to select architectures based on task affinity and power optimisation.
- Deployment of dynamic power management, including clock gating and power-down strategies for idle components.
- Creation of user- and developer-friendly frameworks to simplify software development on reconfigurable CHC platforms.

Reconfigurable CHC architectures offer a promising solution to the evolving challenges posed by AI workloads. By addressing both compute- and memory-bound limitations and enabling systems to adapt over time, they provide a pathway toward extending hardware lifespans and maintaining relevance amid the rapid pace of AI innovation. This thesis concludes the research in this direction, providing a foundation for future contributions.

Chapter 8

Conclusions

This chapter concludes this dissertation with a summary of the work presented, and it provides promising future work directions.

Dissertation Summary

Throughout this dissertation, two critical challenges in sustainable high-performance computing have been addressed: 1) the accurate energy accounting of computational workloads, and 2) the development of energy-efficient AI accelerators utilising FPGAs.

First, this work proposed EfiMon, a tool for fine-grained energy consumption analysis, marking a significant advancement in the ability to monitor and model energy usage in shared computing environments. It extracts data from multiple sources and combines them to develop new mathematical models for estimating energy consumption. Unlike traditional methods that require execution isolation, EfiMon enables energy estimation at the process level while maintaining low error margins, particularly on CPU and GPU architectures. This contribution lays the groundwork for more energy-conscious resource scheduling and system management, essential for the evolution of next-generation supercomputers.

Within Efimon, this work has also presented a mathematical model for estimating the energy consumption of a running software process, through the analysis of its overall CPU utilisation and executed instructions, which resulted to be more accurate than traditional methods that were based only on the utilisation, ignoring the nature of the instructions dispatched to the CPU. Together with EfiMon, this work achieved energy accounting with a relative error of 1.93% for the CPU and 9.7% for the GPU, without the need for execution isolation, thereby pioneering the use of instructions to enhance energy estimations.

Second, this dissertation explored a more sustainable solution for AI inference execution using FPGAs. The Flexible Accelerator Library (FAL), a novel framework for the automatic generation of FPGA-based AI accelerators, was presented. This library supports

parameterisation in operand size, numerical precision, approximate computing techniques, and structural reuse, allowing its adaptability to diverse environments and requirements in terms of performance and resource constraints. Experimental evaluations have demonstrated that adopting flexible, multiple algorithms and approximate computing strategies can significantly reduce hardware resource consumption with minimal impact on controlled accuracy. This study demonstrates an 18.93% reduction in resource consumption, accompanied by a 9.6% degradation in AI model accuracy, using a LeNet 5. For MobileNet v2, the resource reduction was approximately 20%, accompanied by an accuracy improvement of 16.6% due to healthy numerical disturbances in the softmax layer. This work extends the application of FPGAs into fields traditionally dominated by GPGPUs, offering a more sustainable alternative.

This exploration was extended by studying the use of FPGA-based architectures for Large Language Model (LLM) inference. It has been shown that with careful design optimisations, FPGAs can approach or even surpass GPU performance for specific tasks, particularly when leveraging techniques such as spatial acceleration and precision customisation. This work achieved a speedup ranging from 1.37× to 10.98× over two AMD EPYC 7H12 CPUs with 64 cores each, outperformed by an NVIDIA Tesla V100 by a factor of 1.66×.

The exploration of FPGAs concluded with a study on mechanisms to democratise the use of FPGAs in AI inference acceleration, where a framework (CYNQ framework) and a runtime library (CYNQ) were proposed and open-sourced, laying the groundwork for further research in this area.

This dissertation closed with a qualitative study of the new trends in computer AI-driven architectures, emphasising the research of reconfigurability in Cooperative Heterogeneous Computing. This dissertation proposed the use of heterogeneous systems equipped with CPUs, GPUs, ASICs, and reconfigurable devices to mitigate high-arithmetic-intensity tasks (compute-bound) and the insertion of compute units into the memory modules to address low-arithmetic-intensity workloads (memory-bound). Furthermore, this work proposed the introduction of reconfigurable components in each component of the heterogeneous system, allowing for more aggressive optimisations and making the system more adaptable and resilient to the rapid pace of AI evolution.

Future Work

Throughout this work, several items have been left for future research, emphasising their relevance in the field.

Extension of energy models. This work was limited to studying the influence of instructions dispatched to the CPU by a process on estimating its energy consumption. The model was derived from a general principle of conservation of energy and multiple assumptions, such as fixed clock and fan speeds, constant temperature, and homogeneous power consumption, whereas reality presents more complex patterns. It is necessary

to integrate more metrics and utilise enhanced relationships to enhance the estimations under actual system conditions.

Enhanced accelerator design. The architectures and implementations presented in this thesis show the potential of the FPGA to surpass the GPU capabilities by consuming less energy. These architectures can be enhanced by utilising dynamic partial reconfiguration during runtime to replace accelerators on demand, employing more advanced approximations, such as frequency scaling through energy-aware scheduling, and arbitrary-precision floating-point operations.

Expanding the application to training. The overall contributions of this work were focused on inference, assuming it is the most repetitive task in AI and, thus, the most power-consuming. Nevertheless, the application of these work ideas can be further extended by incorporating the frameworks into training tasks, which can be made more sustainable with the introduction of FPGAs in the loop.

Democratising the FPGAs. One of the most difficult barriers to overcome in FPGA development is the closed-source nature. Further development of developer-friendly ecosystems, like CYNQ, to lower the barrier to FPGA adoption across diverse AI workloads.

Reconfigurable Cooperative Heterogeneous Computing. This thesis concluded the research by proposing the combination of Reconfigurable Computing with Cooperative Heterogeneous Computing. This will allow the systems to mitigate several workloads by offloading the work to the most suitable architecture for their execution, including the memory as an active processing element using PIM. However, hard silicon can render these systems obsolete in a short time due to the rapid evolution of AI algorithms. For this reason, this work proposes integrating reconfigurability into the processing blocks, allowing the architecture to adapt to the challenges that future AI workloads may pose, including new quantisations, memory accesses, and non-linear functions.

