Efficient Hardware Acceleration of Deep Neural Networks via Arithmetic Complexity Reduction

A Thesis submitted in fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in

Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona, 2023

by

Enrico Reggiani

Advisors:
Dr. Adrián Cristal Kestelman
Prof. Mauro Olivieri

Tutor:
Prof. Miquel Moretó Planas

Collaborator:
Dr. Osman Sabri Ünsal
Acknowledgements

I would like to express my deepest gratitude and appreciation to all the individuals who have contributed to the completion of this Ph.D. thesis. Their guidance, support, and encouragement have been invaluable throughout this challenging journey.

First and foremost, I am immensely grateful to my Ph.D. advisors, Dr. Adrián Cristal, Dr. Osman Sabri Unsal, and Prof. Mauro Olivieri, and to my thesis tutor, Prof. Miquel Moretó Planas, for their exceptional mentorship and unwavering commitment to my academic and personal growth. Their expertise, patience, and dedication have been instrumental in shaping the direction of this research and helping me overcome various obstacles along the way. I am truly lucky to have had the opportunity to work under their guidance.

I extend my heartfelt appreciation to the members of my thesis committee Prof. Ramon Canal Corretger, Prof. Adrià Armejach Sanosa, Prof. Daniel Jiménez González, Prof. Gulay Yalcin, and Prof. Pascal Felber for their insightful feedback, constructive criticism, and invaluable suggestions. Their expertise and thorough evaluation have significantly strengthened this work and broadened my understanding of the subject matter.

I also want to thank the main funding partners who enabled these Ph.D. studies, mainly founded by the ERDF Operational Program of Catalonia 2014-2020, with a grant from the Spanish State Research Agency [PID2019-107255GB].

Special thanks are also due to the supervisors, mentors, and colleagues I had the honor to work with during my internships and research activities. Especially, I want to thank Dr. Paolo Costa, Dr. Thomas Karagiannis, and Dr. Krzysztof Jozwik from Microsoft Research to have believed in my professional abilities even before I did, to Dr. Renzo Andri and Dr. Lukas Cavigelli from Huawei Research for their exceptional mentorship and for sharing their wisdom and expertise during our research, and to Dr. Filippo Mantovani and his team, who made me feel at home from day one at BSC.
My sincere thanks go to my colleagues and fellow researchers at BSC, who have been a source of inspiration and motivation. Their intellectual discussions, collaborative efforts, and willingness to share ideas have greatly enriched my research experience. I am thankful for their friendship and the camaraderie we have shared.

I am deeply indebted to my family for their unwavering love, encouragement, and belief in my abilities. Their constant support, understanding, and sacrifices have been the driving force behind my pursuit of higher education. Thank you for always believing in me and for instilling in me the values of perseverance and determination. I am forever grateful for your sacrifices and the countless ways you have shaped my life.

To Anna, words cannot express the depth of my appreciation for your constant support and patience during challenging times. Your presence by my side has provided the comfort and reassurance I needed to navigate the demands of this Ph.D. journey. Thank you for being my rock and for sharing this remarkable adventure with me.

Lastly, I would like to acknowledge the countless individuals who have played a part, no matter how big or small, in shaping my academic and personal journey. Their support, encouragement, and belief in my abilities have been instrumental in my growth and development as a researcher and as an individual.

In conclusion, I would like to express my profound appreciation to everyone who has contributed to the completion of this Ph.D. thesis. I am truly grateful for the opportunity to have worked with such remarkable individuals.
Abstract

Over the past decade, artificial intelligence (AI) has witnessed significant advancements, driven by the remarkable progress in deep learning, a subfield of machine learning that employs deep neural networks (DNNs) to achieve unprecedented levels of accuracy in various tasks such as image recognition, speech recognition, and natural language processing. However, the increasing complexity of DNN models presents challenges for efficient computation on modern computing systems. On the one side, the number of layers in modern DNNs has grown significantly, resulting in an exponential increase in the number of operations and parameters required for their computation, and posing a challenge to edge and mobile computing systems, featuring tight power and memory constraints. On the other side, the adoption of sophisticated computational layers, such as recent complex activation functions, can significantly impact the runtime of DNNs on current high-performance computing (HPC) and cloud computing accelerators.

This thesis proposes several techniques to reduce the arithmetic complexity of DNN computations, as well as hardware architectures to improve the computation of DNNs considering both edge and HPC scenarios. We first propose a novel hardware microarchitecture, called Bison-e, that accelerates important linear algebra operations, including inner-product and convolution, used in a broad range of applications, such as deep learning, pattern matching, and graph processing. Bison-e exploits the use of a mathematical technique, called binary segmentation, to reduce the arithmetic complexity of linear algebra operations involving narrow integers on general-purpose central processing unit (CPU) architectures, achieving high performance through single instruction multiple data (SIMD) operations on off-the-shelf scalar processor functional units (FUs), with minimal area and energy overhead. Our experimental evaluation shows that Bison-e achieves up to $24\times$ better performance for 2-bit data sizes compared to a scalar RISC-V core, and $5\times$ energy efficiency improvement for string matching tasks compared to a RISC-V-based vector processing unit.
(VPU). We then utilize principles from Bison-e to design a novel hardware-software co-designed architecture, called Mix-GEMM, specifically designed to efficiently compute arbitrary quantized DNNs convolutional kernels exploiting data sizes ranging from 8- to 2-bit, including mixed-precision computations. Mix-GEMM enhances a state-of-the-art (SoA) matrix-matrix multiplication framework called BLAS-like library instantiation software (BLIS) with custom instructions extending the RISC-V instruction set architecture (ISA) to perform high-performance convolutional kernels on CPUs. We propose an experimental evaluation performed on representative quantized convolutional neural networks (CNNs) targeting edge and mobile CPUs, demonstrating that Mix-GEMM achieves an energy efficiency of up to 1.3 TOPS/W and throughput of up to 13.6 GOPS, outperforming the considered baseline by 5.3× to 15.1×, while only accounting for 1% of the overall system-on-chip (SoC) area consumption. This thesis finally investigates the main computational bottlenecks of modern HPC-based hardware accelerators, and proposes Flex-SFU, a novel hardware accelerator for complex DNNs activation functions. Flex-SFU serves as lightweight special function units to accelerate activation functions computations performed on application-specific deep learning processors. It utilizes non-uniform piecewise linear (PWL) approximation and supports multiple data formats, enabled by a binary-tree based address decoding unit. The proposed experimental evaluation, considering over 700 computer vision and natural language processing models, demonstrates an average of 22.3× improvement in mean squared error (MSE) compared to previous PWL approaches. Moreover, Flex-SFU improves the end-to-end performance of the considered AI hardware accelerator by 35.7% on average, achieving up to 3.3× speedup with negligible impact on model accuracy. The architecture introduces a modest area and power overhead of 5.9% and 0.8%, respectively, relative to the baseline VPU.
# Contents

1 Introduction ................................................. 1
   1.1 Thesis Overview ........................................... 5
   1.1.1 Contributions ........................................... 5
   1.1.2 Publications ............................................ 7
   1.2 Timeline .................................................... 9
   1.3 Outline .................................................... 10

2 BiSon-e: Accelerating Narrow Integer Linear Algebra Computing on the Edge via Binary Segmentation 11
   2.1 Introduction ............................................... 12
   2.2 Binary Segmentation ........................................ 14
      2.2.1 Inner Product of Two Vectors via Binary Segmentation 15
      2.2.2 Convolution of Two Vectors via Binary Segmentation 17
   2.3 Design Space Exploration ................................. 18
      2.3.1 Inner Product Kernel Analysis ........................ 19
      2.3.2 Linear Convolution Kernel Analysis .................... 22
   2.4 BiSon-e Architecture ..................................... 24
      2.4.1 Enhanced Inner Product Computation .................. 28
      2.4.2 Fused Overlap-Add ................................... 29
   2.5 Experimental evaluation ................................... 30
      2.5.1 Experimental Setup .................................... 30
      2.5.2 Workload Description .................................. 30
      2.5.3 Performance .......................................... 32
      2.5.4 Area and Power Analysis ............................. 35
   2.6 Related work .............................................. 36
   2.7 Discussion ................................................. 37
   2.8 Summary ................................................... 38
3 Mix-GEMM: An efficient HW-SW Architecture for Mixed-precision Quantized Deep Neural Networks Inference on Edge Devices
3.1 Introduction
3.2 Background
3.2.1 Deep Neural Networks Computation
3.2.2 Deep Neural Networks Quantization
3.2.3 Efficient Matrix-Matrix Multiplication
3.3 MIX-GEMM HW-SW Architecture
3.3.1 µ-engine general matrix multiplication (GEMM) Software Library
3.3.2 µ-engine Hardware Architecture
3.3.3 Design Space Exploration
3.4 Experimental Evaluation
3.4.1 Experimental Setup
3.4.2 Performance
3.4.3 Physical Design and Energy Efficiency
3.5 Comparison with State-of-the-Art Solutions
3.5.1 Optimized Software Libraries
3.5.2 Specialized Arithmetic Units
3.5.3 Decoupled DNN Accelerators
3.6 Discussion
3.6.1 Comparison with Bison-e
3.6.2 Performance scalability
3.6.3 Mix-GEMM workflow
3.7 Summary

4 Flex-SFU: Accelerating Deep Neural Networks Activation Functions by Non-Uniform Piecewise Approximation
4.1 Introduction
4.2 Background and Related Work
4.3 Flex-SFU Hardware Architecture
4.4 Flex-SFU Approximation Methodology
4.5 Experimental Evaluation
4.5.1 Performance, Power and Area Analyses
4.5.2 Function Approximation Precision Analysis
4.5.3 End-to-End Evaluation
4.6 Discussion
CONTENTS

4.6.1 Function Approximation Order Trade-Offs 87
4.7 Summary 87

5 Summary and Conclusion 89
  5.1 Overview of the Main Results 90
  5.2 Outlook 91

List of Figures 93
List of Tables 96
List of Algorithms 97
List of Abbreviations 97
Bibliography 101
Chapter 1
Introduction

Over the past decade, artificial intelligence (AI) has made tremendous strides, leading to numerous breakthrough technologies. One of the main driving forces behind these advances is deep learning, a subfield of machine learning that involves deep neural networks (DNNs). Indeed, DNNs have been shown to achieve never-before-seen levels of accuracy in various tasks, such as image recognition, speech recognition, and natural language processing [89] [45] [114]. A remarkable example of how DNNs improved the accuracy of image recognition tasks in the last decade can be clearly perceived in Figure 1.1. Specifically, approaches based on deep learning completely overtake other computer vision solutions starting from 2012 with the introduction of the AlexNet convolutional neural network (CNN) [82], featuring a top-1 accuracy\(^1\) of 63.3\%, and surpassed human-level accuracy in 2015 with the ResNet architecture [66], attaining up to 78.6\% top-1 accuracy. Recent DNN architectures such as CoCa [160] and BASIC-L [37] show even higher accuracies, achieving up to 91\% top-1 accuracy in computer vision tasks.

While advances in DNN accuracies have led to unparalleled achievements, the computational demands associated with these models restrict their applicability to various platforms and applications. These limitations include a constant increase in the overall number of parameters and operations required for their execution, as well as higher complexities in terms of models topologies and types of layers. As detailed in the remainder of this section, these computational drawbacks are posing several challenges to a wide spectrum of computing systems, ranging from small general-purpose central processing unit (CPU) processors targeting edge and mobile applications, to large-scale hardware accelerators designed for cloud and high-performance computing (HPC) environments.

\(^1\)Probability that the target object corresponds to the object class predicted with the highest probability.
Introduction

Figure 1.1: Accuracy over time evolution of 800+ computer vision DNNs [7].

**Models size increase:** The increase in the number of layers of modern DNNs result in an exponential growth of the number of operations and parameters required for their computation. This trend is highlighted in Figure 1.2, analyzing more than 800 state-of-the-art (SoA) DNN models [7] targeting image classification and relating each network accuracy with the corresponding number of operations and parameters needed to perform a single inference task. As Figure 1.2 shows, traversing the models Pareto frontier, aiming at improving quality of results, requires relying on networks featuring more operations and parameters, mainly coming from layers performing dense linear algebra operations, such as the convolution and fully-connected layers. Efficiently handling such a high number of operations and parameters is particularly demanding in the edge computing scenario, exploiting DNNs in a wide range of latency-sensitive applications, such as autonomous vehicles or real-time monitoring of industrial processes [163, 167]. Specifically, as edge systems process data closer to the source, they are tightly constrained by power consumption, with direct limitations in the processor area and memory requirements. As a result, the high number of operations and the large memory footprints of DNNs make their computation on general-purpose edge processors almost impracticable considering reasonable performance and energy constraints. For this reason, exploring novel computing systems capable of fulfilling the performance requirements of DNNs on edge-based applications, while satisfying the energy and memory caps of these systems currently represent a bright research area of interest in both industry [1, 2, 5, 9] and academia [39, 58, 77].

**Models complexity increase:** As a way to improve end-to-end accuracy, modern networks are increasingly exploiting sophisticated computational layers and complex topologies over time. For example, contemporary activation functions, such as the Sigmoid Linear unit (SiLU) and the Gaussian Error Linear unit (GELU) are used more frequently in recent years in place of simpler activation functions like Rectified Linear Unit (ReLU). While these
newer activation functions have been shown to improve the accuracy of DNNs, especially in deep architectures, their computation also involves a wide set of complex operations (e.g., exponentiations, divisions). This trend can be seen clearly in Figure 1.3, analyzing the activation functions distribution in DNNs over the past years, considering more than 600 computer vision DNNs and 150 natural language Processing (NLP) transformers from TIMM [154] and Hugging Face [155], respectively. As Figure 1.3 shows, while ReLU was the dominant activation function from 2015 to 2017, it declined to 20.7% in 2021 and 5% in 2022, while functions like SiLU and GELU emerged over the last years, jointly accounting for 32.1%, 44.2%, and 67.8% of the total activation functions count in 2020, 2021, and 2022, and requiring 4× and 12× more arithmetic operations than ReLU, respectively. While these operations generally represent a minor fraction of the total network computations, they could heavily impact the total network runtime in application-specific HPC and cloud computing systems, well-suited for applications ranging from large-scale image and speech recognition to image/video tagging and machine translation [100,141,142]. Indeed, these computing systems typically leverage on large decoupled accelerators to com-

Figure 1.2: Accuracy against number of operations and parameters for 800+ computer vision DNNs from [7].
Figure 1.3: Activation functions distribution by year of model publication, extracted from 700+ SoA AI models of the TIMM and Hugging Face collections.

DNNs, exhibiting high performance and energy efficiency on specific tasks such as convolutions and matrix-matrix multiplications, but not equally efficient on the computation of the latest DNN operators, and thus less flexible than general-purpose processors in adapting to the complexity increase of modern models.

Because of their criticalities concerning many applications and computing system domains, the analyzed challenges are paving the way for many research opportunities, concerning several areas. For example, many solutions exploit methods to optimize DNN computations by leveraging on mathematical transformations to reduce the computational complexity of DNN kernels [85, 105]. Network quantization and pruning [71, 110, 164] represent other popular techniques to reduce the memory footprint of DNNs with minimal losses in accuracy, either by representing their parameter using narrow integer data formats (typically ranging from 8- to 1-bit), or by removing redundant parameters. Function approximation methodologies [62, 95] are used to reduce the arithmetic complexity of non-linear DNN operators, such as activation functions. Other popular solutions propose algorithmic improvements to map convolutions to dense matrix-matrix multiplication kernels [35], allowing to compute them through highly-optimized implementations and exploiting a wide range of hardware architectures, such as CPUs [157], vector processing units (VPUs) [61], and graphics processing units (GPUs) [40]. Finally, many research works propose application-specific accelerators [38, 151], and instruction set architecture (ISA) extensions [57, 83, 147]. Exploring these challenges remains critical for the continued advancement of deep learning and its successful integration into a range of computing systems, exploring efficient hardware architecture capable of keeping pace with the DNNs advancements is a relevant research challenge in both academia and industry.
1.1 Thesis Overview

In this thesis, we investigate several research opportunities aimed at optimizing the computation of DNNs on modern hardware architectures. Specifically, we explore a novel mathematical transformation aimed at reducing the computational complexity of DNN computational kernels, and we investigate algorithmic improvements intended for optimizing the underlying architecture in terms of compute and memory utilization ratios. We also propose ISA extensions and novel hardware microarchitectures designed to enhance performance, instruction count, and energy efficiency of the computational kernels that, as analyzed in Figure 1.2 and Figure 1.3, represent the main bottlenecks for current DNNs processors.

1.1.1 Contributions

The key contributions of this thesis are summarized as follows:

1. We explore the applicability of a mathematical technique, called binary segmentation on general-purpose CPU architectures. The main benefit of binary segmentation relies on reducing the arithmetic complexity of linear algebra operations between narrow integers. We also propose a novel hardware microarchitecture, called Bison-e, to accelerate linear algebra kernels between narrow integers exploiting binary segmentation. Bison-e allows computing important linear algebra operations, such as the inner-product and the convolution among vectors, representing the core kernels not only of deep learning workloads, but also of other application classes, such as pattern matching and graph processing. Bison-e allows performing single instruction multiple data (SIMD) operations on off-the-shelf scalar processor functional units (FUs), thus achieving high performance with negligible area and energy overheads. We implement Bison-e exploiting the gem5 simulator and in register transfer level (RTL) to perform a complete performance, power, and area evaluation of the proposed solution. Our experimental evaluation shows that Bison-e improves the convolution and fully-connected layers computations of the AlexNet and VGG-16 CNNs up to $5.6 \times$, $13.9 \times$ and $24 \times$ considering 8-, 4-, and 2-bit data sizes compared to the scalar implementation of a single RISC-V core, and improves the energy efficiency of string matching tasks by $5 \times$ when compared to a RISC-V-based VPUs. We also integrate Bison-e into a complete system-on-chip (SoC) based on RISC-V, performing synthesis and place-and-route (PnR) in 65nm and 22nm technologies, and we show that the proposed microarchitecture only introduces a negligible 0.07% area overhead with respect to the baseline architecture.
2. We propose Mix-GEMM, a hardware-software co-designed architecture capable of efficiently computing quantized DNN convolutional kernels based on arbitrary combinations of byte and sub-byte data sizes. Specifically, Mix-GEMM accelerates the matrix-matrix multiplication operation, representing the core kernel of DNNs, supporting all data size combinations from 8- to 2-bit, including mixed-precision computations. On the one hand, Mix-GEMM exploits a software library that enhances the current SoA framework performing high-performance matrix-matrix multiplications on CPU architectures (i.e. BLIS), exploiting custom instructions extending the RISC-V ISA. On the other hand, Mix-GEMM enhance the Bison-e microarchitecture performing SIMD computations among narrow integers, featuring performance that scale with the decreasing of the computational data sizes. Our experimental evaluation, performed on representative quantized CNNs and targeting edge and mobile CPUs, shows that a RISC-V based edge SoC integrating Mix-GEMM achieves up to 1.3 TOPS/W in energy efficiency, and up to 13.6 GOPS per core in throughput, gaining from 5.3× to 15.1× in performance over the OpenBLAS general matrix multiplication (GEMM) framework running on a commercial RISC-V based edge processor. By performing synthesis and PnR of the enhanced SoC in Global Foundries 22nm FDX technology, we show that Mix-GEMM only accounts for 1% of the overall SoC area consumption.