Ultimately, this dissertation and future work position **reconfigurable** and **energy-aware computing architectures** as a sustainable and vital foundation for the **next generation of AI systems**.

References

[1] L. G. León-Vega, E. Salazar-Villalobos, A. Rodriguez-Figueroa, and J. Castro-Godínez, "Automatic Generation of Resource and Accuracy Configurable Processing Elements," *ACM Trans. Embed. Comput. Syst.*, Apr. 2023, ISSN: 1539-9087. DOI: 10.1145/3594540.

[2] L. G. León-Vega, E. Salazar-Villalobos, and J. Castro-Godínez, "An Exploration of Accuracy Configurable Matrix Multiply-Addition Architectures using HLS," in *2022 IEEE 15th Dallas Circuit And System Conference (DCAS)*, 2022, pp. 1–6. DOI: 10.1109/DCAS53974.2022.9845501.

[3] L. G. León-Vega and J. Castro-Godínez, "Generic Accuracy Configurable Matrix Multiplication-Addition Accelerator using HLS," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2023, pp. 171–174. DOI: 10.1109/DSN-W58399.2023.00048.

[4] L. G. León-Vega, A. Chacón-Rodríguez, E. Salazar-Villalobos, and J. Castro-Godínez, "Acceleration of Fully Connected Layers on FPGA using the Strassen Matrix Multiplication," in *2023 IEEE 5th International Conference on BioInspired Processing (BIP)*, 2023, pp. 1–6. DOI: 10.1109/BIP60195.2023.10379257.

[5] Luis G. León-Vega and Diego Avila-Torres and Jorge Castro-Godínez, "CYNQ: Speeding Up FPGA Applications with Simplicity," in *2024 31st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2024.

[6] Luis G. León-Vega and Erick Obregón-Fonseca and Jorge Castro-Godínez, "A User-Friendly Ecosystem for AI FPGA-based Accelerators," in *2024 IEEE International Conference on Omni Layer Intelligent Systems (COINS)*, 2024.

[7] L. G. Leon-Vega, N. Tosato, and S. Cozzini, "A Comprehensive Analysis of Process Energy Consumption on Multi-Socket Systems with GPUs," *Latin American High Performance Computing Conference (CARLA)*, 2024. DOI: 10.1007/978-3-031-80084-9_4.

[8] L. G. Leon-Vega, N. Tosato, and S. Cozzini, "EfiMon: A Process Analyser for Granular Energy Prediction," *Latin American High Performance Computing Conference (CARLA)*, 2024. DOI: 10.1007/978-3-031-80084-9_8.

[9] L. D. Prieto-Sibaja *et al.*, "LLM Acceleration on FPGAs: A Comparative Study of Layer and Spatial Accelerators," in *2024 IEEE 42nd Central America and Panama Convention (CONCAPAN XLII)*, 2024, pp. 1–6. DOI: 10.1109/CONCAPAN63470.2024.10933896.

[10] D. Cordero-Chavarría, L. G. León-Vega, and J. Castro-Godínez, "Configurable High-Level Synthesis Approximate Arithmetic Units for Deep Learning Accelerators," in *2024 IEEE 42nd Central America and Panama Convention (CONCAPAN XLII)*, 2024, pp. 1–6. DOI: 10.1109/CONCAPAN63470.2024.10933846.

[11] A. Leiva-Valverde, F. Elizondo-Fernández, L. G. León-Vega, C. Meinhardt, and J. Castro-Godínez, "A Quantitative Evaluation of Approximate Softmax Functions for Deep Neural Networks," *Accepted in 10th Workshop on Approximate Computing (AxC 2025)*, 2025. DOI: 10.48550/arXiv.2501.13379.

[12] E. Salazar-Villalobos, L. G. Leon-Vega, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Matrix Accelerator*, version v1.1.0, 2022. DOI: 10.5281/zenodo.6413238.

[13] A. Rodriguez-Figueroa, L. G. Leon-Vega, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Convolution Accelerator*, version v0.1.0, 2022. DOI: 10.5281/zenodo.6413243.

[14] D. Cordero-Chavarria, L. G. Leon-Vega, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Math Operators*, version v0.1.1, Feb. 2023. DOI: 10.5281/zenodo.7708216.

[15] L. G. Leon-Vega, D. Cordero-Chavarria, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Computing Executer (AxC Executer)*, version v0.1.0, Mar. 2023. DOI: 10.5281/zenodo.7712042.

[16] L. G. León-Vega, N. Tosato, and S. Cozzini, *EfiMon: An Instruction-Driven Process Energy Consumption Analyser for Multi-Socket Computers*, version v0.1.0, Apr. 2024. DOI: 10.5281/zenodo.11072569.

[17] León-Vega, L., Ávila-Torres, D., Castro-Godínez, J., *CYNQ (v0.2)*, 2024. [Online]. Available: https://github.com/ECASLab/cynq.

[18] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*. Society for Industrial and Applied Mathematics, 1998. DOI: 10.1137/1.9780898719611. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719611. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898719611.

[19] M. Garland *et al.*, "Parallel Computing Experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008. DOI: 10.1109/MM.2008.57. [Online]. Available: https://doi.org/10.1109/MM.2008.57.

[20] T. Hoefler, M. Copik, P. Beckman, A. Jones, I. Foster, M. Parashar, D. Reed, M. Troyer, T. Schulthess, D. Ernst, and J. Dongarra, " XaaS: Acceleration as a Service to Enable Productive High-Performance Cloud Computing," *Computing in Science & Engineering*, vol. 26, no. 03, pp. 40–51, Jul. 2024, ISSN: 1558-366X. DOI: 10.1109/MCSE.2024.3382154. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MCSE.2024.3382154.