3. We investigate, design, and implement a novel hardware accelerator for complex DNN activation functions. The proposed architecture, called Flex-SFU, extends the set of FUs hosted in deep learning VPUs, which are used as general-purpose co-processors flanking the main matrix multiplication units. Flex-SFU represents a lightweight special function unit targeting the acceleration of activation functions, relying on non-uniform piecewise interpolation and supporting multiple data formats. Non-Uniform segments are enabled by implementing a binary-tree comparison within its address decoding unit. Thanks to these features, Flex-SFU achieves on average 22.3× better mean squared error (MSE) compared to previous piecewise linear interpolation approaches. We evaluate Flex-SFU with more than 700 computer vision and natural language processing models, showing that we can, on average, improve the end-to-end performance of state-of-the-art AI hardware accelerators by 35.7%, achieving up to 3.3× speedup with negligible impact in the models accuracy, and only introducing an area and power overhead of 5.9% and 0.8% relative to the baseline VPU.
1.1.2 Publications

The content of this thesis and the main contributions have been published in the following conference papers:


[125] Enrico Reggiani†, Renzo Andri†, Lukas Cavigelli, ”Flex-SFU: Accelerating DNN Activation Functions by Non-Uniform Piecewise Approximation”, ACM/IEEE Design Automation Conference (DAC 2023)

In particular, the contributions made in Bison-e [128] and Mix-GEMM [129] aim to improve the challenges highlighted in Figure 1.2, considering quantized DNNs with a focus on edge and mobile scenarios. These contributions exploit mathematical transformations (i.e., quantization and binary segmentation), algorithmic improvements of SoA linear algebra frameworks, ISA extensions, and custom hardware accelerators to improve the DNNs computation on edge CPUs considering performance, as well as memory and energy consumption. On the other hand, the contribution made with Flex-SFU [125] focuses on the research challenge detailed in Figure 1.3, by proposing a hardware accelerator for complex activation functions on large-scale tensor units targeting HPC and cloud scenarios.

In addition to conference papers, some of the intellectual properties of this thesis have also been protected by filing a patent application. This patent application represents a valuable addition to the thesis, as it demonstrates the practical implications and potential real-world impact of our research findings:

[EP22382173.7] Enrico Reggiani, Adrián Cristal, Osman Sabri Unsal, ”Method for the computation of a narrow bit width linear algebra operation”

†Both authors contributed equally to this research. This work was done during Enrico Reggiani’s internship at Huawei Zurich Research Center.
The following contributions have been made in addition to the aforementioned conference publications and patent application. These contributions are not directly included in this thesis either because they are not aligned with the topic of the thesis [32, 109, 126] or because the contribution was limited to providing ideas and helping with the write-up [87, 152].


1.2. Timeline

A Gantt chart summarizing the timeline of the Ph.D. activities is reported in Figure 1.4. We divide the tasks into 4 main activities, reporting the technical Ph.D. activities, the engineering activities, the periods spent abroad for research internships, and the effort for preparing, submitting, and revisioning conference papers, patents and journals. We assign different colors for each activity type, exploiting the same color among activities related to the same task. Specifically, we colored in red, green, and orange all the tasks related to contribution 1 (i.e., Bison-e), contribution 2 (i.e., Mix-GEMM), and contribution 3 (i.e., Flex-SFU), respectively. As highlighted in Figure 1.4, the first 7 months of the Ph.D. (i.e., from October 2019 to April 2020) has been dedicated to research activity tasks in the context of the European Processor Initiative (EPI) project [6], and to an internship period of 4 months at Microsoft Research Labs in Cambridge (UK), working with the System and Networking research group on the performance enhancement of the next-generation cloud infrastructure. From the second quarter (i.e., Q2) of 2020 to the first quarter (i.e., Q1) of 2021 I worked on the thesis proposal, and on the development of the first contribution of this Ph.D. thesis (i.e., Bison-e), whose results [128] have been published in the ASPLOS 2021 conference. From Q4 of 2020 to Q1 of 2021 I have also been involved in the design
of a RISC-V VPU in the context of the EPI and DRAC [4] projects, whose research results have been published in DCIS 2022 and TACO 2023 [32, 109]. From Q3 of 2021 to Q1 of 2022 I explored the second contribution of this Ph.D. thesis (i.e., Mix-GEMM), whose results have been published in the HPCA 2023 conference [129], and taped out in the context of the DRAC project. During the third quarter of 2021, I also worked, in the context of the DRAC project, on the design and implementation of a high-speed Serializer/Deserializer (SerDes) interconnect. Finally, from Q2 of 2022 to Q1 of 2023, I investigated the third contribution of this Ph.D. thesis (i.e., Flex-SFU), in collaboration with Huawei Research, Zurich (CH) during a research internship period. The results of this collaboration have been published in the DAC 2023 conference [125].

1.3 Outline

The thesis is structured as follows:

In Chapter 2, we present Bison-e, proposing an in-depth design space exploration (DSE) of the binary segmentation technique applied to 32-bit and 64-bit computing systems (Section 2.3), detailing the Bison-e microarchitecture (Section 2.4), and discussing the performed experimental evaluation (Section 2.5). Chapter 3 details Mix-GEMM, discussing its hardware-software co-designed architecture (Section 3.3) and comparing its performance, area, and energy efficiency with the related work (Section 3.4 and Section 3.5). In Chapter 4 we propose Flex-SFU, discussing the rationale behind the work and the comparison with other SoA proposals (Section 4.2), detailing the specification of the proposed solution (Section 4.3 and Section 4.4), and proposing its experimental evaluation (Section 4.5). Finally, in Chapter 5 we summarize the main contributions and results of the thesis, and we draw the conclusions.
Chapter 2

BiSon-e: Accelerating Narrow Integer Linear Algebra Computing on the Edge via Binary Segmentation

In this chapter, we explore techniques for enhancing the computational efficiency of linear algebra kernels by leveraging the binary segmentation method [119]. This method enables a substantial reduction in both memory usage and arithmetic intensity when dealing with linear algebra kernels that involve narrow data sizes. We conduct a comprehensive DSE (Section 2.3), to evaluate the applicability of binary segmentation on 32-bit and 64-bit CPU architectures, exploring its key benefits and limitations. Furthermore, we propose a design methodology (Section 2.4) that enables the acceleration of linear algebra kernels based on narrow integer computations exploiting binary segmentation, that employs SIMD operations on off-the-shelf scalar FUs. We implement our methodology in the form of a hardware accelerator, called Bison-e, tightly coupled with a RISC-V processor specifically designed for edge scenarios. Our experimental evaluation (Section 2.5) demonstrates the efficacy of our solution in improving execution time for the convolution and fully-connected layers of AlexNet and VGG-16 CNNs compared to a scalar implementation performed on the baseline RISC-V core, and shows a better energy efficiency on string matching tasks when compared to a RISC-V-based VPU, while incurring in negligible area overhead compared to the baseline SoC.
2.1 Introduction

Contemporary internet-of-things (IoT), edge and mobile computing applications require high performance. This demand is fueling a large research effort in low-power, high-performance embedded processors [78, 137]. Such devices, mainly constrained by power and cost, have to fulfill the performance and memory requirements of a vast collection of important application domains, including deep learning but also spanning to other areas, such as robotics, graph processing, and cryptography. Most of these application classes represent data as matrices and vectors, and express their computation through a set of linear algebra kernels.

When targeting edge platforms, a popular approach to reduce energy demands and memory footprint requirements is to compact the data layout using a smaller data format while preserving the application accuracy. On the one hand, expressing and computing data exploiting low-precision floating-point formats [13] is gaining traction in the high-performance edge computing (HPEC) community, as they represent a good trade-off between data size and accuracy. On the other hand, narrow fixed-point and integer data representations (i.e., byte and sub-byte) offer a better alternative in terms of Performance per Watt ratio, although they feature smaller number representations. One of the dominant applications of edge computing that leverages these compressed data formats is the quantized neural network (QNN) inference, which exploits quantization to represent data and weights with data sizes typically ranging from eight to one bit with tolerable accuracy penalties [99, 111]. Other kernels belonging to important application classes for edge computing, such as graph computing and cryptography, widely rely on boolean matrix and vector computations to traverse a graph or to encrypt/decrypt a message. These applications would greatly benefit from hardware and software solutions that efficiently compute narrow integer linear algebra kernels. However, despite their low memory and energy demands, their intrinsically high computational intensity makes the execution of these workloads challenging on highly resource-constrained devices.

Accordingly, we present *Bison-e* \(^1\), a high-performance and lightweight architecture aimed at increasing the efficiency of linear algebra narrow integer computations on edge processors. The proposed solution relies on a mathematical technique called *binary segmentation* [119], which reduces the memory footprint of matrices and vectors consisting of narrow integers, and considerably decreases the arithmetic complexity of linear algebra computations. To the best of our knowledge, this is the first work developing a *binary segmentation* based architecture. *Bison-e* is motivated by the lack of sufficient support for

\(^1\)Binary Segmentation on-edge
efficient narrow computations in current edge processors and ISAs, as most of them do not implement memory and arithmetic instructions for data formats smaller than 8-bit. For example, compressing sub-byte data in memory needs a conversion to standard bitwidths before and after each computation, leading to performance and energy consumption inefficiencies. Moreover, processor FUs are overprovisioned for computations involving narrow data sizes, and exhibit an energy-per-instruction that does not scale with the input data size. Our key contribution is to increase the efficiency of narrow data formats in terms of data storage and linear algebra kernel computations, scaling their performance with the decrease of the data size. Instead of extending standard RISC-V ISA for new sub-byte data sizes, and designing custom hardware supporting them, we rely on data segmentation to fuse multiple arithmetic operations in a single instruction, performing SIMD computations on off-the-shelf scalar FUs.

The main contributions of this work are summarized as follows:

- We perform a DSE of binary segmentation on 64-bit architectures. Guided by DSE, we design the Bison-e architecture which features a binary segmentation enhanced CPU pipeline. We analyze a set of linear algebra computational kernels that can leverage binary segmentation to increase the performance of edge-based narrow integer applications;

- We benchmark Bison-e with three algorithms belonging to two demanding edge computing application classes, namely deep learning, and string matching, considering both performance and energy efficiency. Our solution improves the back-to-back runtime performance of the AlexNet and the VGG-16 CNNs by a factor that ranges from $5.6 \times$ to $24 \times$ on 8-bit and 2-bit data sizes with respect to the single-core scalar implementation, and outperforms the string matching use-case vectorized implementation by a factor of $5 \times$ in terms of energy efficiency;

- We integrate, design, and fully implement the proposed architecture, including PnR, on a RISC-V based SoC, in both 65nm and 22nm technologies. We show that Bison-e can be integrated into modern edge processors with a negligible 0.07% area overhead, and without any performance loss;

The remainder of this chapter is organized as follows. Section 2.2 presents the binary segmentation technique. Section 2.3 performs a DSE of binary segmentation on 64-bit CPUs. Section 2.4 details the Bison-e architecture, discussing its design choices and features. Section 2.5 evaluates the experimental results obtained with the proposed solution. Section 2.6 reviews the main related work. Finally, Section 2.8 summarizes Bison-e.
2.2 Binary Segmentation

In the class of applications requiring narrow integer computations, the data size needed by the algorithm is typically lower than the one allowed by the underlying architecture. Modern processors datapaths are often based on 32-bit or 64-bit, and thus byte and sub-byte computations underutilize both their arithmetic capabilities and data movement efficiency. Moreover, the current ISAs and programming languages typically lack adequate support to handle narrow data bitwidths. Consequently, the performance of workloads featuring narrow integer computations does not scale in concert with the data size. This thesis chapter explores the binary segmentation technique to reduce these limitations. Binary segmentation [118] is a mathematical technique that allows reducing the arithmetic complexity of basic linear algebra subprogram (BLAS) computations based on narrow integers data [26, 34, 133]. This technique abstracts the computation of BLAS kernels based on narrow integers as simpler arithmetic operations, by properly representing sets of narrow integer elements as single wider data, called input-clusters. Specifically, this technique allows performing SIMD computations of kernels featuring narrow integers, exploiting the unmodified processor FUs, such as scalar multipliers and adders.

According to this technique, an n-dimensional vector $v = [v_0, \ldots, v_{n-1}]$ populated with integers in the $[0, 2^b)$ range, with $b$ denoting the element bitwidth can be represented by the single integer $V_b$ (i.e., a input-cluster):

$$V_b = \sum_{i=0}^{n-1} v_i 2^{bi} \quad (2.1)$$

This interpolation allows creating a compact storage scheme for matrices and vectors populated with bounded integers, as a single computer word can be composed of several elements belonging to $v$. Enhanced support for lower data sizes would dramatically decrease the applications memory footprint. For example, by following Equation (2.1), applications requiring storing large boolean matrices and vectors on a 64-bit architecture would decrease their memory footprint by $8 \times$, by packing sixty-four elements in a single memory location. However, in modern processors, the lower bound of data sizes that inherently support Equation (2.1) is often in the byte range. As a result, the advantages offered by the compact storage scheme described in Equation (2.1) for sub-byte data sizes can be mitigated by the overhead needed to pack and extract data from non-standard data sizes before and after their computation. Moreover, adding support for narrow data sizes could be a demanding task, as it would imply changes at the hardware, ISA, compiler, and software level. With Bison-e, we propose a novel approach to efficiently represent sub-byte data sizes via binary segmentation, and to compute linear algebra arithmetics with minimal changes in
2.2. BINARY SEGMENTATION

hardware and ISA, without the need to define new data formats at the software level. Indeed, the binary segmentation technique has been successfully explored, from a theoretical perspective, to decrease the arithmetic complexity of several arithmetic expressions: polynomial multiplication [52], multiplication of two complex numbers [118], discrete Fourier transform (DFT) [133], inner and outer product of two vectors [119], polynomial division [26], and polynomial greatest common divisor (GCD) [34], and supports both signed and unsigned computations [133]. As a simple example of how this technique can be used to compute arithmetic operations on matrices and vectors, we can consider the sum of two vectors \( u \) and \( v \), both composed of elements in the \([0, 2^b)\) range. Following Equation (2.1), and defining the clustering width \( (cw) \) as \( cw = b+1 \), we can create two integers \( U_{cw} \) and \( V_{cw} \) via binary segmentation, sum them as a single sum of integers, and recover the output vector, obtaining the element-wise sum of the two vectors. The \( cw \) is defined to be greater than the actual bitwidth of the input elements \( b \), as it includes extra guard-band bits to avoid overflows in the segmented data due to carry propagation. This allows performing the summation of \( n \) narrow integers with only one summation of two large integers, instead of \( n \) summations of short integers (i.e., 8-bit and below). It is worth noticing that binary segmentation is not an approximate computing technique, since it guarantees exact computations, as the \( cw \) dimension already accounts for the number of bits needed to represent the computation output without introducing precision losses.

Below, we describe how binary segmentation can improve the efficiency of representative linear algebra kernels, namely inner product (IP) and linear convolution (LC).

2.2.1 Inner Product of Two Vectors via Binary Segmentation

The IP of two vectors composed of \( n \) elements \( u = [u_0, \ldots, u_{n-1}] \) and \( v = [v_0, \ldots, v_{n-1}] \), having bitwidths \( b_u \) and \( b_v \), can be obtained by the following expression:

\[
\text{IP} = \sum_{i=0}^{n-1} u_i v_i
\]  

(2.2)

To compute the IP via binary segmentation, it is first necessary to reverse the vector \( v \) such that:

\[
v'_i = v_{n-1-i}, \quad i = 0, 1, \ldots, n - 1
\]  

(2.3)

According to Equation (2.1), we create the integers \( U_{cw} \) and \( V_{cw} \) from \( u \) and \( v' \), with the following \( cw \):

\[
cw \geq b_u + b_v + \lceil \log_2(n) \rceil
\]  

(2.4)
2.2. BINARY SEGMENTATION

Figure 2.1: Examples of IP (a) and LC (b) kernel computations via binary segmentation, with clustering widths of 7-bit and 5-bit, respectively. The input vectors are represented with clustering widths bits (blue), merged into single variables (green), and multiplied (yellow). The final result is then extracted from the multiplication output (red).

Then, the IP computed via binary segmentation is the multiplication between \( U_{cw} \) and \( V_{cw} \), resulting in the integer \( W_{cw} \). The IP result is derived from \( W_{cw} \) by extracting the bits expressed as follows:

\[
IP = W_{\{(n-1)_{cw}+cw, (n-1)_{cw}\}}
\]  

(2.5)

Considering the reference example depicted in Figure 2.1a, we can evaluate the IP between \( u = [7, 5] \) and \( v = [4, 2] \) via binary segmentation employing a single integer multiplication. Specifically, we represent each element of the input vector with a bitwidth equal to the \( cw \) defined in Equation (2.4) (i.e., 7-bit), and we revert the order of the elements belonging to \( v \) (blue). Then, we express the resulting vectors as single integers (green), and we perform their multiplication (yellow). Finally, we extract the IP result from the seven bits resulting from Equation (2.5) (red). For this example, we employed a single integer multiplication in place of two multiplications and one sum to obtain the final result. As detailed in Section 2.3, the same approach can be used to compute the IP between three to ten elements concurrently on a 64-bit architecture, for input sizes ranging from 8-bit to 1-bit.
2.2. BINARY SEGMENTATION

2.2.2 Convolution of Two Vectors via Binary Segmentation

Given \( u = [u_0, \ldots, u_{m-1}] \) and \( v = [v_0, \ldots, v_{n-1}] \), we compute the vector \( w = [w_0, \ldots, w_K] \), having length \( K = m+n-1 \), as the LC between \( u \) and \( v \):

\[
w_k = \sum_{i=0}^{K} u_i v_{k-i}, \quad k = 0, 1, \ldots, K - 1 \tag{2.6}
\]

The same expression can be computed via binary segmentation by representing \( U_{cw} \) and \( V_{cw} \) as in Equation (2.1), with a \( cw \) of:

\[
cw \geq b_u + b_v + \left\lceil \log_2(\min\{m,n\}) \right\rceil \tag{2.7}
\]

Then, we recover \( w \) from the output of the integer multiplication between \( U_{cw} \) and \( V_{cw} \). An example of LC between two vectors \( u = [3, 1, 2] \) and \( v = [1, 2] \) is shown in Figure 2.1b. First, the bitwidth of each element belonging to \( u \) and \( v \) is represented as a 5-bit number, according to Equation (2.7) (blue). Then, the two input vectors are converted to single integers (green) and multiplied (yellow). By segmenting the multiplication result into four 5-bit binary numbers, it is possible to recover the LC result\(^2\) (red). This example only uses one multiplication to compute the LC between \( u \) and \( v \), which would have required six multiplications and two additions to be computed with Equation (2.6).

The IP and LC examples reported in Figure 2.1 reduce their arithmetic complexity by a factor of 3\times and 7\times, respectively. Certainly, the ratio between the processor word size and the data size highly impacts the achievable arithmetic complexity reduction. Moreover, if the vector length does not fit the processor word size, the arithmetic problem must be partitioned into smaller-size subproblems. It is also worth considering the effort required to convert a set of vector elements into a single integer, and to extract the output elements from the integer multiplication result. These requirements increase the overall arithmetic complexity of binary segmentation if the underlying architecture is not efficient in clustering, extracting, and masking data, leading to a decrease in the overall performance gain.

We deeply explore and quantify these considerations in Section 2.3, while in Section 2.4 we show how our architecture overcomes these limitations, allowing narrow integer linear algebra kernels to fully benefit from the advantages that binary segmentation offers.

\(^2\)LC\(_{out} = [3 \times 1, 1 \times 1 + 3 \times 2, 2 \times 1 + 1 \times 2, 2 \times 2] \)
2.3 Design Space Exploration

The proposed DSE aims to explore the benefits and the pitfalls of implementing binary segmentation on edge processors, exploiting standard CPU architectures. The efficiency of binary segmentation strictly depends on the ratio between the CPU registers size and the application data bitwidths, as described in Equation (2.8).

\[
\text{input-cluster}_{\text{size}} = \frac{\text{CPU bitwidth}}{cw}
\]

(2.8)

On the one hand, the greater this ratio is, the larger the number of elements embedded in a single operation. On the other hand, increasing the number of elements clustered in a single register implies a higher overhead required to pack data into single integers, and to extract the results from the multiplication output. Following Section 2.2, our evaluation mainly focuses on the IP and LC, as they represent the core kernels of our benchmarks. However, the proposed methodology can be extended and applied to other arithmetic operators that would benefit from this approach, such as the one listed in Section 2.2.

To characterize the binary segmentation technique on CPU architectures, it is important to define the number of elements that can be computed concurrently. We denote this set of elements as input-cluster, and we evaluate the input-cluster\(_{size}\) on both 32-bit and 64-bit CPU architectures. As a reference, Figure 2.1a has input-clusters composed of two elements, while Figure 2.1b features asymmetric input-clusters of three and two elements. In this work, we consider the same input-cluster\(_{size}\) for each operand of the FU. This choice is optimal for CPUs architectures, as they are equipped with symmetric FUs. Moreover, the \(cw\) of the IP and LC input-clusters, defined in Equation (2.4) and Equation (2.7), is reduced to the same expression if the input-cluster\(_{size}\) of the two input vectors is the same. However, architectures featuring asymmetric multipliers, like field programmable gate arrays (FPGAs) [91], could benefit from having asymmetric input-cluster\(_{size}\).

The maximum number of elements composing an input-cluster can be derived as the ratio between the CPU register bitwidth and the clustering width:

Figure 2.2 reports the maximum input-cluster\(_{size}\) on 32-bit and 64-bit registers, for input bitwidths ranging from 1-bit to 16-bit. As Figure 2.2 shows, the input-cluster\(_{size}\) dimension is inversely proportional to the input data size. Specifically, a 32-bit architecture handles from two 7-bit to six 1-bit input data concurrently for the selected kernels, while a 64-bit architecture can compute from two 15-bit to ten 1-bit elements concurrently. Thus, when the ratio between the underlying hardware architecture and the target data size is wide enough, using binary segmentation allows computational concurrency. Specifically, data concurrency is exploited when the input-cluster\(_{size}\) is equal or greater than two, as
2.3. DESIGN SPACE EXPLORATION

Figure 2.2: Maximum input-cluster size achievable on 32-bit and 64-bit architectures, for data sizes ranging from 1-bit to 16-bit. The input-cluster size is defined as the number of elements that can be packed in a single register, following the binary segmentation constraints.

multiple data are processed in parallel using a single operation. According to Figure 2.2, this concurrency starts to be effective for 15-bit data sizes on 64-bit architectures, and it is supported on 32-bit architectures for data ranging from 1-bit to 7-bit.

Figure 2.2 also shows that the number of elements composing the input-cluster is rarely improving the kernels memory footprint. Indeed, only 1-bit input-clusters can hold more elements than a conventional byte-based allocation. For this reason, creating the input-clusters before each computation while keeping data compressed in memory would be beneficial from a memory consumption perspective. However, such data manipulation would increase the computational cost of performing binary segmentation, reducing its overall arithmetic complexity improvement. These considerations are analyzed in detail in Section 2.3.1 and Section 2.3.2. We focus our study on 64-bit architectures, as they are capable of supporting more data sizes than the 32-bit ones.

2.3.1 Inner Product Kernel Analysis

From Equation (2.2), we can notice that computing the IP of two vectors having length \( n \) requires \( n \) multiplications and \( n-1 \) additions. As discussed in Section 2.2, we can perform the same computation by means of a single multiplication, as long as the input-cluster can hold \( n \) elements. We use the maximum input-cluster size for every considered data size reported in Figure 2.2 to derive the arithmetic complexity decrease, defined as the ratio between the
2.3. DESIGN SPACE EXPLORATION

Arithmetic Complexity Reduction

(a) IP arithmetic complexity reduction
(b) LC arithmetic complexity reduction

Figure 2.3: Arithmetic complexity reduction when computing IP (a) and LC (b) kernels on 64-bit architectures, accounting for multiplications and additions.

arithmetic operations (i.e., multiplications and additions) needed to compute the IP kernel by using either Equation (2.2) or binary segmentation. As shown in Figure 2.3a, the analyzed technique can save a considerable number of arithmetic operations to compute the IP kernel. Specifically, the number of multiplications and additions required by the binary segmentation technique is reduced from a $3 \times$ for 15-bit computations, to a $19 \times$ for 1-bit computations.

Although this reduction can have a huge impact on linear algebra kernels computing IPs, the implementation of this technique exploiting standard ISAs leads to sub-optimal results, mostly due to their inability to efficiently support non-standard bitwidth data manipulations. Indeed, each element of the input-cluster needs to be converted to non-standard bitwidths (i.e., the $cw$) to respect Equation (2.4). As a result, the input-cluster creation becomes the bottleneck of the IP kernel using binary segmentation. Figure 2.4a reports the profiling of the IP kernel execution, computed via binary segmentation on 64-bit architectures, for the input-cluster width defined in Figure 2.2. We split the arithmetic operations into three main categories: data are firstly pre-processed through Pack instructions to create the input-clusters, then a Multiply operation performs their IP computation, whose result is filtered by the Extract operation. In particular, we accounted for one left-shift and one bitwise OR to process each element of the input-clusters, and for one right-shift and one bitwise AND to extract the result from the multiplication output. Ideally, the Multiply phase of Figure 2.4a should cover the whole execution time percentage, enabling the performance improvements of Figure 2.3a. However, Figure 2.4a shows that the number of instructions
required for the actual IP computation using binary segmentation is minimal, while the greatest contribution is attained by the pre-processing phase, whose purpose is to create the input-clusters. Aiming to alleviate the data pre-processing overhead introduced in the Pack phase, we also implemented custom bit-manipulation instructions, namely PACK and MASK, to quickly create the input-cluster and extract a specific data slice from the output result. These instructions are common in ISAs bit-manipulation extensions [60]. As an example, Algorithm 1 reports the IP between two 8-bit vectors, \(v_0\) and \(v_1\), composed of \(\dim\) elements. Each input-cluster (i.e., \(\ic_0\) and \(\ic_1\)) is composed of three elements, each one having a width equal to the \(\cw\). The implementation represented in Algorithm 1 requires three PACK instructions per input-cluster, instead of three left-shift and three bit-wise OR per input element. The created input-clusters are then multiplied, and the extracted result (i.e., \(\ip_i\)) is accumulated to produce the final IP result. Since a single loop iteration computes the partial IP among three elements, the total number of iterations amount to the ratio between the vector length and the input-cluster size (i.e., \(\ic_{\dim}\)). However, as further discussed in Section 2.5, our evaluation reports that these bit-manipulation instructions can slightly increase the time spent in the Multiply phase of Figure 2.4a, by a factor ranging from 4.5% to 15% for 1-bit and 15-bit data, respectively. We tackle this challenge by proposing an enhanced architecture to compute the IP kernel via binary segmentation. As detailed in Section 2.4.1, we implemented Bison-e to fuse both the input-cluster creation, the multiplication, and the output extraction into a single operation. We also exploit the data compression scheme offered by binary segmentation in Equation (2.1) to reduce the memory movements of this kernel.
2.3. DESIGN SPACE EXPLORATION

Algorithm 1 IP exploiting binary segmentation and bit-manipulation instructions.

1: procedure INNER PRODUCT
2: \( ip_{lsb} = (i_{csize} - 1) \times cw \)
3: \( ip_{msb} = extract_{lsb} + cw \)
4: \( ip = 0 \)
5: for \( i = 0; i < v_{dim}; i+ = i_{csize} \) do
6: \( Pack(ic_0, v_0[i], 0) \) \( \triangleright \) create input-clusters: Pack(Rd, Rs1, shiftamount)
7: \( Pack(ic_0, v_0[i+1], cw) \)
8: \( Pack(ic_0, v_0[i+2], 2 \times cw) \)
9: \( Pack(ic_1, v_1[i], 2 \times cw) \)
10: \( Pack(ic_1, v_1[i+1], cw) \)
11: \( Pack(ic_1, v_1[i+2], 0) \)
12: \( m_{out} = ic_0 \times ic_1 \) \( \triangleright \) actual IP computation
13: \( Extract(ip, m_{out}, ip_{msb}, ip_{lsb}) \) \( \triangleright \) extract partial IP: Extract(Rd, Rs1, msb, lsb)
14: \( ip+ = ip_1 \) \( \triangleright \) Accumulate

2.3.2 Linear Convolution Kernel Analysis

We can similarly analyze the LC kernel expressed in Equation (2.6), by counting the number of multiplications

\[
LC_{mul} = m(n - m + 1) + 2 \sum_{i=1}^{m-1} i, \quad m \leq n
\]  

(2.9)

Similarly, the number of additions needed to sum the partial multiplication results is given by:

\[
LC_{add} = (m - 1)(n - m + 1) + 2 \sum_{i=0}^{m-2} i, \quad m \leq n
\]  

(2.10)

Both Equation (2.9) and Equation (2.10) are composed of two main parts: the first contribution is given by the central body of the convolution, while the summation represents the computations needed to execute the convolutions head and tail. Figure 2.3b shows the LC arithmetic reduction on 64-bit architectures, for the input-cluster size defined in Figure 2.2, with data sizes ranging from 1-bit to 15-bit. The complexity reduction is defined as the ratio of arithmetic operations required to solve the LC exploiting either the reference or the binary segmentation-based implementations. From Figure 2.3b, it can be noted that computing a 2 \( \times \) 2 LC among 15-bit integers induces a 2.5 \( \times \) arithmetic saving, while a 10 \( \times \) 10 LC of boolean data obtains a 90.5 \( \times \) arithmetic reduction with respect to a standard implementation. As Figure 2.3b shows, the arithmetic complexity reduction of LC is greater than the IP one. This is because, as reported in Section 2.2, the LC implementation via binary segmentation produces a complete sub-vector on every iteration, while the IP
2.3. DESIGN SPACE EXPLORATION

Algorithm 2 LC exploiting binary segmentation.

1: procedure LINEAR_CONVOLUTION
2: for (i = 0; i < m/\text{ic\_size}; i++) do
3: \hspace{1em} CREATE_{IC}(ic_0, \&V_{in0}[i \times \text{ic\_size}])
4: \hspace{1em} for (j = 0; j < n/\text{ic\_size}; j++) do
5: \hspace{2em} if i == 0 then
6: \hspace{3em} CREATE_{IC}(ic_1, v[j], \&V_{in1}[j \times \text{ic\_size}]) \hspace{1em} \triangleright \text{buffer input-cluster}
7: \hspace{2em} m_{out_l} = ic_0 \times ic_1[j] \hspace{1em} \triangleright \text{actual C1D low computation}
8: \hspace{2em} m_{out_h} = MULH(ic_0, ic_1[j]) \hspace{1em} \triangleright \text{actual C1D high computation}
9: \hspace{2em} CREATE_{OC}(oc_v, m_{out_l}, m_{out_h}) \hspace{1em} \triangleright \text{extract the output-cluster}
10: OVERLAP-ADD(LC_v, oc_v) \hspace{1em} \triangleright \text{accumulate using overlap-add}

only produces one output element per iteration (i.e., the inner product). As a result, the number of arithmetic operations performed with a single integer multiplication in LC is greater than the IP one.

As the number of output elements of each LC computation, called output-cluster, is greater than the input-cluster (i.e., n+m-1), to recover each output-cluster, for the input-cluster\_size reported in Figure 2.2, it is necessary to perform two separate multiplications, computing the low and high slices of the multiplication, respectively.

Algorithm 2 reports the pseudo-code of the binary segmentation-based LC between two input vectors $V_{in0}$ and $V_{in1}$, whose lengths are $m$ and $n$.

The inner-most loop of Algorithm 2 computes the LC among the $i$-th input-cluster belonging to $V_{in0}$ (i.e., ic0) and the whole $V_{in1}$ vector. The createIC function creates both the input-clusters, exploiting either bitwise instructions (i.e., left-shift and OR) or a set of PACK instructions. Since the resulting output-cluster is divided among the two multiplications output, the createOC function extracts the result of each ic\_dim \times ic\_dim convolution, composed of ic\_dim+ic\_dim-1 elements, from the two multiplications output, creating the oc\_v vector. Finally, the OVERLAP-ADD method [63] composes the LC output vector, called LC\_v. Since each output-cluster represents a segment of LC\_v, the OVERLAP-ADD method accumulates each segment into a given position of LC\_v.

From Algorithm 2, we can notice that the LC offers more data-reuse possibilities at the input-cluster level than the IP kernel, as the first inner-most loop iteration creates the LC\_v, that is reused till the program ends. Instead, the advantages offered by binary segmentation for LC are mitigated by the extra computation required to extract LC\_v and to perform overlap-add. Indeed, for every inner-most loop iteration, it is required to extract ic\_dim+ic\_dim-1 elements from the multiplication results, storing them into the LC\_v vector, and compute the element-by-element addition between LC\_v and a segment of oc\_v. This overhead is reported in Figure 2.4b, showing the percentage of time spent on each phase.
of the computation. Specifically, we can note that extracting the output-cluster from \( LC_v \) takes roughly the 50% of the total execution time, and that the overlap-add kernel takes on average 26% of the overall time to be computed. As detailed in Section 2.4.2, we propose an enhanced implementation of the LC algorithm, that improves the analyzed limits by properly creating the output-cluster, allowing to skip the \( LC_v \) extraction and to compute the overlap-add kernel exploiting binary segmentation.

**2.4 BiSon-e Architecture**

The architecture proposed within this research work has been built on top of the binary segmentation technique, presented in Section 2.2. Although this technique has proved its strength to optimize memory compactness and arithmetic complexity of integer linear algebra kernels, to the best of our knowledge, this is the first work that investigates it from a computer architecture perspective. As Section 2.3 shows, exploiting binary segmentation naively leads to practical inefficiencies, mainly due to the standard ISAs and architectures lack of support for non-standard data sizes bit-manipulation operations. In this section, we tackle this problem by presenting BiSon-e, a lightweight architecture that enables exploiting binary segmentation on resource-constrained devices. Specifically, we firstly detail the microarchitecture of BiSon-e, and we describe the defined RISC-V ISA extension. Then, Section 2.4.1 and Section 2.4.2 highlight the benefits of the proposed solution on the enhancement of IP and LC kernels, respectively.

**BiSon-e** extends general-purpose ISAs with instructions facilitating narrow integer computations by leveraging the extremely area-efficient binary segmentation idea. The insight behind BiSon-e is to fill the gap between application-specific accelerators and SIMD/Vector units for die-area sensitive edge computing use cases. On the one hand, as detailed in Section 2.5, BiSon-e is comparably more efficient than a high-performance VPU for narrow integer computations, while featuring \( 600 \times \) less area overhead. On the other hand, BiSon-e features more flexibility than an application-specific accelerator, as it can be used for any kernel exploiting SIMD-style narrow computations. BiSon-e is efficient for modern edge computing systems for the following reasons. It leverages existing FUs on scalar architectures to provide a more flexible integer compute fabric than SIMD architectures. Its flexibility implies a better fine-tuning of the data sizes involved in the computation than standard narrow-SIMD units, which rarely support arithmetic and memory instructions for data formats below 8-bits, and typically neither cover all the possible data size granularities nor support mixed-precision computations. As an example, the current RISC-V vector extension [131] has deprecated its support for narrow-SIMD computations (i.e., Zvediv),
whose initial specifications only accounted for 8-bit, 4-bit, 2-bit, and 1-bit data types. As opposed to standard SIMD units, Bison-e allows every data size discussed in Section 2.3 to be kept compressed in memory, and computed in a SIMD fashion, without incurring data manipulation related area overheads. Bison-e also features mixed-precision computation support by design, as the clustering widths of Equation (2.4) and Equation (2.7) already account for different data sizes between the data sources. Thus, Bison-e is capable of computing compressed data and performing flexible SIMD-style computations, whose width is proportional to the data size of every operation operand, without associated overheads. Moreover, its execution embeds the usage of complex instructions (e.g., multiply-add), with performance gains and high data compressions that can be fine-tuned up to the bit granularity. Moreover, implementing Bison-e, whose key novelty relies on hardware reutilization, does not require any additional datapath, or a separate register file (RF) or FU, leading to a negligible area and power overheads.

As detailed in Section 2.3.1 and Section 2.3.2, the main bottleneck of implementing binary segmentation on standard architectures is either represented by the pre-processing phase, responsible for the input-clusters creation, or by the post-processing phase, whose main purpose is to extract the output from the segmented data and perform accumulations using the overlap-add method. On the contrary, binary segmentation does not affect the computation complexity, as it can rely on standard arithmetic units (e.g., integer multi-
2.4. BISON-E ARCHITECTURE

Table 2.1: Bison-e Control parameters list

<table>
<thead>
<tr>
<th>Input Data</th>
<th>Input Cluster</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwidth</td>
<td>N.Elements</td>
<td>Bitwidth</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>6</td>
</tr>
</tbody>
</table>

| Pack | 8 | 8 | 1 | 64 | - | 8 |
| 8 | 8 | 2 | 32 | - | 4 |
| 8 | 8 | 4 | 16 | - | 2 |

As an example, when configuring the control register with the parameters listed in the first row of Table 2.1, the extend-unit expects eight 8-bit elements in both src1 and src2, and creates the input-clusters composed of three 21-bit elements. The Pre-Proc parameter, defined in Table 2.1, is used to cyclically offset the src1 and src2 registers content, depending on the cnt,i value, spanning from 0 to Pre-Proc - 1. Indeed, as expressed in the first row of Table 2.1, a single 64-bit register containing eight elements requires three iterations (i.e., clock cycles) to be completely computed, each one processing three elements of the input registers. The input-clusters are forwarded to the multiplier through the ic_1 and ic_2 output busses. Then, the multiplication result is processed by the mask-unit, which composes the final result depending on the instruction opcode. Specifically, the mask-unit extracts data in the range expressed in Equation (2.5) if a bs.ip() instruction is decoded, while it outputs either the lower or the higher part of the convolution in case of a bs.lc.l() or a bs.lc.h() instruction.
Table 2.2: Overview of the Bison-e custom instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bs.set()</td>
<td>control registers configuration</td>
</tr>
<tr>
<td>bs.pack()</td>
<td>pack $n$ elements from Rs1 and Rs2</td>
</tr>
<tr>
<td>bs.ip()</td>
<td>returns the Inner Product</td>
</tr>
<tr>
<td>bs.lc.l()</td>
<td>returns the lower slice of the Linear Convolution</td>
</tr>
<tr>
<td>bs.lc.h()</td>
<td>returns the higher slice of the Linear Convolution</td>
</tr>
</tbody>
</table>

To speed up the data compression phase, Bison-e implements the pack-unit, which compresses its input data into their actual data sizes. The pack-unit functionality is inferred by the bs.pack instruction, listed in Table 2.2. The source operands of this instruction contain the input register to be compressed (i.e., src1) and the final compressed register (i.e., src2).

Each instruction call converts the src1 elements into the target output bitwidth, and forwards both the results and src2 to the mask-unit to merge them. As an example, the first row of the Table 2.1 Pack block could convert eight 8-bit input elements to a single register, composed of sixty-four 1-bit elements. To do that, Bison-e only requires eight iterations, more precisely, eight bs.pack instructions. On every iteration, the pack-unit converts the eight elements of src1 into 1-bit format, while the mask-unit concatenates the created data slice into src2. The concatenation offset used by the mask-unit on every iteration depends on the cnt_o value, ranging from 0 to Post-Proc - 1. Therefore, the post-processing stage either acts as a filter to extract the meaningful slice of data from the multiplier output, or it is used to compress data in case of bs.pack instructions. Its behavior depends on the decoded instruction, as well as on the values set in the control register.

As detailed in Section 2.5, the proposed solution introduces a minimal impact on power and area consumption, as the actual computation is performed by the existing processor multiplier. Moreover, we designed the proposed architecture to avoid a latency overhead increase. Indeed, as in the case of standard multiplication operations on the considered target processor, both bs.ip() and bs.lc() instructions feature a latency of three clock cycles. In the first clock cycle, data are read from the RF, processed by the pre-processing stage, and forwarded to the multiplier input registers. The second clock cycle performs the multiplication, while the third one stores the result into the RF, after being properly extracted by the post-processing stage. As Table 2.1 reports, the number of Pre-Proc iterations required by all the Extend configurations is greater than one. As detailed in Section 2.4.1, this implies that consecutive bs.ip() or bs.lc() instructions share the source operands, allowing to pipeline the execution of multiple iterations.
Algorithm 3 Pseudo-code of the IP kernel using Bison-e.

1: procedure INNER PRODUCT
2: \[ bs.set(\text{config.params}) \] $\triangleright$ Configure Control Register
3: \[ for \; do (i = 0; i < v_{\text{dim}}/el_{\text{in.reg}}; i++) \]
4: \[ for \; do (j = 0; j < \text{Pre} - \text{Proc}; j++) \]
5: \[ bs.ip(ip_{tmp}, v_{\text{in0}}[i], v_{\text{in1}}[i]) \] $\triangleright$ Compute Partial Inner Products
6: \[ ip + = ip_{tmp} \] $\triangleright$ Accumulate Final Inner Product

From Table 2.2, we can also note that Bison-e requires a minimal set of simple instructions. As today’s compilers can optimize, through vectorization, computations such as IP and LC, the proposed methodology can be exploited by a compiler, as soon as the target language supports sub-byte data types. Alternatively, as performed in this work and as a current trend in many fields like deep learning, users can define high-level libraries, optimized with the low-level instructions of Table 2.2.