[21] H. Kim, G. Ye, N. Wang, A. Yazdanbakhsh, and N. S. Kim, "Exploiting Intel Advanced Matrix Extensions (AMX) for Large Language Model Inference," *IEEE Computer Architecture Letters*, vol. 23, no. 1, pp. 117–120, 2024. DOI: 10.1109/LCA.2024.3397747.

[22] Meta Platforms, Inc. "Meta's 2GW data center: How the company plans to grow AI." Accessed: 2025-04-28. (2025), [Online]. Available: https://technologymagazine.com/articles/metas-2gw-data-centre-how-the-company-plans-to-grow-ai.

[23] N. Koutsoukos. "OpenAI pitches Biden admin on need to build massive AI data centers." Accessed: 2025-04-28. (2024), [Online]. Available: https://nypost.com/2024/09/25/business/openai-pitches-biden-admin-on-need-to-build-massive-ai-data-centers/.

[24] P. Campbell. "AI supercomputers could soon consume as much power as entire cities." Accessed: 2025-04-28. (2025), [Online]. Available: https://www.businessinsider.com/ai-supercomputers-power-2030-city-nuclear-2025-4.

[25] TOP500.org, *TOP 500: November 2023*, 2023.

[26] J. Li, J. Xu, S. Huang, Y. Chen, W. Li, J. Liu, Y. Lian, J. Pan, L. Ding, H. Zhou, Y. Wang, and G. Dai, *Large Language Model Inference Acceleration: A Comprehensive Hardware Perspective*, 2025. arXiv: 2410.04466 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2410.04466.

[27] C. Zhao, L. Zhao, J. Li, and Z. Xu, *DeepGEMM: clean and efficient FP8 GEMM kernels with fine-grained scaling*, https://github.com/deepseek-ai/DeepGEMM, 2025.

[28] International Energy Agency, *World Energy Outlook 2022*, https://www.iea.org/countries/italy/energy-mix, (Accessed on 05/04/2024), 2022.

[29] J. Treibig *et al.*, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of PSTI2010*, San Diego CA, 2010.

[30] AMD, *AMD uProf*, 2024. [Online]. Available: %7Bhttps://www.amd.com/en/developer/uprof.html%7D.

[31] NVIDIA, *Jetson Modules*, 2024. [Online]. Available: https://developer.nvidia.com/embedded/jetson-modules.

[32] A. Yoo *et al.*, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60, ISBN: 978-3-540-39727-4.

[33] H. Moazamigoodarzi *et al.*, "Modeling temperature distribution and power consumption in IT server enclosures with row-based cooling architectures," *Applied Energy*, vol. 261, p. 114 355, 2020, ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2019.114355.

[34] A. Lewis, S. Ghosh, and N.-F. Tzeng, "Run-time energy consumption estimation based on workload in server systems," in *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, San Diego, California: USENIX Association, 2008, p. 4.

[35] W. Dargie, "A Stochastic Model for Estimating the Power Consumption of a Processor," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1311–1322, 2015. DOI: 10.1109/TC.2014.2315629.

[36] O. Sarood *et al.*, "Maximizing throughput of overprovisioned hpc data centers under a strict power budget," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 807–818. DOI: 10.1109/SC.2014.71.

[37] R. Bertran *et al.*, "A Systematic Methodology to Generate Decomposable and Responsive Power Models for CMPs," *IEEE Transactions on Computers*, vol. 62, no. 7, pp. 1289–1302, 2013. DOI: 10.1109/TC.2012.97.

[38] A. Shahid *et al.*, "Energy Predictive Models of Computing: Theory, Practical Implications and Experimental Analysis on Multicore Processors," *IEEE Access*, vol. 9, pp. 63 149–63 172, 2021. DOI: 10.1109/ACCESS.2021.3075139.

[39] L. Bogdanov and G. Zhelezov, "Energy Consumption of Assembly Instructions in Load-Store Microprocessor Architectures," in *2021 12th National Conference with International Participation (ELECTRONICA)*, 2021, pp. 1–4. DOI: 10.1109/ELECTRONICA52725.2021.9513670.

[40] Y. S. Shao and D. Brooks, "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013, pp. 389–394. DOI: 10.1109/ISLPED.2013.6629328.

[41] L. Zhang, X. Wu, and Y. Zhao, "Instruction-Level Instantaneous Power Modeling for VLIW Processor," in *2015 IEEE (UIC-ATC-ScalCom)*, 2015, pp. 1451–1456. DOI: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.261.

[42] Y. Wang and N. Ranganathan, "An Instruction-Level Energy Estimation and Optimization Methodology for GPU," in *2011 IEEE 11th International Conference on Computer and Information Technology*, 2011, pp. 621–628. DOI: 10.1109/CIT.2011.69.

[43] M. Krunic *et al.*, "Instructions energy consumption on a heterogeneous multicore platform," in *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems*, ser. ECBS '17, Larnaca, Cyprus: Association for Computing Machinery, 2017, ISBN: 9781450348430. DOI: 10.1145/3123779.3123795.

[44] J. Corbalan, L. Alonso, J. Aneas, and L. Brochard, "Energy optimization and analysis with ear," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 464–472. DOI: 10.1109/CLUSTER49012.2020.00067.

[45] H. Jagode, A. Danalis, G. Congiu, D. Barry, A. Castaldo, and J. Dongarra, "Advancements of papi for the exascale generation," *The International Journal of High Performance Computing Applications*, vol. 39, no. 2, pp. 251–268, 2025. DOI: 10.1177/10943420241303884. eprint: https://doi.org/10.1177/10943420241303884. [Online]. Available: https://doi.org/10.1177/10943420241303884.

[46] T. Allen and R. Ge, "Characterizing power and performance of gpu memory access," in *2016 4th International Workshop on Energy Efficient Supercomputing (E2SC)*, 2016, pp. 46–53. DOI: 10.1109/E2SC.2016.012.