2.4.1 Enhanced Inner Product Computation

The pseudo-code of the IP computation exploiting Bison-e is reported in Algorithm 3. Firstly, the control register is configured according to Table 2.1. Once the parameters have been loaded into the control register, the main loop computes the IP between two vectors having length $v_{\text{dim}}$. Note that the number of iterations required to perform the computation is given by the ratio between $v_{\text{dim}}$ and the number of elements packed in the register (i.e., $el_{\text{in.reg}}$). Indeed, every loop iteration computes the IP of $el_{\text{in.reg}}$ elements belonging to $V_{\text{in0}}$ and $V_{\text{in1}}$, and each $bs.ip()$ instruction processes input-cluster elements. For example, considering the Bison-e configuration for 1-bit input data in Table 2.1, a single iteration of the loop contains seven $bs.ip()$ instructions, each tackling ten elements of $V_{\text{in0}}$ and $V_{\text{in1}}$. The partial IP is then further accumulated into the final result.

As Algorithm 3 shows, the IP computation exploiting Bison-e has several advantages with respect to both the naïve algorithm and the one exploiting binary segmentation on standard CPUs. Firstly, it allows to fully take advantage of the binary segmentation benefits of reducing the memory footprint and the computation arithmetic complexity. Indeed, Bison-e supports compressed data as inputs, that do not require extra manipulation to be extracted into a standard bitwidth before performing the computation. Moreover, as every $bs.ip()$ instruction belonging to the same loop iteration deals with the same input registers, its execution can be pipelined at the hardware level, allowing reducing the overall computation latency. As Figure 2.5 shows, we used the input and output registers of the two-stage multiplier to pipeline the execution of the instructions, allowing to execute up to two instructions concurrently.
2.4.2 Fused Overlap-Add

As detailed in Section 2.3.2, the main bottleneck of performing the LC kernel with binary segmentation relates with the post-processing phase, since the result of every convolution has to be extracted and accumulated into the output vector. To solve this issue, Bison-e exploits binary segmentation to perform fused overlap-add accumulations. The bs.lc.l() and bs.lc.h() custom instructions compute and extract the whole ic\_dim+ic\_dim-1 result of the convolution. As in the bs.ip() case, the two instructions share their inputs. However, bs.lc.l() returns the lower ic\_dim elements of the result, while bs.lc.h() returns the higher ic\_dim-1 elements. Thus, differently from the LC reference implementation of Algorithm 2, the LC result does not require to be further manipulated before the overlap-add phase, as it is possible to perform overlap-add via binary segmentation without extracting the data. Indeed, the configurations of Table 2.1 allow for extra computation in the segmented data format. As an example, considering the first row of Table 2.1, we can notice that the input-cluster bitwidth (i.e., 21 bits) is greater than the cw resulting from Equation (2.7), which is equal to 18 bits for input-clusters of three elements and 8-bit data sizes. We accounted for the remaining three bits to compute overlap-add via binary segmentation. As an example, Figure 2.6 illustrates the fused overlap-add of a LC performed on two 16×12 input vectors having 4-bit data size, and with Bison-e configured to produce input-clusters with four 4-bit elements. With that configuration, after every bs.lc.l() and bs.lc.h() instructions sequence, the post-processing stage outputs seven elements divided into two registers. Specifically, the first register includes four 16-bit elements corresponding to the outcome of the bs.lc.l() instruction, while the second register con-
2.5. EXPERIMENTAL EVALUATION

tains three 16-bit elements created by the \texttt{bs.lc.h()} instruction. As can be seen from Figure 2.6, the overlap-add can be reduced to two additions per iteration, the first adding the \texttt{bs.lc.l()} result into the current \textit{ova} register (\textit{i.e.}, the current output register), and the second adding the \texttt{bs.lc.h()} result into the next \textit{ova} register. The twenty-seven elements are accumulated via \textit{binary segmentation}, and stored in seven \textit{ova} registers. Thus, the overlap-add phase can be computed without pre-extracting the result of every multiplication, and computing only twenty-four additions, instead of the eighty-four needed by the standard overlap-add implementation to create the twenty-seven elements final result.

2.5 Experimental evaluation

This section evaluates \textit{Bison-e} in terms of performance, energy efficiency, and area consumption. We also perform a comparison with an embedded VPU, integrated into the target SoC, to show the efficiency of \textit{Bison-e} when compared to a more conventional high-performance embedded architecture.

2.5.1 Experimental Setup

Performance numbers have been measured using the gem5 simulator [27], configured with a 5-stage, single-issue in-order pipeline, supporting the 64-bit RV64IM RISC-V ISA. The cache hierarchy comprises 4-way 16 KB L1D and L1I caches, having 2-cycle access latency, and a unified 8-way 64 KB L2 cache with 20-cycle access latency. Moreover, the processor is equipped with a VPU processor, exploiting 2 lanes and a maximum vector length of 4096 bits. We use the gem5 VPU proposed in [124], implementing the RISC-V-V v0.7.1 vector extension [131] to run the vectorized implementation of the workloads implemented with vector intrinsics instructions. We extended the RISC-V GNU Compiler Toolchain [130] with the custom instructions of Table 2.2 to support \textit{binary segmentation} and \textit{Bison-e}, integrated into C/C++ implementations of the benchmarks through intrinsic instructions. We used the MacPat simulator [94] to extract energy efficiency metrics.

2.5.2 Workload Description

\textbf{Convolutional Neural Networks} We leverage on \textit{Bison-e} to improve the efficiency of the AlexNet [82] and VGG-16 [138] QNNs, leveraging on dense matrix-matrix multiplications to perform their convolutional and fully-connect layers, representing the most compute-intensive kernels of DNNs, and typically requiring most of the overall execution time. Further details regarding how these layers are mapped into matrix multiplications, which
2.5. EXPERIMENTAL EVALUATION

![Bar chart comparison of execution time](image)

Figure 2.7: Execution time comparison of IP (a) and LC (b) kernels relying on a naïve implementation or binary segmentation. The three binary segmentation implementations either rely on software exploiting the standard RISC-V ISA (RV64IM), leverage on bit-manipulation instructions (bit-manip), or exploit Bison-e.

other techniques can be used for their computations, as well as an introduction to DNNs quantization techniques are provided in Section 3.2.

For these benchmarks, we exploit the IP kernel computed on Bison-e, improving its execution by reducing the overall number of required multiplications and additions. To test the performance scalability of Bison-e, we explored different data sizes of both data and weights of the selected networks.

**Approximate String Matching**  Research in pattern matching applies to many important use cases, ranging from biological sequence alignments and genome pre-alignment filters [15, 16, 25], to web search engines and data compression. One of the principal fields of pattern matching, called string matching, verifies if a sequence of characters belonging to a given alphabet (i.e., the pattern) matches into a reference string (i.e., the text). The string matching problem can be solved by following many different methods [14]. One widespread algorithm breaks up both the text and the pattern into boolean vectors, one for every letter of the alphabet, and computes boolean LCs among each vector pair. Each LC result is then accumulated into the output vector, whose elements identify the number of mismatches among the pattern and the text, starting from each text position. This solution, firstly proposed in [52], is an extension of the knuth-morris-pratt (KMP) algorithm [103], and represents a valid candidate to solve the problem of approximate string matching with
2.5. EXPERIMENTAL EVALUATION

Figure 2.8: Speed-ups (a, c), and energy efficiency (b, d) of the AlexNet and the VGG-16 CNNs with respect to the scalar implementation, exploiting either Bison-e or the VPU.

don’t care conditions. Although this solution has proved its efficiency in several works in the literature [17, 54], its performance can be further improved by applying data compression, as well as by decreasing the arithmetic complexity of boolean LCs. Both of these optimizations can be efficiently achieved by the enhanced LC computation enabled by Bison-e.

2.5.3 Performance

As a first performance analysis, we study the IP and the LC kernels exploiting binary segmentation, and we compare their run-time with their naïve implementations of Equation (2.2) and Equation (2.6). In Figure 2.7, we benchmark three implementations using binary segmentation: one exploiting the standard 64-bit integer RISC-V ISA (i.e., RV64IM), one featuring bit-manipulation instructions, and one computing with Bison-e.

For the IP, the speed-ups obtained with respect to the naïve kernel are reported in Figure 2.7a, and range from $1.3 \times$ to $1.5 \times$, and from $2.5 \times$ to $3.4 \times$, for the binary segmentation
implementation featuring standard and bit-manipulation instructions, respectively. These results are in line with our analysis in Figure 2.3a, as the instructions needed to pack and extract data typically exploit a smaller latency than the multiplication one, and because of the smaller number of loop iterations required by the binary segmentation implementations. Figure 2.7a also shows that, in contrast to Figure 2.3a, the achieved speed-up does not scale with the number of elements composing the input-cluster. As analyzed in Figure 2.4a, this behavior is due to the input-cluster creation complexity, which grows with the number of elements composing the cluster. While the speed-ups obtained in Figure 2.7a highly differ from the theoretical performance gain that binary segmentation could exploit, the implementation exploiting Bison-e leads to a $4.4 \times$, $7.1 \times$, $14.1 \times$ and $22.6 \times$ speed-up with respect to the reference implementation of Figure 2.7a, for data sizes of 8-bit, 4-bit, 2-bit, and 1-bit, accordingly. Note that the performance of Bison-e are comparable with the maximum theoretical improvement allowed by binary segmentation for the IP kernel, reported in Figure 2.3a.

The experimental evaluation of the LC kernel is reported in Figure 2.7b. The RV64IM-based implementation improves the reference by a factor ranging from $1.1 \times$ to $3.5 \times$, while the implementation exploiting bit-manipulation instructions reaches from $1.6 \times$ to $3.8 \times$ with respect to the reference. When compared to Figure 2.7b, our implementation leveraging Bison-e features a speed-up of $3.5 \times$, $17.4 \times$, $42 \times$ and $61 \times$ for input bitwidth of 8-, 4-, 2- and 1-bit, respectively, proving that the proposed architecture can significantly improve the naïve binary segmentation computation reported in Figure 2.7b.

We implement three workloads belonging to two different application classes, namely deep learning and approximate string matching. For the deep learning benchmark analysis, we focus our evaluation on the Convolutional and Fully-Connected layers of both the AlexNet and the VGG-16 CNNs, for input and weights data sizes ranging from 8-bit to 2-bit. All the kernels (i.e., scalar, vector, and featuring Bison-e) have been implemented using img2col [35], reshaping the convolutional layers as blocked matrix-matrix multiplications, and the fully-connected layers as matrix-vector multiplications. The obtained results, in terms of performance and energy efficiency, are summarized in Figure 2.8. On the performance side, Figure 2.8a reports the speed-ups of the vectorized and the Bison-e implementations of the AlexNet CNN, with respect to the scalar reference. As Figure 2.8a shows, the Bison-e implementation performance scales with the decrease of the input data size. Specifically, the proposed solution runs up to $5.4 \times$, $10.1 \times$, and $20.5 \times$ faster than the scalar implementation for Convolutional layers, and up to $3.7 \times$, $7.2 \times$, and $14.2 \times$ for the Fully-Connected layers, for 8-, 4- and 2-bit data sizes, respectively. Averagely, when compared with the VPU implementation, Bison-e exhibits comparable performance on the
2.5. EXPERIMENTAL EVALUATION

Figure 2.9: Approximate string matching kernel speed-up exploiting either Bison-e (dark-green bars) or the VPU (light-green bars) with respect to the scalar implementation, featuring a 4 (a) and a 256 (b) letters alphabet.

4-bit test, and outperforms it on the 2-bit test by a factor of $1.9 \times$. Bison-e shows a $1.8 \times$ higher run-time than the VPU only for the 8-bit test. However, as Figure 2.8b illustrates, the 8-bit AlexNet performed with Bison-e shows comparable energy efficiency with respect to the vectorized counterparts, and exhibits better efficiency in the 4-bit and 2-bit layers, by a factor that ranges from $1.5 \times$ to $3.1 \times$.

Similarly, Figure 2.8c shows the performance improvement of Bison-e and the VPU of the VGG-16 network, with respect to the scalar baseline. Specifically, a back-to-back execution of the network performed with Bison-e peaks a $4.7 \times$, $9.1 \times$, and $18.5 \times$ with respect to the scalar implementation, for 8-, 4-, and 2-bit computations, also showing comparable and better performance than the VPU, by a factor up to $1.9 \times$ for 2-bit data sizes. In terms of energy efficiency, as detailed in Figure 2.8d, the VGG-16 network is computed with Bison-e gains averagely $1.1 \times$, $1.8 \times$, and $3.6 \times$ with respect to the VPU implementation, for 8-, 4-, and 2-bit computations. For both the AlexNet and the VGG-16 networks, the performance scalability of Bison-e is guaranteed by the increasing number of operations performed concurrently, as well as by the compressed input format on both data and weights, which allows decreasing the overall memory transfers.

Concerning the approximate string matching workload, we consider a pattern of 256 characters, a text whose length ranges from 4K to 128K characters, and an alphabet of
Figure 2.10: Layout of the DRAC SoC including Bison-e, highlighted in green and circled in black, for the 65nm technology. This design, also including the VPU, a PLL, an ADC and several peripherals controllers, is ready for fabrication.

4 (e.g., A, T, G, C) and 256 letters. Figure 2.9 reports the speed-ups of Bison-e and the VPU with respect to the scalar reference. Bison-e outperforms both the scalar and the VPU implementations for all the considered datasets. Concerning the 4-letters alphabet benchmark, Bison-e gains from $21.8 \times$ to $51.6 \times$, and from $1.3 \times$ to $3.2 \times$ execution time with respect to the scalar and the VPU implementations. We obtain similar results for the 256-letters alphabet benchmark, where Bison-e outperforms both the scalar and the VPU execution time by a factor ranging from $25.9 \times$ to $60.6 \times$, and from $1.4 \times$ to $3 \times$, respectively. Furthermore, for the approximate string matching benchmarks of Figure 2.9, Bison-e gains an average energy improvement of $40 \times$ and $5 \times$ when compared to the scalar and the VPU implementations.

2.5.4 Area and Power Analysis

We integrated Bison-e into the DRAC SoC design [12], using the Cadence tool flow (Genus/Innovus), to obtain the layout and main performance metrics of the overall microarchitecture. We implemented the design in two different technologies, namely TSMC 65nm bulk CMOS and GlobalFoundries 22 nm FDSOI. The 65nm layout, reported in Figure 2.10, is ready for production, and includes the SoC along with the VPU, peripheral controllers, a PLL, an ADC, and the IO pad-ring. The design employs standard- and low-threshold
cells, and the synthesis and PnR target frequency is set to 600 MHz. The 22nm layout includes the SoC along with the VPU and peripheral controllers, as an IP block ready to be connected to a PLL and an IO pad-ring. The design employs 8-track standard cells without using body-biasing, with a target frequency of 1GHz. For both implementations, we analyzed the physical impact of \textit{Bison-e} incorporation on area, timing and power figures, referring to the above-defined timing constraints. Regarding the timing performance, target constraints are met in the typical corner, and it has been demonstrated that \textit{Bison-e} does not introduce new critical paths in the processor datapath. Regarding area and power consumption, results are summarized in Table 2.3. In both technologies, the area overhead of \textit{Bison-e} in the whole layout is below 0.07%, and the contribution to the total power consumption is lower than 0.04%. The cell count of \textit{Bison-e} is 1210 in the 65 nm library, while it is 1081 in the 22 nm library. For comparison with \textit{Bison-e}, we implemented and synthesized a narrow-SIMD unit in 22nm, capable of computing 8-,4-,2-,1-bit data on a 64-bit datapath, and our evaluation reported a 10$\times$ area increase with respect to \textit{Bison-e}, whose key novelty relies on reutilizing hardware, featuring low area-overhead and high flexibility.

### 2.6 Related work

Although, to the best of our knowledge, this is the first work investigating the application of binary segmentation on processor cores, several works have analyzed the reduction of arithmetic complexity by packing multiple computations in a single arithmetic operation. The authors in [55] exploit the Xilinx FPGA DSP48E2 slices to pack two 8-bit multiplications, both sharing one of the multiplicands, into a single DSP slice, achieving a 1.75$\times$ speed-up compared to a naïve multiplication on the same device. The same approach has been improved in [29], where the authors propose an enhanced DSP slice architecture able to compute four 9-bit concurrent multiplication with 0.6% area overhead.

Among the works investigating the optimization of narrow integer computing on edge processors targeting CNNs, [41] proposes a ternary weight quantized DNN GEMM library, that replaces multiply-add operations with SIMD bitwise operations to compute the...
GEMM kernel, showing a performance improvement over GEMMLowp of up to $2.9 \times$ when computing the MobileNet-V2 CNN. Compared to Bison-e, [41] shows less flexibility in terms of data size combinations, as they are limited to 3-bit for the weights, and from 6- to 3-bit for the activations. Moreover, our solution does not include instructions and FUs exploiting SIMD to further increase its performance, and still reaches better speed-ups than [41] over GEMMLowp. Other approaches deploying quantized DNNs on edge processors [147, 148] make use of bit-serial implementations of GEMMs computations to exploit standard bitwise operations like the `and` and the `popcount` instructions. Specifically, the authors in [148] propose an algorithm to express the matrix-multiplication kernel as a set of a weighted sum of binary matrix multiplications, thus allowing supporting computations based on variable precisions. Although [148] demonstrates promising performance for 1-bit based computations, it shows no performance improvement with respect their baseline for matrices featuring depths (i.e., the k-dimension) smaller than 256 elements, or having more than a few hundred of elements per dimension. For example, [147] presents a GEMM implementation based on bit-serial matrix multiplications, built on top of the BLAS-like library instantiation software (BLIS) library and targeting DNNs inference on edge devices. Although [147] achieves good speed-ups with respect to their baseline on both the Arm Cortex-A53 and the Arm Cortex-A7 devices, they only support computations based on data sizes lower than 4-bit for both activations and weights.

Other works, such as the one proposed in [57, 58, 117], explore custom SIMD multiply-accumulates (MACs) units and bit manipulation instructions to improve the computation of QNNs on embedded processors. We propose a direct comparison between these works and Bison-e in Table 3.3.

## 2.7 Discussion

Bison-e extends a general-purpose ISA leveraging on instructions that accelerate narrow integer computations exploiting binary segmentation. The insight behind Bison-e is to fill the sweet spot between application-specific accelerators and SIMD units for die-area sensitive edge computing use-cases. On the one hand, Bison-e has proven to be comparably more efficient than a high-performance VPU for narrow integer computations, while featuring $600 \times$ less area overhead. On the other hand, the methodology proposed in this work through Bison-e features more flexibility than high-performance application-specific accelerators like [39, 151], as it can be extended to any linear algebra kernel exploiting SIMD-style narrow computations. For example, [39], represents a state-of-the-art DNN accelerator for mobile devices, featuring 192 processing elements and line buffers for a
2.8. SUMMARY

This work proposes a novel methodology to accelerate linear algebra kernels based on narrow integers. The proposed solution, built upon the binary segmentation mathematical technique, reduces both the memory footprint and the arithmetic complexity of integer linear algebra computations, exploiting an efficiency that is proportional to the ratio between the architecture bitwidth and the application data sizes. We perform a detailed DSE of binary segmentation on 64-bit architecture, highlighting its strengths and pitfalls. We then propose Bison-e, a lightweight and high-performance accelerator targeting narrow integer linear algebra computing on resource-constrained processors, to overcome the main limitations of the analyzed technique on standard CPU architectures and ISAs. Bison-e runs important linear algebra kernels such as IP and LC from $3.5 \times$ to $61 \times$ faster than a scalar RISC-V edge processor. We integrate the proposed architecture into a complete SoC, based on RISC-V, past the PnR step. Our analysis shows that Bison-e considerably enhances the performance of narrow integer computations, introducing a negligible 0.07% impact on the
overall processor area. Specifically, our experimental evaluation shows that \textit{Bison-e} outperforms the scalar processor from $4.7 \times$ to $19.3 \times$ on the AlexNet and the VGG-16 CNN benchmarks in terms of execution time, and shows comparable or higher energy efficiency than a VPU on the same tasks. Moreover, \textit{Bison-e} on approximate string matching tasks reaches execution speed-up from $1.4 \times$ to $3 \times$ when compared to the VPU implementation, showing an average $5 \times$ improvement in terms of energy efficiency.
Chapter 3

Mix-GEMM: An efficient HW-SW Architecture for Mixed-Precision Quantized Deep Neural Networks Inference on Edge Devices

DNN inference based on quantized narrow-precision integer data represents a promising research direction toward efficient deep learning computations on edge and mobile devices. On one side, recent progress of quantization-aware training (QAT) frameworks aimed at improving the accuracy of quantized DNNs allows achieving results close to floating-point 32 (FP32), providing high flexibility on the data sizes selection. However, as previously discussed in Chapter 1 and Chapter 2, current CPU architectures and ISAs targeting edge devices present limitations in the range of data sizes supported to compute DNN kernels.

This chapter presents Mix-GEMM, a hardware-software co-designed architecture (detailed in Section 3.3) capable of efficiently computing quantized DNN convolutional kernels. The Mix-GEMM microarchitecture is built upon Bison-e, presented in Chapter 2, and optimized as an application-specific accelerator performing GEMM, representing the core kernel of DNNs, supporting all data size combinations from 8- to 2-bit, including mixed-precision computations, and featuring performance that scale with the decreasing of the computational data sizes. Our experimental evaluation (Section 3.4), performed on representative quantized CNNs, shows that a RISC-V based edge SoC integrating Mix-GEMM achieves higher performance than SoA frameworks running on commercial Arm and RISC-V based edge processors, while only accounting for 1% of the SoC area consumption. The microarchitecture of Mix-GEMM has been taped out in the context of the DRAC project.
3.1 Introduction

DNNs are currently the preferred choice for artificial intelligence and computer vision tasks in both research and industrial applications. DNNs are composed of a stack of layers, whose execution time is typically dominated by the computation of large linear algebra operations like GEMMs. Optimizing DNNs represents a major challenge in many fields, in particular when targeting the deployment to hardware architectures designed for edge and mobile segments, requiring high performance but presenting tight constraints in terms of area, memory, and energy consumption.

A widespread solution aimed at decreasing this burden is quantization, a family of techniques designed to reduce the numerical precision required to represent the parameters of a DNN and the data computed by its layers. In particular, integer quantization focuses on deploying trained DNNs with narrow integer formats, typically ranging from 8- down to 2-bit [42, 96, 153], rather than with the standard FP32 data size. Quantizing DNNs exposes a large design space, as each layer can be quantized to its own precision. Moreover, input data and parameters can also be quantized differently within a layer, resulting in mixed-precision operations. Exploring this design space allows to trade-off computational requirements against quality of results, which is a key enabler of deployment in resource-constrained devices. Indeed, reducing the precision of highlighted parameters and data decreases the memory and the bandwidth required to store and load them, allowing resource-constrained devices to support larger models, or to relax constraints around the sizing of their memory hierarchy and power envelope. Quantization also enables computing operations like GEMM at low precision, with a consequent improvement in terms of performance and energy efficiency. However, in practice, most of the current general-purpose CPU architectures lack adequate support for efficiently handling narrow-precision formats, as most of the ISAs neither support data sizes smaller than 8-bit, nor support mixed-precision computations. Although modern SIMD extensions [131] and hardware accelerators [59, 117] are increasing their support for narrow-precision data sizes and mixed-precision computations on CPU architectures, they only consider a small subset of data sizes granularities. As a result, exploiting fine-grained quantization of DNNs on modern processors does not always provide a real benefit to the actual computation performance, as quantized data have to be either saved in memory in a sub-optimal format (i.e., with data sizes supported by the processor ISA), or decompressed before the actual computation exploiting costly bit-manipulation operations. Therefore, investigating hardware and software architectures capable of leveraging quantization not only to save memory space, but also to efficiently compute quantized data in terms of performance and energy efficiency, while respecting
the tight area and power caps of resource-constrained devices, represents a critical research challenge for the computer architecture field.

We propose Mix-GEMM, a hardware-software architecture capable of efficiently performing GEMM-based computations targeting narrow integers. The hardware microarchitecture of Mix-GEMM, hosted in the processor execution stage, is built upon the binary segmentation technique, detailed in Section 2.2, which allows computing high-performance SIMD operations on narrow integers, reusing the processor FUs with a negligible impact on area overhead. The key novelty of Mix-GEMM lies in supporting computations based on all the data size combinations between 8- and 2-bit, including mixed-precision, while exploiting high performance, that scales with the decrease of the data sizes involved in the computation. This feature allows tuning the performance of quantized DNNs inference on edge devices with high flexibility, accounting for latency, energy consumption, model accuracy, and memory footprint. Moreover, through a state-of-the-art framework for quantization-aware training [120], we perform an exhaustive design space exploration on the accuracy of relevant DNN networks, exploiting quantization with a fine-grained selection of the data sizes, with the granularity of 1-bit including mixed-precision computations.

The main contributions of this work are listed as follows:

- We design an area- and energy-efficient hardware accelerator, integrated into an edge processor pipeline and capable of computing mixed-precision GEMM kernels based on narrow integers. The proposed architecture, called \( \mu \text{-engine} \), leverages the binary segmentation technique to perform from 3 to 7 MAC per cycle while reusing the processor multiplier;

- We extend the RISC-V ISA with custom instructions used to design a high-performance GEMM software library handling the \( \mu \text{-engine} \), allowing a fine-grained selection of the data sizes and a balance of the overall DNNs performance in terms of throughput, energy efficiency, and memory footprint;

- We integrate our \( \mu \text{-engine} \) into a RISC-V based edge SoC, and we benchmark the performance of Mix-GEMM on six CNNs, namely AlexNet, VGG-16, ResNet-18, MobileNet-V1, RegNet-x-400mf, and EfficientNet-B0. For these networks, Mix-GEMM reaches performance ranging from 4.8 GOPS to 13.6 GOPS, and from 477.5 GOPS/W to 1.3 TOPS/W energy efficiency;

- We investigate the considered quantized CNNs in terms of top-1 accuracy, exploring an exhaustive set of data size combinations exploiting QAT. Our evaluation shows that narrow and mixed-precision quantized CNNs can be Pareto optimal in terms of
computational requirements, and show minimal accuracy losses (i.e., up to 1.5%) for data sizes larger than 4-bit;

- We implement the RISC-V SoC integrating our hardware accelerator, in the Global Foundries 22nm FDX technology node, past the PnR phase, showing that the proposed \( \mu \)-engine only accounts for 1% of the total SoC area, and for an overall 2.3% on its total power consumption;

The remainder of the chapter is laid out as follows. Section 3.2 introduces DNNs and the main techniques at the base of this work. Section 3.3 details \textit{Mix-GEMM}. Section 3.4 presents the experimental evaluation. Section 3.5 compares the main features and performance of \textit{Mix-GEMM} with the most relevant related work. Finally, Section 3.6 discusses the conclusions.

3.2 Background

3.2.1 Deep Neural Networks Computation

DNNs can be defined as a computational graph, where each node represents a layer. The various types of functions computed by each layer can be typically organized in two classes: linear layers, such as convolution or fully-connected, and non-linear operations, such as activation functions like \textit{ReLU}, \textit{softmax}, GELU, or \textit{Tanh}, with DNNs typically alternating between the two from one layer to the next one. DNNs inference becomes a bottleneck when moving to edge computing platforms [64], which are typically provided with a small amount of memory and require low energy consumption.

Convolutions, representing the most computationally and memory intensive kernel of DNNs, can be accelerated in a variety of ways depending on the layer dimensions [8], such as the size of the convolution kernel or the stride. While the direct approach implements it as a series of nested loops, fast algorithms like FFTs [105] or Winograd [85] exploit a numerical transformation of the input and the weights to reduce the overall number of operations. On the other hand, GEMM-based algorithms, such as the \textit{im2row} or the \textit{im2col} approaches [35], maps a convolution to a highly-optimized GEMM implementation.

Direct approaches typically require tuning each kernel with respect to the layer dimensions, either by providing optimized kernels for common choices of dimensions, as in libraries like cuDNN [40], or by generating code just-in-time, as in libraries such as MKL-DNN [61]. Fast algorithms are efficient only for certain dimensions of the layer, and have
3.2. BACKGROUND

additional limitations when applied to quantized values [108]. On the other hand, GEMM-based approaches retain better generality, since they all call into the same pre-compiled backend for any dimensions of the layer, and thus they represent the focus of this work.

In the \textit{im2col} GEMM-based approach, input activations and weights are reshaped and duplicated to fit into the GEMM input matrices, namely $A$ and $B$. Each row of $A$ is composed of the flattened input values that contribute to that pixel, potentially taken from a batch of multiple input images, while each column of $B$ corresponds to flattened parameters computing a single output pixel. A direct implementation of \textit{im2col} incurs a non-trivial overhead in terms of memory and bandwidth, because activations and weights are duplicated across $A$ and $B$. However, as modern \textit{im2col} approaches [49, 107, 146, 165] remove this overhead by implicitly composing $A$ and $B$ in memory, this work only focuses on the compute aspect of GEMM.

3.2.2 Deep Neural Networks Quantization

To further cope with the runtime requirements of CNNs inference, one of the most attractive solutions is quantization [70, 99, 111], a technique that converts CNN data and parameters from floating-point to integer formats, whose size typically ranges from 8-bit down to 1-bit [123] featuring negligible accuracy losses when compared to the floating-point baseline [23, 42, 96, 112, 153].

The acceleration strategy presented in this work applies to uniform affine integer quantization at inference time, which is defined as:

$$y = q(x) = \text{clamp} \left( \text{round} \left( \frac{x}{s} + z \right), y_{\text{min}}, y_{\text{max}} \right)$$ (3.1)

where $x$ is the tensor to quantize, $s$ is the scale, $z$ is the zero-point, while $y_{\text{min}}$ and $y_{\text{max}}$ are defined as:

$$[y_{\text{min}}, y_{\text{max}}] = \begin{cases} [0, 2^{n_b} - 1] & \text{if unsigned} \\ [-2^{n_b-1}, 2^{n_b-1} - 1] & \text{if signed} \end{cases}$$ (3.2)

where $n_b$ is the bitwidth to quantize to. Depending on how $s$, $z$, and $b$ are defined, different variants emerge. The case where $z = 0$ is referred to as symmetric quantization, while $z \neq 0$ is asymmetric quantization. Quantization is named channel-wise if $s$ is a 1-dimensional tensor, while it is layer-wise or tensor-wise in case $s$ is a scalar value.

Quantization is typically adopted in a DNN through either post-training quantization (PTQ) or QAT. PTQ starts from a pre-trained model in floating-point, and relies on a small amount of calibration to determine appropriate values for scales and zero-points. QAT
instead models quantization at training time, allowing to compensate for quantization errors during training. While PTQ requires limited extra computation and data, and is effective at higher precisions like 7- and 8-bit, QAT carries the cost of full training, but can scale down to narrower data sizes. For this reason, QAT represents an excellent candidate to optimize DNN computations on resource-constrained devices, and it is the technique adopted in this work (see Section 3.4.1 for more details).

3.2.3 Efficient Matrix-Matrix Multiplication

The GEMM software library proposed in this work is built upon the double-precision general matrix multiplication (DGEMM) kernel of BLIS [150], a state-of-the-art framework for high-performance BLAS computations [86]. BLIS exploits different compile-time strategies to improve data re-usage (e.g., blocking) and to optimize data movements across the memory hierarchy during the GEMM computation, guaranteeing optimal performance by minimizing the number of cache misses. Specifically, DGEMM computes a block-based multiplication between two 64-bit dense matrices $A$ and $B$, with sizes $m \times k$, and $k \times n$, respectively. The multiplication result is stored in the output matrix $C$, having dimensions $m \times n$. To improve cache efficiency, BLIS partitions the matrices into blocks of smaller dimensions, called panels, stored in contiguous memory arrays. Specifically, a panel of the input matrix $A$ is composed of $mc \times kc$ elements, arranged as $\mu$-panels having size $mr \times kc$. Similarly, each panel of $B$ holds $nc \times kc$ elements, divided into $\mu$-panels holding $nr \times kc$ elements. This specific panels reorganization assures that their elements are accessed with unit stride during the $\mu$-panels computation. Each $C$ $\mu$-panel, having dimension $nr \times mr$, is evaluated in the so-called $\mu$-kernel, computing the matrix-matrix multiplication between single $A$ and $B$ $\mu$-panels. BLIS performance are optimal if its parameters $(mc, nc, kc, mr,$ and $nr)$ are correctly set. Their optimal values can be found analytically [101], and mainly depend on the target processor characteristics, such as the number of cache levels, sizes, and associativities. According to the methodology presented in [101], the $C$ $\mu$-panel is kept in the processor RF, by assigning to $mr$ and $nr$ values whose product does not exceed the number of RF registers. Indeed, each $\mu$-kernel execution updates a different $C$ $\mu$-panel element multiple times, and therefore its partial results must be kept in memories featuring low latencies. Similarly, the $kc$ dimension is set to allow storing the whole $B$ $\mu$-panel in the L1 cache, as its elements are reused for different $\mu$-kernel iterations. Finally, $mc$ is set to ensure that the $A$ panel fits in the L2 cache.
3.3 MIX-GEMM HW-SW Architecture

This section details Mix-GEMM, a hardware-software architecture that allows fast GEMM computations based on narrow integers. On the one hand, as detailed in Section 3.3.1, the proposed software library modifies the BLIS framework to support narrow-precision integers, including all granularities of mixed-precision computations from 8- to 2-bit data sizes, supporting the complete range of bitwidths typically exploited in quantized DNNs. On the other hand, Section 3.3.2 describes the Mix-GEMM hardware microarchitecture, called μ-engine. The μ-engine exploits the binary segmentation technique, introduced in Section 2.2 and investigated in Section 2.3, to perform SIMD MAC operations through the unmodified 64-bit processor scalar multiplier, and to efficiently handle narrow mixed-precision GEMM computations, at a negligible cost in terms of area and power consumption. Specifically, the μ-engine computes the inner-product of two vectors $a$ and $b$ (henceforward called μ-vectors), having bitwidths $bw_a$ and $bw_b$ via binary segmentation, as a set of multiplications among input-clusters. As detailed in Section 2.2, $a$ and $b$ are cleverly packed to compose the input-clusters, whose multiplication results in the μ-vectors inner-product. This packing strategy follows the rule defined in Equation (3.3), which specifies the bitwidth of the elements packed into each input-cluster, called clustering-width (cw). Specifically, given input-cluster size number of elements of each input-cluster, we determine the cw as:

$$cw \geq 1 + bw_a + bw_b + \lceil \log_2(input-cluster\_size + 1) \rceil$$  \hspace{1cm} (3.3)

Note that in Equation (3.3) we exploit a slightly different expression than Equation (2.4) on the cw computation, as Mix-GEMM supports both signed and unsigned computations. However, the conclusions of the DSE proposed in Section 2.3 still apply to this work, as an IP computations of elements ranging from 8- to 2-bit via binary segmentation can still provide SIMD computations ranging from 3 to 7 MAC per cycle.

3.3.1 μ-engine GEMM Software Library

We build our narrow-precision GEMM software library on top of the DGEMM algorithm implemented in the BLIS framework, described in Section 3.2.3. The proposed library leverages BLIS to efficiently move vectors of narrow-precision data through the processor cache hierarchy. Our GEMM library keeps the input matrices $A$ and $B$ compressed over their common $k$ dimension, in chunks ranging from 8 to 32 elements, for 8- and 2-bit data sizes, respectively. Each chunk of compressed elements composes a μ-vector.

As a result, the proposed software library leverages cache-friendly data movements of the BLIS-based DGEMM algorithm, abstracting each μ-vector as a single 64-bit element.
This data organization allows handling non-standard data sizes without extending the processor ISA, increasing the efficiency of quantized DNN computations from different perspectives. First, it enables keeping the DNN activations and weights compressed in main memory (even if their data sizes are not supported by the processor ISA), thus allowing to deploy bigger DNNs on resource-constrained devices. Second, it significantly reduces the number of memory instructions required to perform the GEMM computation, directly impacting performance and energy consumption. As described in Section 3.3.2, each $\mu$-vector pair is forwarded to the $\mu$-engine through a single instruction, which computes the inner-product among them by exploiting the binary segmentation technique.

Figure 3.1 details the dimensions and the memory locations of the matrices used in the $\mu$-engine GEMM software library. In Figure 3.1, we follow the approach proposed in [101] and detailed in Section 3.2.3 to partition panels and $\mu$-panels into a specific level of the
Algorithm 4 Mix-GEMM pseudo-algorithm

1: procedure $\mu$-KERNEL($A_{\mu p}, B_{\mu p}, C$)  
2: for $kca/kua$ iterations do  
3: for $i = 0 \rightarrow nr - 1$ do  
4: for $j = 0 \rightarrow mr - 1$ do  
5: for $k = 0 \rightarrow kua - 1$ do  
6: $A_{\mu \text{vector}} = A_{\mu p}[k + mr \times j]$  
7: $B_{\mu \text{vector}} = k < kub ? B_{\mu p}[i + nr \times k] : 0$  
8: $bs.ip(A_{\mu \text{vector}}, B_{\mu \text{vector}})$  
9: LoadNextAddress($A_{\mu p}$)  
10: LoadNextAddress($B_{\mu p}$)  
11: for $i = 0 \rightarrow nr - 1$ do  
12: for $j = 0 \rightarrow mr - 1$ do  
13: $C_{\mu p}[i, j] = bs.get(j + i \times mr)$  
14: UpdateC($C_{\mu p}, C$)  
15: procedure MACRO-KERNEL($A_p, B_p, C$)  
16: for $nc.nr$ iterations do  
17: $B_{\mu p} = Create\mu\text{Panel}(B_p)$  
18: for $mc.mr$ iterations do  
19: $A_{\mu p} = Create\mu\text{Panel}(A_p)$  
20: $\mu$-KERNEL($A_{\mu p}, B_{\mu p}, C$)  
21: procedure M-GEMM  
22: $bs.set(aX - wY)$  
23: for $n/nc$ iterations do  
24: for $ka/kca$ iterations do  
25: $B_p = CreateBPanel()$  
26: for $m/mc$ iterations do  
27: $A_p = createAPanel()$  
28: MACRO-KERNEL($A_p, B_p, C$)

The processor memory hierarchy. Blocks of elements featuring fewer data reuse are kept in main memory or in the L2 cache, while the ones reused more often are sized to fit either the L1 cache or the processor RF. To further improve data locality, Mix-GEMM defines a further level in the memory hierarchy, called accumulator memory (AccMem) and held inside the $\mu$-engine. The AccMem locally stores an entire $C$ $\mu$-panel having dimension $mr \times nr$ elements, allowing to further increase data locality, and to free the RF registers formerly reserved to the $C$ $\mu$-panel, which are instead allocated to slices of the $A$ and $B$ $\mu$-vectors, avoiding thus to load the same data from cache multiple times. The AccMem is sized to store the whole $C$ $\mu$-panel, holding the result of the matrix-matrix multiplication between the $A$ and $B$ $\mu$-panels.