[47] Y. Wang and H. Kim, "Work-in-progress: Understanding the effect of kernel scheduling on gpu energy consumption," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 584–587. DOI: 10.1109/RTSS46320.2019.00070.

[48] J. Coplin and M. Burtscher, "Energy, Power, and Performance Characterization of GPGPU Benchmark Programs," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1190–1199. DOI: 10.1109/IPDPSW.2016.164.

[49] G. Alavani *et al.*, "GPPT: A Power Prediction Tool for CUDA Applications," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 2021. DOI: 10.1109/ASEW52652.2021.00054.

[50] G. Alavani *et al.*, "Program Analysis and Machine Learning–based Approach to Predict Power Consumption of CUDA Kernel," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 8, no. 4, Jul. 2023, ISSN: 2376-3639. DOI: 10.1145/3603533.

[51] M. Dayarathna, Y. Wen, and R. Fan, "Data Center Energy Consumption Modeling: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 732–794, 2016. DOI: 10.1109/COMST.2015.2481183.

[52] C. Jin *et al.*, "A review of power consumption models of servers in data centers," *Applied Energy*, vol. 265, p. 114 806, 2020, ISSN: 0306-2619. DOI: 10.1016/j.apenergy.2020.114806.

[53] T. L. Vasques, P. Moura, and A. de Almeida, "A review on energy efficiency and demand response with focus on small and medium data centers," *Energy Efficiency*, vol. 12, no. 5, pp. 1399–1428, Jun. 2019, ISSN: 1570-6478. DOI: 10.1007/s12053-018-9753-2.

[54] I. Alan, E. Arslan, and T. Kosar, "Energy-Aware Data Transfer Tuning," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 626–634. DOI: 10.1109/CCGrid.2014.117.

[55] F. Chen *et al.*, "Experimental analysis of task-based energy consumption in cloud computing systems," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '13, Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 295–306, ISBN: 9781450316361. DOI: 10.1145/2479871.2479911.

[56] X. Qi and D. Zhu, "Power Management for Real-Time Embedded Systems on Block-Partitioned Multicore Platforms," in *2008 International Conference on Embedded Software and Systems*, 2008, pp. 110–117. DOI: 10.1109/ICESS.2008.43.

[57] NVIDIA, *CUDA C++ Programming Guide*, May 2024. [Online]. Available: %7Bhttps://docs.nvidia.com/cuda/cuda-c-programming-guide/#%7D.

[58] NVIDIA, *NVML API Reference*, Mar. 2024. [Online]. Available: %7Bhttps://docs.nvidia.com/deploy/nvml-api/nvml-api-reference.html%7D.

[59] Q. Wang *et al.*, "AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12. DOI: 10.1145/2503210.2503219.

[60] NVIDIA, *NVIDIA TESLA V100 GPU ARCHITECTURE*, Aug. 2027. [Online]. Available: %7Bhttps://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf%7D.

[61] D. Mahajan *et al.*, "TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 14–26. DOI: 10.1109/HPCA.2016.7446050.

[62] G. K. Thiruvathukal, Y.-H. Lu, J. Kim, Y. Chen, and B. Chen, *Low-power computer vision: Improve the efficiency of artificial intelligence*. CRC Press, 2022, p. 36. DOI: 10.1201/9781003162810. [Online]. Available: https://doi.org/10.1201/9781003162810.

[63] Xilinx, *Zynq-7000 SoC*, u.d. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[64] Xilinx, *Kria K26 SOM: The Ideal Platform for Vision AI at the Edge*, 2021. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/white_papers/wp529-som-benchmarks.pdf.

[65] S. Zeng, J. Liu, G. Dai, X. Yang, T. Fu, H. Wang, W. Ma, H. Sun, S. Li, Z. Huang, Y. Dai, J. Li, Z. Wang, R. Zhang, K. Wen, X. Ning, and Y. Wang, "FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '24, Monterey, CA, USA: Association for Computing Machinery, 2024, pp. 223–234, ISBN: 9798400704185. DOI: 10.1145/3626202.3637562. [Online]. Available: https://doi.org/10.1145/3626202.3637562.

[66] E. Mohsen, "Performance and Bandwidth in a Cost-Optimized," vol. 423, pp. 1–14, 2018.

[67] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, *Pruning and Quantization for Deep Neural Network Acceleration: A Survey*, 2021. arXiv: 2101.09671 [cs.CV].

[68] Xilinx Inc., *DPUCAHX8L for Convolutional Neural Networks*, 2021.

[69] D. Rongshi and T. Yongming, "Accelerator Implementation of Lenet-5 Convolution Neural Network Based on FPGA with HLS," *2019 3rd International Conference on Circuits, System and Simulation, ICCSS 2019*, pp. 64–67, 2019. DOI: 10.1109/CIRSYSSIM.2019.8935599.

[70] M. A. Arshad, S. Shahriar, and A. Sagahyroon, "On the Use of FPGAs to Implement CNNs: A Brief Review," *Proceedings - 2020 International Conference on Computing, Electronics and Communications Engineering, iCCECE 2020*, pp. 230–236, 2020. DOI: 10.1109/iCCECE49321.2020.9231243.

[71] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 73–82, ISBN: 9781450361378. DOI: 10.1145/3289602.3293915. [Online]. Available: https://doi.org/10.1145/3289602.3293915.

[72] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2019. DOI: 10.1109/TCAD.2017.2785257.

[73] G. Zervakis, H. Saadat, H. Mrouch, A. Gerstlauer, S. Parameswaran, and J. Henkel, "Approximate Computing for ML: State-of-the-art, Challenges and Visions," *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pp. 189–196, 2021. DOI: 10.1145/3394885.3431632.

[74] A. Lavin and S. Gray, *Fast algorithms for convolutional neural networks*, 2015. arXiv: 1509.09308 [cs.NE].