Algorithm 4 shows the pseudo-code of the proposed BLIS-based library implementation, whose top-function is represented by the M-GEMM procedure. The M-GEMM proce-
 procedure loads the current \( \mu \)-engine configuration through a custom RISC-V instruction called \texttt{bs.set()} (line 22), and then splits the \( \mathbf{A} \) and \( \mathbf{B} \) input matrices in \textit{panels}, holding \( mc \times kca \) and \( nc \times kcb \) \( \mu \)-vectors, respectively (line 25 and 27). Note that we introduce two separate \( k \)-dimensions for the \( \mathbf{A} \) and \( \mathbf{B} \) panels, (\textit{i.e.}, \( kca \) and \( kcb \)), to account for mixed-precision computations, where the input matrices show different \( k \) values. Specifically, in Figure 3.1, \( kca \) is two times \( kcb \), implying that the compressed elements stored in \( \mathbf{A} \) have twice the data size as the compressed elements of \( \mathbf{B} \). While the M-GEMM procedure is used to partition the \( \mathbf{A} \) and \( \mathbf{B} \) matrices into \textit{panels}, the \texttt{MACRO-KERNEL} procedure of Algorithm 4 splits each \textit{panel} into \textit{\( \mu \)-panels} (lines 17 and 19), which are then forwarded to the \texttt{\( \mu \)-KERNEL} procedure (line 20). The \texttt{\( \mu \)-KERNEL} computes the actual matrix-matrix multiplication between an \( \mathbf{A} \) \( \mu \)-panel, composed of \( mr \times kca \) \( \mu \)-vectors, and a \( \mathbf{B} \) \( \mu \)-panel, composed of \( kcb \times nr \) \( \mu \)-vectors, thus creating a \( \mathbf{C} \) \( \mu \)-panel of \( mr \times nr \) elements. Each innermost iteration of the \texttt{\( \mu \)-KERNEL} loads a \( \mu \)-vector pair from the \( \mu \)-panels (lines 6 and 7), and forwards it to the \texttt{\( \mu \)-engine} through a \texttt{bs.ip()} instruction (line 8), that computes its inner-product. Each inner-product is stored in the \texttt{AccMem} and, once the entire \( \mu \)-panels have been issued to the \texttt{\( \mu \)-engine} through \texttt{bs.ip()} instructions (lines 2 to 10), the result is collected from the \texttt{AccMem} using \( mr \times nr \) \texttt{bs.get()} instructions (lines 11 to 13), and accumulated in the output matrix \( \mathbf{C} \) (line 14). The \texttt{bs.set()}, \texttt{bs.ip()}, and \texttt{bs.get()} instructions are implemented as single-cycle instructions, and exploited in the \texttt{\( \mu \)-engine} GEMM library as intrinsics extending the RISC-V ISA.

Note that in the case of mixed-precision computations, each \( \mu \)-vector pair holds a different number of narrow elements. For example, for a mixed-precision configuration where the \( \mathbf{A} \) and \( \mathbf{B} \) input matrices are composed of 8- and 2-bit data (\textit{i.e.}, 8 and 32 elements per \( \mu \)-vector, respectively), a single \( \mathbf{B} \) \( \mu \)-vector requires four \( \mathbf{A} \) \( \mu \)-vectors to issue the same number of narrow-elements to the \( \mu \)-engine. As a result, to balance the number of elements effectively computed by each inner-product, the number of \( \mathbf{A} \) and \( \mathbf{B} \) \( \mu \)-vectors sequentially issued to the \( \mu \)-engine could differ for some data size configurations. Therefore, we extend BLIS with two parameters, namely \( kua \) and \( kub \), aimed at selecting the actual number of subsequent \( \mu \)-vectors on each innermost \( \mu \)-kernel iteration.

Examples of mixed-precision computations requiring different combinations of \( kua \) and \( kub \) are reported in Figure 3.2. Each example considers a different combination of \( \mathbf{A} \) and \( \mathbf{B} \) data sizes (\textit{e.g.}, the data sizes of DNN activations \( aX \) and weights \( wY \)). As the \( a8-w8 \) configuration is composed of \( \mu \)-vectors holding 8-bit for both input matrices, \( kua \) and \( kub \) are equal (\textit{e.g.}, set to 4). As a result, each \( \mu \)-KERNEL innermost execution (lines 5 to 8 in Algorithm 4) issues 4 \( \mu \)-vector pairs (\textit{i.e.}, 32 narrow-elements) to the \textit{Mix-GEMM} \( \mu \)-engine, which computes their inner-product. On the other hand, in both the \( a8-w6 \) and
a6-w4 configurations of Figure 3.2, the number of narrow-elements held in a single A μ-vector is not equal to the one stored in a single B μ-vector. Consequently, \(k_a\) and \(k_b\) are set to guarantee a match in the overall number of narrow elements issued to the \(μ\)-engine. Specifically, the a8-w6 configuration features \(k_a\) and \(k_b\) equal to 4 and 3, while the a6-w4 example sets \(k_a\) and \(k_b\) equal to 3 and 2. When the overall number of μ-vectors elements does not match, the μ-vector having left-over elements is zero-padded (identified in Figure 3.2 with the "-" symbol and colored in white).

### 3.3.2 \(μ\)-engine Hardware Architecture

The \(μ\)-engine architecture, depicted in Figure 3.3, is composed of a computational pipeline, whose stages perform a specific binary segmentation step.

The \(μ\)-engine is fully integrated into the scalar processor execution stage as an additional FU, and its functionalities are completely integrated with the processor pipeline. Note that, to graphically describe the functionality of each \(μ\)-engine component, we adopt the same color scheme for Figure 3.3 and the binary segmentation example of Figure 3.4.

As discussed in Section 3.3.1, we extend the RISC-V ISA with three R-type instructions, named \texttt{bs.set()}, \texttt{bs.ip()}, and \texttt{bs.get()}. The \texttt{bs.set()} instruction is issued to the \(μ\)-engine once for the entire GEMM computation, and it is used to configure its Control Unit. The parameters used to configure the Control Unit either provide details about the incoming \(μ\)-vectors, such as their data sizes and computation type (i.e., signed or unsigned), or specify binary segmentation related constraints, such as the input-cluster size, the \(cw\), the inner-product length, and the slice of data to extract from the multiplication output.
The Control Unit requires a single clock cycle to be reconfigured, and thus introduces a negligible overhead in the computation with respect to the complete GEMM execution. As a result, the data sizes of weights and activations can be easily tuned for each layer of the model, providing a further degree of freedom when exploring the data size configurations, and allowing selecting the best trade-off between performance and accuracy.

Once the Control Unit is properly configured, the µ-engine GEMM library starts issuing multiple bs . ip() instructions to perform the computation. The source operands of each bs . ip() instruction (i.e., a µ-vector pair) are buffered in two separate Source Buffers, and then handled by the data selection unit (DSU), whose main purpose is to select the appropriate number of narrow-elements (i.e., the sub-µvectors) on every clock cycle. Figure 3.2 reports three examples of DSU activity. For each configuration, different colors represent different execution cycles. On each cycle, sub-µvectors starting from the element with ID 0 are selected by the DSU. The maximum number of elements selected per cycle is equal to the input-cluster size of the corresponding configuration. For example, according to Equation (2.8), the a8-w8 and the a8-w6 configurations in Figure 3.2 can perform up to 3 MAC/cycle, while the a6-w4 configuration, featuring a input-cluster size of 4 elements, can perform up to 4 MAC/cycle. When the number of elements left in one of the µ-vectors is less than the input-cluster size, the DSU selects a smaller chunk of elements, and reads a new µ-vector from the corresponding Source Buffer. For example, considering the a6-w4 configuration, while in most of the execution cycles the DSU selects 4 elements from the µ-vectors, there are 3 iterations where it selects 2 elements (i.e., 8-9, 14-15, and 28-29), because in at least one of the current µ-vectors only 2 elements are left to be read.

Data selected by the DSU are then forwarded to the data conversion unit (DCU), which converts them to the appropriate cw, according to Equation (3.3). The main DCU purpose is to create the input-clusters, and to forward them to the 64-bit processor multiplier. The DCU also performs operand sign extensions before the actual multiplication in case of signed computations, or zero-extends each data in case the Control Unit flags an unsigned computation. Note that the DSU and the DCU modules apply the first two binary segmentation steps (respectively colored in green and pink in Figure 3.3 and Figure 3.4).

The processor multiplier computes the input-cluster pair inner-product on each execution cycle, thus performing SIMD computations whose throughput ranges from 3 MAC/cycle to 7 MAC/cycle depending on the selected configuration.

The multiplication output is then filtered by the data filtering unit (DFU) which, according to Equation (2.5), extracts the input-clusters inner-product, which is then accumulated into the AccMem through the internal adder. The Control Unit selects the suitable AccMem address among its mr × nr available slots, depending on the number of execution
cycles required by the loaded configuration. For example, in the $a8-w8$, $a8-w6$, and $a6-w4$ configurations in Figure 3.2, the Control Unit increments the AccMem address after 12, 12, and 9 accumulations, respectively, as these represent the number of execution cycles required to compute their inner-products.

The AccMem facilitates data reuse by updating the $C \mu$-panel elements multiple times during the $\mu$-kernel execution, thus avoiding increased latency, instruction count, and memory traffic. Once the whole matrix-matrix multiplication between the $mr \times kca A \mu$-panel and the $kcb \times nr B \mu$-panel has been computed by the $\mu$-engine, a series of $mr \times nr$ $\text{bs.get()}$ instructions collect the AccMem elements holding the $C \mu$-panel, which are then accumulated into the output matrix $C$.

The processor treats the $\text{bs.set()}$, $\text{bs.ip()}$, and $\text{bs.get()}$ as single-cycle latency instructions. Therefore, the processor does not wait for the $\text{bs.ip()}$ instructions completion before moving forward with the pipeline execution. As a result, while the $\mu$-engine processes the $\mu$-vectors, independent memory and branch instructions can make forward progress by utilizing the address generation and branch resolution FUs. The extra cycles the $\mu$-engine needs to compute the entire $\mu$-vectors are partially compensated by the instructions latencies interleaving $\text{bs.ip()}$ instructions, and in part alleviated by the Source Buffers. This paradigm allows overlapping computational and memory operations, saving
### Figure 3.4: Example of inner-product computation

(i.e., $4 \times 3 + 7 \times 2 + 3 \times 0 + 6 \times 1 = 32$) evaluated via binary segmentation through a pipelined approach. Each color represents a step required by binary segmentation to compute the inner-product. Each tick depicts the pipeline status over time.

A high number of execution cycles from the baseline GEMM algorithm and abstracting the inner-product computation of each $\mu$-vector pair as a single-cycle latency instruction. Considering $\mu$-vectors based on 2-bit data, this implies that the 63 operations needed to compute their inner-product (i.e., 32 multiplications and 31 additions) are replaced by a single `bs.ip()` instruction. As a reference, as described in Algorithm 3, Bison-e would have required 4 `bs.ip()` instructions and 4 `add()` instructions to perform the same computation.

**$\mu$-engine Execution Example:** Figure 3.4 details the steps required by the Mix-GEMM $\mu$-engine to compute via binary segmentation the inner-product of two $\mu$-vectors $a = [4, 7, 3, 6]$ and $b = [3, 2, 0, 1]$ composed of $n = 4$ elements, and having bitwidths $bw_a$ and $bw_b$ equal to 3- and 2-bit, respectively. Supposing, in favor of simplicity, that the example in Figure 3.4 exploits a multiplier having `mulwidth` equal to 16-bit (i.e. instead of 64-bit), Equation (3.3) and Equation (2.8) allow evaluating a $cw$ equal to 8-bit, and a `input-cluster size` of 2 elements per `input-cluster`. As the number of elements of each $\mu$-vector (i.e., $n$) is twice as `input-cluster size`, the complete inner-product computation requires applying
**3.3. MIX-GEMM HW-SW ARCHITECTURE**

*binary segmentation* on two separate $a$ and $b$ slices. To ensure continuity with the Mix-GEMM hardware architecture depicted in Figure 3.3, the example in Figure 3.4 express the inner-product of $a$ and $b$ via *binary segmentation* as a computational pipeline, whose stages follow the same color scheme of Figure 3.3 over time (*i.e.*, *Ticks*). In the first computational step of Figure 3.4 (highlighted in green), $a$ and $b$ are partitioned into *sub-µvectors*, having a number of elements equal to the *input-cluster size*, and the elements order of each $b$ *sub-µvector* (*i.e.*, $b_r$, and $b_r^\prime$) is reverted, according to Equation (2.3). A second step (pink) converts each *sub-µvector* element to a $cw$-bit element, and packs it in the respective *input-cluster*. As Figure 3.4 shows, each *input-cluster* can be seen as a single wide integer (*i.e.*, 1031, 515, 774, and 256) having bitwidth equal to the *mul width* (*i.e.*, 16 bit). The third step (blue) performs the *input-clusters* multiplication. A fourth step (orange) filters a slice of the multiplication output, holding the *input-clusters* inner-product, according to Equation (2.5), thus extracting the partial inner-products (*i.e.*, 26 and 6). Finally (grey), the partial results are accumulated to obtain the inner-product (*i.e.*, 32).

As Figure 3.4 shows, an inner-product of 4 elements is performed via *binary segmentation* with only 2 16-bit multiplications and a single addition, with a consequent $2.33 \times$ arithmetic complexity reduction compared to a standard implementation. As detailed in Section 2.3.1, applying *binary segmentation* to a 64-bit architecture implies an arithmetic decrease of $5 \times$ and $13 \times$ for 8- and 2-bit data sizes, allowing thus computing inner-products with performance ranging from 3 MAC/cycle to 7 MAC/cycle exploiting a single 64-bit multiplier. Therefore, *binary segmentation* allows computing the inner-product of narrow elements exploiting single 64-bit multiplications, with performance ranging from 3 to 7 MAC per cycle. Using the *Bison-e* microarchitecture as a base pillar of the µ-engine allows to dynamically select the number of elements computed per cycle (*i.e.*, the *input-cluster size*) depending on the computation data sizes. This feature allows higher flexibility than traditional SIMD FUs on the supported data sizes combinations, as with the proposed methodology mixed-precision data are anyhow converted to a common data size (*i.e.*, the $cw$) and computed in clusters (*i.e.*, from 3 to 7 elements per cycle). Moreover, the proposed solution does not require specialized FUs, as it reuses the processor multiplier with a consequent area saving.

### 3.3.3 Design Space Exploration

As detailed in Section 3.3.1 and Section 3.3.2, Mix-GEMM defines several parameters, that need to be fine-tuned to guarantee minimal overheads during the GEMM computation. Therefore, we conduct a DSE to select the Mix-GEMM parameters allowing the best trade-off between area and performance. The optimal value of the parameters obtained during
the proposed DSE, considering both the Mix-GEMM software library and the µ-engine, are reported in Table 3.1.

We performed an experimental evaluation to find the best panels and µ-panels dimensions, according to the main SoC characteristics, sweeping through all their combinations for different matrices sizes, finding the optimal mc, nc, and kc values equal to 256, while the optimal mr and nr equal to 4. Accordingly, we set the µ-engine AccMem dimension to 16 elements, as it holds the entire C µ-panel composed of mr × nr elements.

We then analyze the memory overhead introduced by the µ-vectors zero-padded elements with respect to the maximum theoretical memory compression improvements (i.e., from 1 × to 4 × for 8- and 2-bit data). Our analysis shows that the memory overhead introduced by the padded elements with kua and kub equal to 4 is 2.4% on average, considering all the supported configurations. As 4 is the maximum ratio among the data sizes supported by Mix-GEMM (i.e. 8- and 2-bit), it represents the lower bound for kua and kub. Further increasing kua and kub would benefit the memory footprint, as it would increase the number of solutions requiring fewer zero-padded elements on each µ-vectors set. However, increasing kua and kub would be sub-optimal from a performance perspective. Indeed, according to Figure 3.1, the GEMM kernel needs to store kua × mr A µ-vectors and kub × nr B µ-vectors in the processor RF to minimize the number of load operations during the µ-kernel execution. As the RF leveraged by the target processor holds 32 registers, and since the optimal value for both mr and nr is equal to 4, setting kua and kub equal to 4 elements leads to an optimal RF utilization.

Another key parameter we explore is the Source Buffers depth, as small Source Buffers can fill too quickly, stalling the processor pipeline and preventing it from moving forward with the execution of subsequent instructions, while too deep Source Buffers could increase the µ-engine latency, forcing the processor to stall the bs.get() completion until the whole C µ-panel has been completely computed. We equip the µ-engine with a performance monitoring unit (PMU) to collect its metrics during execution, and we benchmark GEMM tasks considering all the supported data sizes configurations, exploring Source Buffers depths of 8, 16, and 32 µ-vectors. Our analysis shows that the number of cycles where the processor is stalled because of full Source Buffers accounts for the 17.8%, 14.3%, and 11.2% for Source Buffers having depths of 8, 16, and 32 µ-vectors. The PMU also registered stalls

---

Table 3.1: Mix-GEMM optimal parameters obtained in the DSE.

<table>
<thead>
<tr>
<th>Number of Elements</th>
<th>µ-kernel</th>
<th>µ-engine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mc 256</td>
<td>nc 256</td>
</tr>
</tbody>
</table>

---
due to `bs.get()` instructions only for Source Buffers of 32 µ-vectors, accounting for 2.3% of the total execution time, closing the overhead gap between Source Buffers holding 16 and 32 µ-vectors. Moreover, post-synthesis results show an area increase of the µ-engine of 67.6% when passing from 16 and 32 elements. For these reasons, we set the Source Buffers depth equal to 16 µ-vectors.

3.4 Experimental Evaluation

This section presents the experimental evaluation of Mix-GEMM in terms of throughput, energy efficiency, and area. We also perform an in-depth evaluation of representative quantized image classification CNNs, namely AlexNet, VGG-16, ResNet-18 [66], Mobilenet-V1 [68], RegNet-x-400mf [159], and EfficientNet-B0 [144]. Our analysis focuses on CNNs since computer vision is a major driving task for artificial intelligence at the edge, which is the focus of this work. However, Mix-GEMM can be applied to all the DNNs quantized with any uniform affine quantization technique, and as such, any advancement in that area can be potentially leveraged by Mix-GEMM. For example, recent works [51,136,143] have demonstrated competitive quality of results for low mixed-precision quantization of BERT for NLP, whose compute expensive kernels based on matrix-matrix multiplications could be accelerated exploiting Mix-GEMM.

The proposed experimental evaluation aims to find the best trade-offs in terms of accuracy and throughput, showing the potential of combining quantization and efficient narrow-precision inference acceleration. Indeed, the main novelty of Mix-GEMM is its ability to support all combinations of precisions between 8- and 2- bit, while guaranteeing performance increasing with the decrease of activations and weights bitwidths. This feature enables a new degree of freedom in deploying DNNs on edge devices. Indeed, the large number of configurations supported by Mix-GEMM widen the design space used to trade-off performance, memory, energy, and accuracy, which is of fundamental importance when targeting resource-constrained devices.

3.4.1 Experimental Setup

To benchmark Mix-GEMM in terms of performance and energy efficiency, we integrate its hardware µ-engine on an edge RISC-V SoC [140]. The target edge processor, implementing the RV64G instruction set, features a single-core, 7-stage, in-order, single-issue pipeline, while the memory hierarchy features L1 and L2 data caches having sizes of 32KB, and 512KB, respectively. The Mix-GEMM performance results have been compiled with
3.4. EXPERIMENTAL EVALUATION

Figure 3.5: Speed-up of Mix-GEMM over the baseline BLIS-based DGEMM algorithm on square input matrices. Configurations sharing the activations data size \(a\) are represented with the same color, with different line patterns differentiating the weights data size \(w\).

For QAT, we adopt PyTorch 1.8 [121], a popular deep learning framework, and Brevitas 0.7.1 [120], a neural networks quantization library. All experiments are retrained with QAT from post-training quantization of FP32 models [3, 104], which we also consider as the accuracy baseline of the target CNNs. We train on the ImageNet training dataset [43] with four NVIDIA V100 GPUs, reporting the best top-1 validation accuracy obtained for each configuration. We experiment with multiple separate precisions for activations and weights, except for the first and last layers, which are kept at 8-bit to preserve accuracy. Weights are quantized \textit{per-channel} with \textit{scale} computed from the \textit{absmax} of the weight tensor [11], while activations are quantized \textit{per-tensor} with \textit{scale} learned in log domain [73]. Quantization \textit{scales} and biases are left in floating-point. To simplify training, both activation and weights are trained with \textit{zero-point} equal to zero. ResNet-18, AlexNet, MobileNet-V1, VGG-16, RegNet-x-400mf, and EfficientNet-B0 retrain with Stochastic Gradient Descent (SGD) featuring momentum of 0.9, weight decay \(1e^{-4}\), and initial learning rate of \(1e^{-3}, 1e^{-4}, 1e^{-2}, 1e^{-3} 4e^{-2}\), and \(3.2e^{-3}\). We respectively employ 90, 90, 120, 45, 150, and 90 epochs, lowering the learning rate by \(0.1\) every 30, 30, 15, 30, and 30 epochs, with a
batch size of 256, 128, 128, 32, 128, and 64 per GPU. An exception is made for combinations of 8- and 7-bit, where models are fine-tuned for 5 epochs at the lowest learning rate they would reach in the normal training schedule, and for the EfficientNet-B0 configurations showing data sizes lower than 4-bits, that are trained employing 270 epochs. The initial activation post-training quantization is performed by averaging the 99.999 percentile of the activation absolute values for 8 batches [156], and then performing bias correction [113] for 8 more batches (except for VGG-16, where bias correction would lead to overflow). To improve convergence at low precision without overhauling the whole approach to quantization, AlexNet, ResNet-18, MobileNet-V1, RegNet-x-400mf, and EfficientNet-B0 a4-w3 and a3-w3 are retrained from a4-w4 instead of FP32, with the same training settings as above except for weight decay at 5e−5. Similarly, a3-w2 and a2-w2 are retrained from a3-w3 results. For VGG-16, only a3-w2 and a2-w2 are handled separately, by first replacing relu with relu6 in the pretrained FP32 network, and then retraining with the settings above.