[75] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional Neural Networks using Logarithmic Data Representation," 2016. arXiv: 1603.01025. [Online]. Available: http://arxiv.org/abs/1603.01025.

[76] J. Li and A. Louri, "AdaPrune : An Accelerator-aware Pruning Technique for Sustainable CNN Accelerators," vol. XX, no. XX, 2021. DOI: 10.1109/TSUSC.2021.3060690.

[77] X. Chang, H. Pan, W. Lin, and H. Gao, "A Mixed-Pruning Based Framework for Embedded Convolutional Neural Network Acceleration," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–10, 2021, ISSN: 15580806. DOI: 10.1109/TCSI.2020.3048260.

[78] K. Bhardwaj, N. Suda, and R. Marculescu, "EdgeAI: A Vision for Deep Learning in IoT Era," *IEEE Design and Test*, vol. 2356, no. c, pp. 1–6, 2019, ISSN: 21682364. DOI: 10.1109/MDAT.2019.2952350.

[79] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-M. Hwu, and D. Chen, "Dnnexplorer: A framework for modeling and exploring a novel paradigm of fpga-based dnn accelerator," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[80] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. D. Guglielmo, P. Harris, J. Krupa, D. Rankin, M. Blanco, V. J. Hester, Y. Luo, J. Mamish, S. Orgrenci-Memik, T. Aarestaad, H. Javed, V. Loncar, M. Pierini, A. A. Pol, S. Summers, J. Duarte, S. Hauck, S.-C. Hsu, J. Ngadiuba, M. Liu, D. Hoang, E. Kreinar, H. Herndon, Z. Wu, M. Blanco Valentin, and J. Hester, "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices," in *TinyML Research Symposium*, 2021. arXiv: 2103.05579v1.

[81] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

[82] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[83] B. Janßen, P. Zimprich, and M. Hübner, "A dynamic partial reconfigurable overlay concept for PYNQ," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056786.

[84] E. Lee, Y. Lee, S.-S. Lee, and B.-H. Choi, "Implementation of a Round Robin Processing Element for Deep Learning Accelerator," in *2020 International SoC Design Conference (ISOCC)*, 2020, pp. 302–303. DOI: 10.1109/ISOCC50952.2020.9333012.

[85] E. Lee, M. Sung, S.-J. Jang, J. Park, and S.-S. Lee, "Memory-centric architecture of neural processing unit for edge device," in *2021 18th International SoC Design Conference (ISOCC)*, 2021, pp. 240–241. DOI: 10.1109/ISOCC53507.2021.9613977.

[86] T. T. Thanh Bui and B. Phillips, "A Scalable Network-on-Chip Based Neural Network Implementation on FPGAs," in *2019 IEEE-RIVF International Conference on Computing and Communication Technologies (RIVF)*, 2019, pp. 1–6. DOI: 10.1109/RIVF.2019.8713613.

[87] Z. Lin, M. Zhang, D. Weng, and F. Liu, "An Efficient Accelerator with Winograd for Novel Convolutional Neural Networks," in *2022 5th International Conference on Circuits, Systems and Simulation (ICCSS)*, 2022, pp. 126–130. DOI: 10.1109/ICCSS55260.2022.9802420.

[88] Luis Gerardo León-Vega, "Design of a Library of Generic Accelerators of DNN-based Inference Algorithms for Low-end FPGAs," Available at RepositorioTEC, Master's thesis, Instituto Tecnológico de Costa Rica, Dec. 2022. [Online]. Available: https://repositoriotec.tec.ac.cr/handle/2238/14360.

[89] F. Fahim *et al.*, "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices," in *TinyML Research Symposium*, 2021. arXiv: 2103.05579v1.

[90] V. Strassen *et al.*, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[91] W. Miller, "Computational Complexity and Numerical Stability," *SIAM Journal on Computing*, vol. 4, no. 2, pp. 97–107, 1975. DOI: 10.1137/0204009.

[92] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, 2015. DOI: 10.1007/s11263-015-0816-y.

[93] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006. DOI: 10.1016/j.patrec.2005.10.010.

[94] A. Ahmad, L. Du, and W. Zhang, "Fast and Practical Strassen's Matrix Multiplication using FPGAs," in *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, 2024, pp. 311–317. DOI: 10.1109/FPL64840.2024.00050.

[95] Borras, Hendrik and others, "Open-source FPGA-ML codesign for the MLPerf Tiny Benchmark," Tech. Rep., 2022. arXiv: 2206.11791. [Online]. Available: https://cds.cern.ch/record/2826586.

[96] C. Banbury *et al.*, "MLPerf Tiny Benchmark," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, J. Vanschoren and S. Yeung, Eds., vol. 1, Curran, 2021. DOI: 10.48550/arXiv.2106.07597. [Online]. Available: https://doi.org/10.48550/arXiv.2106.07597.

[97] Y. Koizumi, S. Saito, H. Uematsu, N. Harada, and K. Imoto, "ToyADMOS: A Dataset of Miniature-Machine Operating Sounds for Anomalous Sound Detection," in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, 2019, pp. 313–317. DOI: 10.1109/WASPAA.2019.8937164.

[98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[99] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size*, 2016. arXiv: 1602.07360 [cs.CV].

[100] M. Sandler *et al.*, "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018. arXiv: 1801.04381. [Online]. Available: http://arxiv.org/abs/1801.04381.

[101] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *2013 18th IEEE European Test Symposium (ETS)*, 2013, pp. 1–6. DOI: 10.1109/ETS.2013.6569370.

[102] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-Power Digital Signal Processing Using Approximate Adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, 2013. DOI: 10.1109/TCAD.2012.2217962.

[103] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate Computing: A Survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2016. DOI: 10.1109/MDAT.2015.2505723.

[104] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 4, pp. 850–862, 2010. DOI: 10.1109/TCSI.2009.2027626.