3.4.2 Performance

We first highlight the Mix-GEMM scalability by analyzing its performance on general GEMM tasks, exploiting a dataset composed of square input matrices with 64 to 2048 elements per dimension. Figure 3.5 shows the performance increase of Mix-GEMM with respect to the BLIS-based DGEMM baseline, running on the same RISC-V SoC integrating the proposed µ-engine, for a subset of 12 activations and weights combinations. This first evaluation allows quantifying the performance benefits of the proposed hardware microarchitecture with respect to the BLIS-based DGEMM baseline. As Mix-GEMM keeps narrow-precision elements compressed in 64-bit data (i.e., from 8 to 32 narrow-elements), it allows reducing the problem size from 8× to 32× with respect to the DGEMM implementation of BLIS. However, reducing the computation data sizes is not sufficient to guarantee high benefits in terms of performance. Indeed, BLIS running with 8-bit data only reaches an average 2.5× performance improvement with respect to the DGEMM baseline. On the other hand, as Figure 3.5 shows, the experimental steady-state performance of Mix-GEMM over the DGEMM baseline ranges from 10.2× to 27.2× for the a8-w8 and a2-w2 data size configurations. Different motivations allow Mix-GEMM running at 8-bit to perform 10.2× and 4.1× averagely faster than the baseline running at 64- and 8-bit. First, Mix-GEMM keeps data compressed until the operands are issued to the µ-engine, thus reducing the overall number of operations, while increasing the throughput in terms of elements fetched from memory on each cycle. The µ-engine, computationally sustains this
3.4. EXPERIMENTAL EVALUATION

Figure 3.6: Performance vs. accuracy Pareto frontier for the selected CNNs. Labels represent activations and weights data sizes \((a\text{ and } w)\), respectively. We measure the performance of the quantized network exploiting \textit{Mix-GEMM}, while the FP32 performance is measured exploiting OpenBLAS running on the SiFive U740 processor.

throughput by performing multiple MAC operations per cycle through the processor multiplier. The AccMem allows then to locally accumulate data, avoiding execution overhead due to additional \textit{store} and \textit{add} operations. Finally, the pipelined structure of the \textit{\(\mu\)-engine} provides a further increase in the overall throughput, as it allows to hide the \texttt{bs.op()} operations latency without waiting for their completion. As Figure 3.5 also highlights, these \textit{Mix-GEMM} benefits remain valid for any data type configuration, allowing it to actually scale its performance with the decrease of the computation data types. Specifically, the \textit{a8-w8} performance shows a 21.6\% performance improvement with respect to the theoretical lower bound of \(8\times\), as \textit{Mix-GEMM} exploits its AccMem to reduce the number of operations needed to update the output matrix. On the other hand, \textit{a2-w2} shows a performance penalty of 15\% with respect to the theoretical upper bound, mainly due to the high ratio between \textit{\(\mu\)-vector size} and \textit{input-cluster size} (i.e., 32 elements per \textit{\(\mu\)-vector} and 7 MAC/cycle of \textit{input-cluster size}), which implies a higher number of cycles to process the complete \textit{\(\mu\)-vector} (i.e., 5 cycles). However, this overhead is only noticeable in a few configurations,
and does not prevent the performance scaling of Mix-GEMM. Indeed, the \textit{a4-w4} configuration in Figure 3.5 shows a $16 \times$ speed-up with respect to the baseline, which is in line with the theoretical one.

Aiming at evaluating Mix-GEMM also on SoCs tight by higher area and power constraints, we explore its performance exploiting smaller L1 and L2 caches. Our exploration, performed on all the supported data sizes and considering the same benchmark proposed in Figure 3.5, shows a small performance decrease when reducing the L1 cache from 64KB to 16KB or the L2 cache from 512KB to 64KB, accounting for 5.2\% and 7\% on average, respectively. Decreasing both the L1 and L2 sizes (i.e., 16KB and 64KB) allows reducing the SoC area by 53\%, and still allows Mix-GEMM to achieve high performance, with an average penalty of 11.8\%.

Figure 3.6 reports the most performant combinations of each network top-1 accuracy and the corresponding Mix-GEMM throughput, accounting for the execution time spent on each convolutional layer. We also highlight the Pareto-optimal curve representing the best trade-offs between performance and network accuracy for each network. Configurations that are not on the Pareto frontier are not reported in Figure 3.6, with the exception of the \textit{a8-w8} configuration. The FP32 performance baseline has been measured with the well-known OpenBLAS library [157], exploiting single-threading and running on the SiFive U740 RISC-V processor, featuring a 64-bit dual-issue in-order pipeline running at 1.2 GHz. As Figure 3.6 shows, Mix-GEMM outperforms the FP32 baseline on all the benchmarked CNNs by a factor ranging from 5.8\texttimes{} to 15.1\texttimes{} for AlexNet, from 5.8\texttimes{} to 14.6\texttimes{} for VGG-16, from 5.7\texttimes{} to 13.8\texttimes{} for ResNet-18, from 5.3\texttimes{} to 10.6\texttimes{} for MobileNet-V1, from 5.7\texttimes{} to 11\texttimes{} for RegNet-x-400mf, and from 5.7\texttimes{} to 14.5\texttimes{} for EfficientNet-B0.

Accuracy-wise, Figure 3.6 shows that all the considered networks maintain a top-1 accuracy close to or better than the FP32 baseline for data sizes larger than 4-bit on both activations and weights, showing accuracy losses below 1.5\%. This result demonstrates the benefits of the proposed solution in supporting non-standard data sizes. For example, Mix-GEMM can exploit the \textit{a5-w5} configuration and reach a 60\% performance improvement with respect to the \textit{a8-w8} configuration on the selected networks, while guaranteeing similar accuracy and saving 60\% in memory usage. Figure 3.6 also shows minimal accuracy drops, with respect to the FP32 baseline, on configurations exploiting 4-bit as minimum data size, with losses ranging from 0.01\% for AlexNet, up to 4.2\% on EfficientNet-B0.

For more aggressive quantizations, exploiting 3- and 2-bit data sizes, the considered networks show accuracy losses ranging from 0.5\% to 5.1\% for AlexNet, from 1.2\% to 6.5\% for VGG-16, from 2.2\% to 8.6\% for ResNet-18, from 7.6\% to 34.5\% for MobileNet-V1, from 2.6\% to 13\% for RegNet-x-400mf, and from 10.3\% to 32.8\% for EfficientNet-
B0. Note that, to minimize complexity and support reproducibility, all results have been obtained by applying the same quantization techniques across all networks and data sizes, with limited hyperparameter tuning. We expect lower losses at 3- and 2-bit data sizes applying more tailored low mixed-precision techniques, such as [21,24,135]. Nonetheless, our results show that Mix-GEMM can extend the Pareto frontier to additional data sizes, capable of providing speed-up for a given accuracy target. This feature is particularly useful for edge deployment scenarios, where a trade-off between performance and quality of results typically has to be reached.

3.4.3 Physical Design and Energy Efficiency

To present the physical layout and extract the main physical design metrics regarding area, timing, and power, we implement the RISC-V SoC including the proposed hardware $\mu$-engine in the Global Foundries 22FDX 22nm FDSOI technology. We set the target frequency to 1.2 GHz for both synthesis and PnR, using 8-track standard cells without exploiting body-biasing. The SoC layout, depicted in Figure 3.7, features a total area of 1.96 mm$^2$, and includes the RISC-V in-order core, the $\mu$-engine (highlighted in green and circled in black), the IO pad-ring, and the uncore composed of L2, L1d, and L1i caches of size 512KB, 32KB, and 16KB, respectively. The $\mu$-engine occupies a total area of 0.014 mm$^2$, and adds an overhead of 1% on the total chip area. Table 3.2 highlights the area breakdown of the proposed hardware $\mu$-engine, and reports the area overhead of every $\mu$-
### 3.5. COMPARISON WITH STATE-OF-THE-ART SOLUTIONS

#### Table 3.2: $\mu$-engine Area Breakdown

<table>
<thead>
<tr>
<th>Component</th>
<th>Area [(\mu m^2)]</th>
<th>SoC Overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src Buffers</td>
<td>4934.63</td>
<td>0.36</td>
</tr>
<tr>
<td>DSU</td>
<td>1094.45</td>
<td>0.08</td>
</tr>
<tr>
<td>DCU</td>
<td>2832.46</td>
<td>0.21</td>
</tr>
<tr>
<td>DFU</td>
<td>1842.25</td>
<td>0.13</td>
</tr>
<tr>
<td>Adder</td>
<td>741.58</td>
<td>0.05</td>
</tr>
<tr>
<td>AccMem</td>
<td>1214.35</td>
<td>0.09</td>
</tr>
<tr>
<td>Control Unit</td>
<td>981.43</td>
<td>0.08</td>
</tr>
</tbody>
</table>

| Total: $\mu$-engine | 13641.14 | 1.00 |

$\mu$-engine component on the SoC. The main area contribution is given by the Source Buffers, implemented as 64-bit wide registers holding 16 entries. The other $\mu$-engine components introduce an area overhead in the SoC smaller than 0.3%. Our synthesis and PnR evaluation evidence that the $\mu$-engine does not add critical paths in the design, and introduces a post-layout estimated power consumption overhead of 2.3%. We compute the energy efficiency of Mix-GEMM by performing a post-PnR gate-level simulation of the SoC executing the selected CNNs, and considering the total power consumption of the $\mu$-engine and the processor multiplier. Our evaluation shows that Mix-GEMM achieves from 522.1 GOPS/W to 1.3 TOPS/W for the computation of AlexNet, from 524.3 GOPS/W to 1.3 TOPS/W on VGG-16, from 509 GOPS/W to 1.2 TOPS/W on ResNet-18, from 477.5 GOPS/W to 944.1 GOPS/W on MobileNet-V1, from 503.3 GOPS/W to 982 GOPS/W on RegNet-x-400mf, and from 509.7 GOPS/W to 1.3 TOPS/W for EfficientNet-B0.

#### 3.5 Comparison with State-of-the-Art Solutions

Accelerating quantized DNNs represents a widespread research topic [36,44,65]. Although several representative works targeting GPUs [92] and FPGAs [28,127] are present in the literature, this section mainly considers related research works targeting CPU architectures in the edge domain. We divide the related work into three main categories proposing different approaches to optimize quantized DNNs computations on edge devices, and we compare Mix-GEMM with the most relevant works of each category. We first consider DNN software libraries exploiting existing edge processors to efficiently compute GEMM kernels based on quantized data. We then analyze hardware-software co-designed architectures computing DNNs on edge processors adopting ISA extensions and custom FUs. We finally list the most relevant works proposing decoupled DNN accelerators for edge devices. A detailed comparison with the most relevant related works is then presented in Table 3.3.
3.5. COMPARISON WITH STATE-OF-THE-ART SOLUTIONS

3.5.1 Optimized Software Libraries

Application-specific libraries targeting commercial edge processors, such as Facebook QNNPack [50], Arm CMSIS-NN [83], and Google GEMMLowp [72] are often used to boost the performance of quantized DNNs on edge processors. To compare the performance of Mix-GEMM with respect to these SoA software libraries, we execute the considered CNNs exploiting GEMMLowp, adopted in TensorFlow Lite [10], and highly optimized for computations based on 8-bit quantized data. The GEMMLowp benchmarks have been performed on an Arm Cortex-A53 processor, one of the most widely used architectures targeting the edge, in single-threaded mode. The Arm Cortex-A53 features a 64-bit, 8-stages, dual-issue in-order pipeline running at 1.2GHz and exploiting the NEON SIMD extension. As Table 3.3 shows, the GEMMLowp performance [72] are comparable with Mix-GEMM when computing the same networks considering its $a_8\cdot w_8$ configuration. However, as GEMMLowp does not currently support less than 8-bit based computations as a consequence of the underlying ISAs limitations, Mix-GEMM allows for better performance, while guaranteeing comparable accuracy. For example, from Figure 3.6 it can be noted that the $a_5\cdot w_5$ configuration of Mix-GEMM is capable of providing up to 70% better performance than GEMMLowp, while losing only 0.22% of accuracy on average among the selected networks. A remarkable solution targeting the RISC-V ISA is Dory [31], a framework to deploy DNNs on the GAP-8 processor [53], reaching up to 4.2 GOPS performance to compute the convolutional layers of MobileNet-V1 at 8-bit on eight cores running in parallel. Compared to Dory, our solution achieves up to $2.6\times$ better performance on MobileNet-V1, even running on a single core.

Although these libraries feature high performance on 8-bit computations, they do not support computations targeting sub-byte operands, as compressed data in memory need to be converted to 8-bit to exploit the SIMD operations offered by current commercial ISAs. These limitations are highlighted in CMix-NN [33], proposing an inference library for DNNs optimized for Arm processors and targeting 8-, 4-, and 2-bit mixed-precision computations. CMix-NN [33] demonstrates the benefits of supporting mixed-precision computations based on narrow integers to compute DNNs inference, as they are able to scale their performance up to $2\times$ in energy efficiency and $1.7\times$ in throughput with respect to their 8-bit implementation. However, their MobileNet-V1 implementation latency is dominated by the lack of mixed-precision and sub-byte SIMD instructions at the ISA level. As a result, Mix-GEMM offers roughly one order of magnitude speedup on MobileNet-V1 when compared to CMix-NN.
### 3.5.2 Specialized Arithmetic Units

As off-the-shelf architectures and ISAs are inefficient in deploying quantized DNNs targeting data sizes lower than 8-bit, several works propose specialized FUs and custom ISA extensions to enable efficient narrow mixed-precision GEMM computations on edge devices. In this context, PULP-NN [58] exploits 8-bit SIMD MAC units and inner-product RISC-V based custom instructions to compute up to 16 8-bit MAC operations concurrently. PULP-NN proposes casting instructions to pack and extract vectors composed of lower data sizes (i.e., 4- and 2-bit) while reusing the same SIMD MAC units. Although their experimental evaluation shows performance improvements against their 8-bit baseline, their casting instructions introduce overheads on 4- and 2-bit based computations, hence decreasing their performance improvement for lower bitwidths. Indeed, their performance reaches 0.6 GOPS for 8-bit computations, while it is limited to 0.2 GOPS for 2-bit data. Bruschi et al. [30] extend PULP-NN to support mixed-precision combinations for 8-, 4-, and 2-bit data sizes on an eight-cores RISC-V processor. As in PULP-NN, however, their work suffers from the same overheads on sub-byte data sizes, responsible for a $2.5 \times$ performance degradation when comparing 8- against 2-bit computations, as they also require additional pack and extract instructions. These limitations do not affect Mix-GEMM, which is capable of scaling its performance by 1.9× when targeting the same Convolution benchmark. Ottavi et al. [117] extend a RISC-V core with 4- and 2-bit based MAC units and custom controllers to enable narrow mixed-precision computations based on 8-, 4-, and 2-bit data. Performance-wise, Mix-GEMM is from 2.4× to 3.8× faster than [117], while supporting a greater number of data size combinations. A set of custom RISC-V ISA instructions and custom FUs to boost the performance of GEMM computations on edge devices are also proposed in XpulpNN [57]. Their hardware microarchitecture comprises SIMD units supporting from 4 8-bit to 16 2-bit MAC/cycle, but it is not supporting mixed-precision computations.

Note that the works in [30,57,58,117] only consider a small convolutional kernel fitting the L1 cache as their experimental evaluation, which is not representative of real DNNs. Also, they neither provide performance results considering entire networks, nor explore how their ISA extensions can be integrated into high-performance software libraries such as BLAS, or how larger convolutional kernels introducing misses in the cache hierarchy would affect the performance of their proposal. Moreover, their baseline processor leverages on custom ISA extensions capable of introducing up to 3.1× performance improvement in the GEMM computation with respect to the standard RISC-V ISA [134]. These optimizations (e.g., zero overhead hardware-loops) are orthogonal to Mix-GEMM, and can
### 3.5. COMPARISON WITH STATE-OF-THE-ART SOLUTIONS

Table 3.3: Comparison with state-of-the-art: performance and efficiency ranges ordered according to the supported data sizes (e.g., 8b–2b). Results gathered from published papers.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Architecture</th>
<th>Convolution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Baseline</td>
<td>FP32</td>
<td>✗</td>
</tr>
<tr>
<td>ARMv8</td>
<td>RV64</td>
<td>#</td>
</tr>
<tr>
<td>ARMv7</td>
<td>RV32</td>
<td>†</td>
</tr>
<tr>
<td></td>
<td>RV32</td>
<td>†</td>
</tr>
<tr>
<td></td>
<td>Decoupled</td>
<td>✗</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data sizes</th>
<th>Supported Tech. Area</th>
<th>Energy Tech. Area</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8b–2b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Benchmarks**
- Baseline FP32
- ARMv8 RV64
- ARMv7 RV32
- Decoupled

**Notes**
- Considers an input tensor of shape \((H \times W \times F_{in})\) and a filter of shape \((F_{out} \times K_{dim} \times K_{dim} \times F_{in})\).

- Performance and efficiency ranges ordered according to supported data sizes (e.g., 8b–2b).

- Explores the Neon SIMD extension.

- Energy efficiency refers to the entire SoC.

- Area only includes extension for + and 2-bit MAC FUs.

- Equipped with custom ISA extension exploiting hardware loops, post-increment load and store, and 4 × 8-bit MAC FUs.

- Area includes extension for 4- and 2-bit MAC FUs.

- Energy efficiency refers to the entire SoC.

- Exploits the Neon SIMD extension.

- Equipped with custom ISA extension exploiting hardware loops, post-increment load and store, and 4 × 8-bit MAC FUs.
3.5. COMPARISON WITH STATE-OF-THE-ART SOLUTIONS

be implemented in the processor integrating Mix-GEMM to allow for a further performance improvement on DNN computations.

Other works [46, 76, 116] explore novel SIMD MAC units supporting sub-byte and mixed-precision computations, providing promising results in terms of area and throughput, and representing a valid alternative to be integrated into the Mix-GEMM pipeline in place of the proposed binary segmentation based approach. However, their implementations lack flexibility when compared to Mix-GEMM. For example, [46] can only handle power-of-two data, including mixed-precision. Moreover, Mix-GEMM can provide performance scalability depending on the exploited data size, while reusing the processor multiplier (i.e., without implementing custom MAC units).

Table 3.3 also shows that most related works do not support computations based on mixed-precision data sizes that, as demonstrated in Section 3.4.2, are essential to enable efficient computations on the edge, as they have the potential to extend the Pareto frontier of modern deep learning models.

3.5.3 Decoupled DNN Accelerators

DNN decoupled accelerators represent a well-studied topic \cite{18, 38, 90, 145, 166}. The high performance characterizing these accelerators is, however, counterbalanced by their lack of flexibility, as a large portion of the SoC has to be dedicated to the computation of a single kernel. Moreover, the software stack required by decoupled accelerators is typically more complex than the one proposed in Mix-GEMM, as they require specific offloading mechanisms and coherence management handled at the hardware or software level. In Eyeriss \cite{38}, the authors exploit a bi-dimensional array of 16-bit processing elements and a custom multi-level hierarchical memory, optimized for both dense and sparse computations, exploiting a total area of 12.25 mm\(^2\) in 65 nm CMOS technology. Aiming to address more aggressive quantization, UNPU \cite{90} explores bit-serial MAC units supporting a fixed activations data size and from 16-bit to 1-bit weights data sizes.