[105] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC Design Automation Conference 2012*, 2012, pp. 820–825. DOI: 10.1145/2228360.2228509.

[106] D. Hernandez-Araya, J. Castro-Godínez, M. Shafique, and J. Henkel, "AUGER: A Tool for Generating Approximate Arithmetic Circuits," in *2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS)*, 2020, pp. 1–4. DOI: 10.1109/LASCAS45839.2020.9069045.

[107] J. Castro-Godínez, J. Mateus-Vargas, M. Shafique, and J. Henkel, "AxHLS: Design Space Exploration and High-Level Synthesis of Approximate Accelerators Using Approximate Functional Units and Analytical Models," in *Proceedings of the 39th International Conference on Computer-Aided Design*, Virtual Event, USA, 2020, ISBN: 9781450380263. DOI: 10.1145/3400302.3415732.

[108] H. Zhang, H. Xiao, H. Qu, and S.-B. Ko, "FPGA-Based Approximate Multiplier for Efficient Neural Computation," in *2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, 2021, pp. 1–4. DOI: 10.1109/ICCE-Asia53811.2021.9641971.

[109] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 258–261. DOI: 10.23919/DATE.2017.7926993.

[110] S. Ullah, S. S. Murthy, and A. Kumar, "SMApproxlib: Library of FPGA-Based Approximate Multipliers," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18, San Francisco, California: Association for Computing Machinery, 2018, ISBN: 9781450357005. DOI: 10.1145/3195970.3196115. [Online]. Available: https://doi-org.ezproxy.itcr.ac.cr/10.1145/3195970.3196115.

[111] S. Ullah, S. Rehman, B. S. Prabakaran, F. Kriebel, M. A. Hanif, M. Shafique, and A. Kumar, "Area-Optimized Low-Latency Approximate Multipliers for FPGA-based Hardware Accelerators," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465781.

[112] S. Ullah, S. Rehman, M. Shafique, and A. Kumar, "High-Performance Accurate and Approximate Multipliers for FPGA-Based Hardware Accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 211–224, 2022. DOI: 10.1109/TCAD.2021.3056337.

[113] B. S. Prabakaran, S. Rehman, M. A. Hanif, S. Ullah, G. Mazaheri, A. Kumar, and M. Shafique, "DeMAS: An efficient design methodology for building approximate adders for FPGA-based systems," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 917–920. DOI: 10.23919/DATE.2018.8342140.

[114] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A low latency generic accuracy configurable adder," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6. DOI: 10.1145/2744769.2744778.

[115] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-Power Digital Signal Processing Using Approximate Adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, 2013. DOI: 10.1109/TCAD.2012.2217962.

[116] H. A. Almurib, T. N. Kumar, and F. Lombardi, "Inexact designs for approximate low power addition by cell replacement," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 660–665.

[117] D. Celia, V. Vasudevan, and N. Chandrachoodan, "Optimizing power-accuracy trade-off in approximate adders," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 1488–1491. DOI: 10.23919/DATE.2018.8342248.

[118] D. Celia, V. Vasudevan, and N. Chandrachoodan, "Probabilistic Error Modeling for Two-part Segmented Approximate Adders," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351273.

[119] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 2010, pp. 807–814.

[120] Y. Cao *et al.*, "Cordic-based Softmax Acceleration Method of Convolution Neural Network on FPGA," in *2020 IEEE International Conference on Artificial Intelligence and Information Systems (ICAIIS)*, 2020, pp. 66–70. DOI: 10.1109/ICAIIS49377.2020.9194894.

[121] M. Wasef and N. Rafla, "Hardware implementation of multi-rate input softmax activation function," in *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2021, pp. 783–786. DOI: 10.1109/MWSCAS47672.2021.9531761.

[122] K. Chen, Y. Gao, H. Waris, W. Liu, and F. Lombardi, "Approximate softmax functions for energy-efficient deep neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 1, pp. 4–16, 2023. DOI: 10.1109/TVLSI.2022.3224011.

[123] Y. Gao, W. Liu, and F. Lombardi, "Design and implementation of an approximate softmax layer for deep neural networks," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180870.

[124] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.)

[125] M. Abramowitz, *Handbook of Mathematical Functions, With Formulas, Graphs, and Mathematical Tables,* USA: Dover Publications, Inc., 1974, ISBN: 0486612724.

[126] R. H. Bartels, J. C. Beatty, and B. A. Barsky, *An Introduction to Splines for Use in Computer Graphics & Geometric Modeling*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, ISBN: 0934613273.

[127] A. S. Roy and A. S. Dhar, "A Novel Approach for Fast and Accurate Mean Error Distance Computation in Approximate Adders," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351171.

[128] A. Markov, "Extension des théorèmes limites du calcul des probabilités aux sommes de quantités liées en chaînes," *Mem. Acad. Sci. St. Petersburg*, vol. 8, pp. 365–397, 1908.

[129] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[130] León-Vega, L., Leiva-Valverde, A., Prieto-Sibaja, L., Blanco, G., Castro-Godínez, J., *HLS FPGA Accelerators*, 2024. [Online]. Available: https://github.com/ECASLab/hls-fpga-accelerators/.

[131] S. Ma *et al.*, *The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits*, 2024. arXiv: 2402.17764 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2402.17764.

[132] E. Chaves-González and L. G. León-Vega, "Benchmarking the NXP i. MX8M+ neural processing unit: smart parking case study," *Revista Tecnología en Marcha*, ág–26, 2022.

[133] M. Blott *et al.*, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[134] P. Qi *et al.*, "Accommodating Transformer onto FPGA: Coupling the Balanced Model Compression and FPGA-Implementation Optimization," in *Proceedings of the 2021 Great Lakes Symposium on VLSI*, Virtual Event, USA, 2021, pp. 163–168, ISBN: 9781450383936. DOI: 10.1145/3453688.3461739. [Online]. Available: https://doi.org/10.1145/3453688.3461739.

[135] B. Li *et al.*, "FTRANS: energy-efficient acceleration of transformers using FPGA," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, Boston, Massachusetts, 2020, pp. 175–180, ISBN: 9781450370530. DOI: 10.1145/3370748.3406567. [Online]. Available: https://doi.org/10.1145/3370748.3406567.

[136] H. Chen *et al.*, "Understanding the Potential of FPGA-Based Spatial Acceleration for Large Language Model Inference," *ACM Trans. Reconfigurable Technol. Syst.*, Apr. 2024, ISSN: 1936-7406. DOI: 10.1145/3656177. [Online]. Available: https://doi.org/10.1145/3656177.

[137] N. Chen *et al.*, *Is Bigger and Deeper Always Better? Probing LLaMA Across Scales and Layers*, 2024. arXiv: 2312.04333 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2312.04333.

[138] A. Vaswani *et al.*, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Long Beach, California, USA: Curran Associates Inc., 2017, ISBN: 9781510860964.

[139] G. Georgi, *GGML - Tensor library for machine learning*, Jun. 2024. [Online]. Available: https://github.com/ggerganov/ggml.

[140] L. Prieto-Sibaja and G. Gerganov, *GGML - Tensor library for machine learning*, Jun. 2024. [Online]. Available: https://github.com/ecaslab/ggml.

[141] AMD, *Xilinx Runtime Library (XRT)*, 2024. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/xrt.html.

[142] AMD, *Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)*, 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1400-vitis-embedded/Vitis-Unified-Software-Platform-Overview.

[143] B. Janßen *et al.*, "A dynamic partial reconfigurable overlay concept for PYNQ," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056786. [Online]. Available: https://doi.org/10.23919/FPL.2017.8056786.

[144] AMD, *Vitis Libraries*, version 2023.2, 2023. [Online]. Available: %7Bhttps://github.com/Xilinx/Vitis%5C_Libraries%7D.

[145] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, "FOS: A Modular FPGA Operating System for Dynamic Workloads," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 4, Sep. 2020, ISSN: 1936-7406. DOI: 10.1145/3405794. [Online]. Available: https://doi.org/10.1145/3405794.

[146] K. D. Pham, A. Vaishnav, M. Vesper, and D. Koch, "ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications," in *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*, 2018, pp. 1–9, ISBN: 978-3-8007-4723-8.

[147] K. Alfaro-Badilla *et al.*, "Prototyping a Biologically Plausible Neuron Model on a Heterogeneous CPU-FPGA Board," in *2019 IEEE 10th Latin American Symposium on Circuits & Systems (LASCAS)*, 2019, pp. 5–8. DOI: 10.1109/LASCAS.2019.8667538. [Online]. Available: https://doi.org/10.1109/LASCAS.2019.8667538.

[148] K. Alfaro-Badilla *et al.*, "Improving the simulation of biologically accurate neural networks using data flow hls transformations on heterogeneous soc-fpga platforms," in *High Performance Computing*, Cham: Springer International Publishing, 2020, pp. 185–199, ISBN: 978-3-030-41005-6. DOI: 10.1007/978-3-030-41005-6_13. [Online]. Available: https://doi.org/10.1007/978-3-030-41005-6%5C_13.

[149] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999, ISBN: 0-89871-447-8 (paperback).

[150] M. Frigo, "A fast fourier transform compiler," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99, Atlanta, Georgia, USA: Association for Computing Machinery, 1999, pp. 169–180, ISBN: 1581130945. DOI: 10.1145/301618.301661. [Online]. Available: https://doi.org/10.1145/301618.301661.

[151] NVIDIA, *CUDA Libraries*, n.d. [Online]. Available: https://docs.nvidia.com/cuda-libraries/.

[152] Xilinx, *Xfopencv - xilinx opencv library*, 2019. [Online]. Available: %7Bhttps://github.com/Xilinx/xfopencv%7D.

[153] C. S. Pabla, "Completely fair scheduler," *Linux J.*, vol. 2009, no. 184, Aug. 2009, ISSN: 1075-3583.

[154] E. Obregón-Fonseca, L. León-Vega, and J. Castro-Godínez, *AxC Kernel Delegate Interface*, 2022. [Online]. Available: https://gitlab.com/ecas-lab-tec/approximate-flexible-acceleration-ml/axc-kernel-delegate-interface.

[155] C. Silvano, D. Ielmini, F. Ferrandi, L. Fiorin, S. Curzel, L. Benini, F. Conti, A. Garofalo, C. Zambelli, E. Calore, S. Schifano, M. Palesi, G. Ascia, D. Patti, N. Petra, D. De Caro, L. Lavagno, T. Urso, V. Cardellini, G. Cardarilli, R. Birke, and S. Perri, "A survey on deep learning hardware accelerators for heterogeneous hpc platforms," *ACM Comput. Surv.*, Apr. 2025, Just Accepted, ISSN: 0360-0300. DOI: 10.1145/3729215. [Online]. Available: https://doi.org/10.1145/3729215.

[156] Q. Wu, Y. Shen, and M. Zhang, "Heterogeneous computing and applications in deep learning: A survey," in *Proceedings of the 5th International Conference on Computer Science and Software Engineering*, ser. CSSE '22, Guilin, China: Association for Computing Machinery, 2022, pp. 383–387, ISBN: 9781450397780. DOI: 10.1145/3569966.3570075. [Online]. Available: https://doi.org/10.1145/3569966.3570075.

[157] Y. Song, Z. Mi, H. Xie, and H. Chen, "Powerinfer: Fast large language model serving with a consumer-grade gpu," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP '24, Austin, TX, USA: Association for Computing Machinery, 2024, pp. 590–606, ISBN: 9798400712517. DOI: 10.1145/3694715.3695964. [Online]. Available: https://doi.org/10.1145/3694715.3695964.

[158] G. Heo, S. Lee, J. Cho, H. Choi, S. Lee, H. Ham, G. Kim, D. Mahajan, and J. Park, "NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 722–737, ISBN: 9798400703867. DOI: 10.1145/3620666.3651380. [Online]. Available: https://doi.org/10.1145/3620666.3651380.

[159] M. Seo, X. T. Nguyen, S. J. Hwang, Y. Kwon, G. Kim, C. Park, I. Kim, J. Park, J. Kim, W. Shin, J. Won, H. Choi, K. Kim, D. Kwon, C. Jeong, S. Lee, Y. Choi, W. Byun, S. Baek, H.-J. Lee, and J. Kim, "Ianus: Integrated accelerator based on npu-pim unified memory system," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 545–560, ISBN: 9798400703867. DOI: 10.1145/3620666.3651324. [Online]. Available: https://doi.org/10.1145/3620666.3651324.

[160] T. Kim, K. Choi, Y. Cho, J. Cho, H.-J. Lee, and J. Sim, "Monde: Mixture of near-data experts for large-scale sparse models," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24, San Francisco, CA, USA: Association for Computing Machinery, 2024, ISBN: 9798400706011. DOI: 10.1145/

3649329.3655951. [Online]. Available: https://doi.org/10.1145/3649329.3655951.

[161] J. Sharda, P.-K. Hsu, and S. Yu, "Accelerator design using 3d stacked capacitorless dram for large language models," in *2024 IEEE 6th International Conference on AI Circuits and Systems (AICAS)*, 2024, pp. 487–491. DOI: 10.1109/AICAS59952.2024.10595901.

[162] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017. DOI: 10.1109/MM.2017.37.

[163] D. D. Sharma, "Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing," in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2022, pp. 5–12. DOI: 10.1109/HOTI55740.2022.00017.

[164] D. Rossi, F. Conti, M. Eggiman, A. D. Mauro, G. Tagliavini, S. Mach, M. Guermandi, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini, "Vega: A ten-core soc for iot endnodes with dnn acceleration and cognitive wake-up from mram-based state-retentive sleep mode," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 1, pp. 127–139, 2022. DOI: 10.1109/JSSC.2021.3114881.

[165] A. D. Mauro, M. Scherer, D. Rossi, and L. Benini, *Kraken: A direct event/frame-based multi-sensor fusion soc for ultra-efficient visual processing in nano-uavs*, 2022. arXiv: 2209.01065 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2209.01065.

[166] Y. Tortorella, L. Bertaccini, L. Benini, D. Rossi, and F. Conti, "RedMule: A mixed-precision matrix–matrix operation engine for flexible and energy-efficient on-chip linear algebra and TinyML training acceleration," *Future Generation Computer Systems*, vol. 149, pp. 122–135, 2023, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.07.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X23002546.

[167] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2007.

[168] R. Prabhakar, Y. Zhang, and K. Olukotun, "Coarse-grained reconfigurable architectures," in *NANO-CHIPS 2030: On-Chip AI for an Efficient Data-Driven World*, B. Murmann and B. Hoefflinger, Eds. Cham: Springer International Publishing, 2020, pp. 227–246, ISBN: 978-3-030-18338-7. DOI: 10.1007/978-3-030-18338-7_14. [Online]. Available: https://doi.org/10.1007/978-3-030-18338-7_14.

[169] A. Podobas, K. Sano, and S. Matsuoka, "A survey on coarse-grained reconfigurable architectures from a performance perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020. DOI: 10.1109/ACCESS.2020.3012084.

[170] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, "RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 546–564. DOI: 10.1109/MICRO56248.2022.00046.

[171] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "REVAMP: a systematic framework for heterogeneous CGRA realization," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22, Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 918–932, ISBN: 9781450392051. DOI: 10.1145/3503222.3507772. [Online]. Available: https://doi.org/10.1145/3503222.3507772.

[172] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "Snafu: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1027–1040. DOI: 10.1109/ISCA52012.2021.00084.

[173] O. Akbari, M. Kamal, A. Afzali-Kusha, M. Pedram, and M. Shafique, "X-CGRA: An Energy-Efficient Approximate Coarse-Grained Reconfigurable Architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2558–2571, 2020. DOI: 10.1109/TCAD.2019.2937738.

[174] Y. Qiu, Y. Mao, X. Gao, S. Chen, J. Li, W. Yin, and L. Wang, "FDRA: A Framework for a Dynamically Reconfigurable Accelerator Supporting Multi-Level Parallelism," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, no. 1, Jan. 2024, ISSN: 1936-7406. DOI: 10.1145/3614224. [Online]. Available: https://doi.org/10.1145/3614224.

[175] N. Dao, A. Attwood, B. Healy, and D. Koch, "FlexBex: A RISC-V with a Reconfigurable Instruction Extension," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 190–195. DOI: 10.1109/ICFPT51103.2020.00034.

[176] D. van den Berg, "A Dynamically Reconfigurable RISC-V Processor Based on the MOLEN Paradigm," Master's thesis, Delft University of Technology, 2024. [Online]. Available: https://repository.tudelft.nl/record/uuid:d5da6107-696e-462f-b18f-654ce9df4369.

[177] A. Delavari, F. Ghoreishy, H. S. Shahhoseini, and S. Mirzakuchaki, "A Reconfigurable Approximate Computing RISC-V Platform for Fault-Tolerant Applications," in *2024 27th Euromicro Conference on Digital System Design (DSD)*, 2024, pp. 81–89. DOI: 10.1109/DSD64264.2024.00020.

[178] U. Köster, T. Webb, X. Wang, M. Nassar, N. Bansal, C. Hsieh, M. Shoeybi, H. Xiao, O. Kuchaiev, M. Garland, *et al.*, "Block Floating Point Arithmetic for Training Deep Neural Networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.