Mix-GEMM achieves 0.2\(\times\) and 0.6\(\times\) in performance compared to Eyeriss on the AlexNet and VGG-16 computations, and exploits an energy efficiency comparable to UNPU when exploiting 8-bit data sizes. Moreover, leveraging on DeepScaleTool \cite{132} to scale their area from 65 nm to 22 nm, we observe that Mix-GEMM requires 96.8\(\times\) and 126.5\(\times\) less area than Eyeriss and UNPU, respectively. Consequently, Mix-GEMM computing at 8-bit reaches a core area efficiency improvements (i.e., GOPS/mm\(^2\)) ranging from 6.7\(\times\) to 24\(\times\) with respect to Eyeriss, and from 1.2\(\times\) to 1.4\(\times\) when compared to UNPU, on the computation of AlexNet and VGG-16. As such, we believe that Mix-GEMM represents a valid alternative to decoupled DNN accelerators targeting resource-constrained devices.
3.6 Discussion

3.6.1 Comparison with Bison-e

The Mix-GEMM µ-engine leverages on the binary segmentation technique exploration proposed in Section 2.3.1, for inner-product acceleration of narrow integers, as well as on the Bison-e microarchitecture detailed in Section 2.4 to achieve high-performance and area-efficient computations. As Bison-e represents a more general microarchitecture for accelerating linear algebra operations between narrow integers, when designing Mix-GEMM, we identify different sources of improvements to specialize it for the computation of matrix-matrix multiplications for DNNs. Firstly, compared to Bison-e, Mix-GEMM reduces the number of instructions required for computing the same µ-vectors. This reduction is achieved by incorporating input Source Buffers and the DSU into the µ-engine. Consequently, fewer bs.\(\text{ip}()\) instructions are needed, resulting in improved performance and energy efficiency. Secondly, Mix-GEMM leverages data locality through the AccMem, which enhances data-reuse opportunities in BLIS. Although the AccMem occupies only 0.09% of the total SoC area (as shown in Table 3.2), it significantly reduces the number of store operations required. Thirdly, unlike Bison-e, Mix-GEMM incorporates a specialized software library for dense matrix-matrix multiplications, in addition to the custom instructions extending the RISC-V ISA. Combining these enhancements, Mix-GEMM outperforms Bison-e by factors ranging from 10.5× to 13× on AlexNet and from 5.4× to 8.8× on VGG-16.

3.6.2 Performance scalability

A key strength of Mix-GEMM relies on its scalability. For processors hosting SIMD units, the µ-engine can be properly sized to sustain a higher throughput. Indeed, the Source Buffers can store wider µ-vectors, whose data size matches the SIMD datapath width, thus allowing the bs.\(\text{ip}()\) operations to offload more data to the µ-engine for each instruction. Consequently, the DSU and DCU units can process a broader cluster of elements by distributing them across the arithmetic FUs within the processor. For instance, the µ-engine computation can be divided between the scalar integer multiplier, which performs inner-products exploiting the binary segmentation, and the 8-bit SIMD MAC units.

Likewise, the performance advantages of Mix-GEMM extend to processors with multiple cores. Our BLIS-based library facilitates straightforward implementation of multithreading support [149], while maintaining performance-per-core close to that of the single-threaded implementation [139]. Each processor core can instantiate a µ-engine with minimal impact on area and power consumption.
3.6. DISCUSSION

3.6.3 Mix-GEMM workflow

The workflow diagram of Mix-GEMM is shown in Figure 3.8. The target Pytorch model is trained using a QAT framework such as Brevitas. The sizes of activations and weights data are gradually increased, either on a per-network or per-layer basis, until the desired accuracy is achieved. This training workflow provides a trade-off between model size and accuracy at a high level of granularity. In a per-network strategy, all the model layers share the same data sizes for activations and weights. On the other hand, a per-layer strategy allows selecting a different set of activations and weights data sizes for each layer of the network. A per-layer strategy increases thus the design space to explore, but allows for a more fine-grained tuning of accuracy and model size. Applying a per-layer strategy to the analyzed CNNs in Figure 3.6, which originally used a per-network quantization strategy, would further enhance the performance of Mix-GEMM. Indeed, the flexibility of Mix-GEMM enables easy reconfiguration of different combinations of data sizes for each layer using the \texttt{bs.set()} instruction, which incurs a single clock cycle of latency.

After selecting the optimal combination of activations and weights data sizes, the quantized model can be converted to an ONNX model and deployed using ONNX Runtime, an engine developed by Microsoft that provides high-performance inference for ONNX models. As ONNX Runtime is designed to optimize and execute models efficiently on various hardware platforms, including CPUs, GPUs, and specialized accelerators, and provides a unified runtime environment that abstracts the underlying hardware and delivers optimized performance across different devices, Mix-GEMM can be seamlessly integrated as an additional backend, facilitating accelerated DNN inference.

Figure 3.8: Mix-GEMM workflow diagram considering training and inference.
3.7 Summary

While quantization has been proposed as a solution to reduce the memory and computation requirements of DNNs, current CPU architectures lack adequate support for efficiently handling narrow-precision formats. To address this issue, this chapter presented *Mix-GEMM*, a hardware-software co-designed architecture capable of accelerating quantized DNNs inference on resource-constrained devices, such as mobile, IoT, and embedded systems.

*Mix-GEMM* combines the benefits offered by the BLIS framework and binary segmentation to perform high-performance GEMMs, supporting all granularities of mixed-precision computations between narrow integers, at the cost of a small area and power overhead. Moreover, *Mix-GEMM* is capable of scaling the performance and the memory requirements of narrow-precision GEMM computations depending on the target data sizes, showing comparable or better performance than state-of-the-art GEMM libraries running on commercial processors. Our experimental evaluation shows that *Mix-GEMM* reaches from 4.8 GOPS to 13.6 GOPS on the computation of relevant CNN workloads, and up to 1.3 TOPS/W energy efficiency, while accounting for a negligible 1% of the total processor area. We believe our solution represents a step forward to fill the gap between the needs of quantized DNNs, requiring high performance and flexibility in the data sizes involved in the computation, and edge-based architectures, demanding tight area and energy constraints.
In recent years, there has been a surge of interest from both industry and academia in developing novel techniques to enhance the computational efficiency of DNNs. Specifically, as detailed in Chapter 2 and Chapter 3, the main effort has been made to tackle the computational demands of convolutions, which are the most computationally intensive layers in DNNs. While hardware accelerators designed for edge environments, such as the one proposed in Chapter 3, are constrained by limited area and power consumption, DNN computing systems targeting HPC and cloud environments are allocating an increasing chip area to this kernel. For instance, the Google tensor processing unit (TPU) [75] doubles its matrix-multiply unit (MXU) size with each new generation [74], and the latest Huawei DaVinci architecture [98] incorporates a specialized matrix multiplication unit capable of computing up to 4096 FLOPS/cycle. However, while current HPC and cloud accelerators are optimized for convolutions and matrix-matrix multiplication tasks, they start facing inherent limitations when dealing with modern DNN architectures, as they feature increasingly heterogeneous topologies and heavily rely on activation functions involving complex operations.

This chapter presents Flex-SFU, a lightweight hardware accelerator for activation functions implementing non-uniform piecewise interpolation supporting multiple data formats. Non-Uniform segments and floating-point numbers are enabled by implementing a binary-tree comparison within the address decoding unit. An optimization algorithm based on stochastic gradient descent (SGD) with heuristics is proposed to find the interpolation func-
tion reducing the MSE, and capable of achieving better MSE compared to previous piecewise linear interpolation approaches. *Flex-SFU* Our experimental evaluation (Section 4.5), considering more than 700 computer vision and NLP models, shows that *Flex-SFU* can averagely improve the end-to-end performance of SoA AI hardware accelerators, while guaranteeing negligible accuracy losses and introducing low area and power overheads.

4.1 Introduction

To keep pace with the evolution of AI models, industry and academia are exploring novel hardware architectures, featuring heterogeneous processing units capable of achieving orders of magnitude improvements in terms of performance and energy efficiency with respect to general-purpose processors. As the execution time of SoA DNNs has been dominated by operations like convolutions and matrix-multiplications [66, 138], DNN hardware accelerators currently allocate most of their computational resources to specialized linear algebra cores, while leaving the execution of the other layers to general-purpose VPU [75, 98].

However, aiming at reducing training and inference time and enabling deployment on IoT and edge devices, recent deep learning research efforts are pushing towards decreasing the models dimensions while providing comparable or better accuracy. To this aim, recent networks increase their heterogeneity by introducing new layers featuring reduced operational intensity and new parameter-free layers. Specifically, parameter-free layers like ReLU are increasingly replaced with activation functions requiring a higher compute effort, such as GELU, SiLU, and Softmax, which are composed of several expensive operations like divisions and exponentiations [20]. Consequently, while modern networks succeed in reducing the execution time allocated to their convolutional layers, they also increase the proportion of execution time consumed by activation function computations within the DNN hardware accelerators. This shift in computational demands underscores the significant impact of activation function computations on the overall execution time of HPC hardware accelerators. Therefore, exploring novel hardware architectures capable of efficiently accelerating the computation of complex activation functions is becoming an increasingly important research topic.

In this chapter, we propose *Flex-SFU*, a flexible hardware accelerator for deep learning activation functions, integrated into general-purpose VPU and relying on a novel piecewise linear (PWL) approximation approach, capable of averagely improving the precision of previous PWL approximation approaches. The main novelty of *Flex-SFU* relies on its
4.2 Background and Related Work

Activation functions represent one of the most common and important layers of DNNs, as they apply non-linear transformations to the network feature maps. While ReLU has been widely used for many deep learning tasks, modern networks are using more complex activation functions [48] to achieve higher accuracies and avoid the well-known “dying ReLU effect” [102]. As these kernels require many complex operations (e.g. logarithm, division, exponentiation), they are typically accelerated via function approximation strategies, whose methods can be grouped into three main categories: polynomial, lookup table (LUT)-based, and hybrid.

Polynomial approximation methods [115,161] compute the activation functions through series expansions, such as Taylor and Chebyshev approximations. Although these methods feature high-precision computations, and reduce the latency compared to naïve computations, their hardware implementations are typically tailored to a specific activation function and are costly in terms of area, as their computation requires several multiply-add (MADD)
4.2. BACKGROUND AND RELATED WORK

operations. For example, the exponential function approximation in [115], based on a 6th-order Taylor expansion and targeting 16-bit fixed-point data, requires 6 multipliers and 5 adders to be computed.

*LUT-based* architectures [19, 88, 106, 158] are a popular approach to accelerate the computation of activation functions in deep learning. In this approach, a pre-computed table of activation function values (i.e., the LUT) is used to approximate the activation function. Specifically, *LUT-based* architectures subdivide the function input range into *intervals* and associate each *interval* to a specific function output, whose value is pre-computed and stored in memories used as LUTs. An addressing scheme is then used to map a given input [106] or an *interval* of input values [88] to a specific LUT address, holding the corresponding function output. This allows for a significant reduction in computation time with respect to *polynomial* approximation methods, as the activation function does not need to be evaluated for each input. Moreover, *LUT-based* solutions can be more flexible than *polynomial* methods, as programmable LUTs can store different sets of output data depending on the target activation function. However, they require a high area footprint to provide good accuracy. Indeed, as the function output is directly provided by the LUT, the approximation precision strictly depends on the number of *intervals* (i.e. the LUT depth) in a selected input range and on the output data size (i.e. the LUT width). Therefore, as the LUT depth grows with the number of input *intervals*, *LUT-based* solutions need to trade area consumption with supported input range, bounding the approximation to a specific region of the function, thus limiting its range.

To overcome these limitations, several works [62, 79, 84, 93, 95] explore *hybrid* solutions, which combine the *polynomial* and *LUT-based* approaches. As in *LUT-based* solutions, *hybrid* methods rely on PWL approximations exploiting LUTs. However, instead of directly providing the approximated function output, LUTs store the interpolation *segment coefficients* that are sent, together with the function input, to a MADD unit that computes the function output. For example, a PWL *hybrid* approach approximates a given activation function as \( N \) straight lines (i.e. *segments*), each satisfying the equation \( f(x) = m_i x + q_i \), for \( i \in \{0, 1, \ldots, N - 1\} \). A MADD operation is then used to evaluate the function output (i.e. \( f(x) \)) starting from a specific set of *segment coefficients* stored in the LUTs (i.e. \( m_i, q_i \)) and from the incoming input data (i.e. \( x \)).

The authors in [93] propose a hardware architecture to accelerate different activation functions on deep learning accelerators, supporting the 16-bit fixed-point format through a PWL approximation relying on a *hybrid* approach. They store the \( i \) *segments coefficients* (i.e. \( m_i \) and \( q_i \)) in a LUT having depth \( D \), and propose an addressing scheme based on a subset of the input most significant bits (MSBs) to load the proper *coefficients* set from the
4.2. BACKGROUND AND RELATED WORK

The input and the coefficients are thus forwarded to a MADD unit computing the function output (i.e. \( f(x) \)). Aiming at improving accuracy, in [62, 95], the authors propose hybrid architectures exploiting a second-order approximation. They exploit Horner’s rule [67] to split the second-order polynomial as a 2-steps MADD operation, and propose optimizations to improve area-efficiency [95] and accuracy [62].

Hybrid solutions outperform LUT-based approaches in terms of area and accuracy, as they are able to correctly approximate the whole segment starting from the coefficients stored in the LUTs instead of selecting a reference output value for a given interval. Moreover, hybrid approaches relax the constraint on the maximum function input range, as they allow approximating any function featuring boundaries that converge to a fixed slope.

However, current hybrid solutions present several limitations. ❶ They are tailored to a single input data type, either converted into a fixed-width fixed-point notation [47,84,93,95] or only considering a single floating-point format [62, 79]. Moreover, their LUT addressing schemes simply rely on a fixed subset of bits, such as the input data MSBs. However, these approaches lack flexibility, firstly because current accelerators support several data formats and SIMD computations [74], (e.g. from four 8-bit to one 32-bit elements/cycle), and secondly because their addressing schemes need to be tailored for each target function and input data type. ❷ Their approximation methods mainly rely on uniform interpolations (i.e. segments share the same length). Although this choice simplifies both

---

**Figure 4.1:** PWL approximation (left y-axis) and squared error (right y-axis) of GELU, exploiting uniform and non-uniform (i.e. Flex-SFU) interpolations, and considering 5 breakpoints (i.e. 4 segments) in the [-2,2] input range.
the HW (LUT\_addr = in\_data\[range\]) and the methodology to find the optimal segment length, it is suboptimal when there is a constraint on the number of total segments (i.e., LUT depth). Indeed, activation function approximations would benefit from non-uniform interpolation granularity among different function intervals, as it would allow increasing the density of segments on more sensitive intervals while relaxing their density on straight intervals. The advantages of supporting a non-uniform interpolation scheme are clearly depicted in Figure 4.1, which shows the PWL approximation of GELU exploiting both uniform and non-uniform interpolations. As Figure 4.1 shows, a uniform interpolation limits the approximation granularity, and suffers from high errors on intervals featuring higher non-linearities. On the opposite, non-uniform interpolations, such as the one proposed in this work, can select the best interval length by cleverly selecting each breakpoint location, with a consequent improvement in terms of MSE by 7× with respect to the uniform interpolation in the [−2;+2] range, while using the same number of breakpoints. Moreover, as depicted in the [−∞;−2], approximations based on uniform interpolations can diverge on the boundaries when the reference function has not converged to a fixed slope on the selected input range.

Although non-uniform strategies exist in the literature, they either rely on simply removing breakpoints from a uniform interpolation while maintaining similar precisions [69,81], or only optimize the interpolation error for narrow input ranges, leaving it diverging outside the selected input range with unknown impact on the end-to-end accuracy [47]. Although many related works [62,79,93,95] perform end-to-end accuracy evaluations on selected deep learning models, none of them quantify the accuracy impact of their solutions on a large set of networks. However, such analyses are crucial to verify the robustness of a given approximation method, as different models can suffer from different sensitivities to activation function errors, or can execute non-linear functions exploiting input ranges exceeding the approximation boundaries.

Overcoming these limitations is critical to enable efficient approximation functions acceleration in terms of both performance, area efficiency, precision, and end-to-end accuracy. Accordingly, we propose Flex-SFU, whose hardware architecture (detailed in Section 4.3) supports all the data sizes typically used by DNNs, and features linear throughput scaling with constant on-chip memory usage. Our reprogrammable hardware architecture implements an addressing scheme supporting non-uniform segments, whose optimal lengths minimizing the approximation MSE are determined by a novel PWL algorithm, described in Section 4.4.
Data Control Unit (DCU) dispatches input data among the other Flex-SFU handled by

depending on the input data value, Flex-SFU provides the proper coefficients to the VPU MADD units, computing the activation function output. Differently from the related work analyzed in Section 4.2, performing the address decoding exploiting a subset of incoming input data bits, Flex-SFU enables support for non-uniform interpolations, as it allows storing the value of each breakpoint on small on-chip memories, and to compare them with the incoming data to find the respective LUT address. As discussed in Section 4.2, this feature allows for higher flexibility than solutions only supporting uniform segments, as it permits the selection of the best length for each segment, thus minimizing the approximation error.

Flex-SFU extends the set of functional units available on current VPUs targeting deep learning, acting as a special function unit (SFU) capable of accelerating activation functions via PWL approximation. Its execution is handled by three custom instructions extending the target VPU ISA, namely \(1d\).bp(), \(1d\).cf(), and \(exe\).af(). Each instruction includes a source vector operand, and a vector elements width field, and the \(exe\).af() also features a destination vector operand. The proposed instructions are decoded by the VPU, and then handled by Flex-SFU, whose main architectural components are depicted in Figure 4.2. A Data Control Unit (DCU) dispatches input data among the other Flex-SFU units. Specifically, \(1d\).bp() and \(1d\).cf() source data, holding either breakpoints or PWL coefficients,
are sent by the DCU to the address decoding unit (ADU) or the lookup table cluster (LTC) unit, and stored in *SIMD single-port memories*. These instructions must be executed only once, when a different activation function has to be computed, and can be pre-executed while other accelerators compute units (*e.g.* the main tensor-unit) are still computing the activation function inputs. Therefore, as discussed in Section 4.5.1, they do not introduce a large overhead in the overall computation. Once breakpoints and LUT coefficients have been loaded in the ADU and LTC units, multiple `exe.af()` can be executed to compute the activation function outputs. These operations are handled by the DCU, which streams the input data through the pipeline composed of the ADU and the LTC. As Figure 4.2 shows, the ADU functionality resembles a binary search tree (BST). Each ADU stage defines a BST level, and exploits *SIMD single-port memories* to implement BST nodes holding breakpoints, which are ordered to allow traversing one BST level per stage to search for the proper LTC address depending on the input data. Each cycle, a *SIMD comparator* supporting both fixed-point and floating-point number formats determines if the current input data is greater or smaller than the breakpoint loaded from memory exploiting the `cmp_o` signal, whose value can be either 1 or 0, respectively. The comparison output and the input address are then used by the *Next Address Generator* unit to find the subsequent ADU stage address, namely $a_o$, according to the following expression:

$$a_o = (a_i << 1) + cmp_o$$  \hfill (4.1)

Specifically, as each ADU stage stores a BST level, the $a_o$ value of each stage is computed by identifying the correct BST level siblings through $a_i << 1$, and by identifying the correct BST node among siblings through the `cmp_o` signal.

The last ADU stage performs the comparison among the BST leaves, thus finding the proper LUT address which is forwarded to the LTC unit. Finally, the LTC loads the appropriate segment coefficients, and sends them and the delayed input data to the VPU MADD functional units, computing the activation function output.

The memory-mapping strategy exploited by the ADU and LTC units consists of four *SIMD single-port memories*. The bit-width of each memory is equal to the product between the minimum bit-width supported by the accelerator (*e.g.* 8-bit) and the number of coefficients, whose value is set to 1 for the ADU and to 2 for the LTC.

Each memory is accessed separately in case of computations based on 8-bit data (*e.g.* $b_0^i, b_1^i, b_2^i, b_3^i$ are accessed as four separate 8-bit elements), while for 16-bit computations each data is segmented among two subsequent memories (*e.g.* $b_0^i - b_1^i$ and $b_2^i - b_3^i$ are accessed as two 16-bit elements), in such a way to support an input throughput of two
16-bit elements/cycle. Similarly, the same data is partitioned among the four 8-bit memories in case of 32-bit computations (e.g. $b_0 - b_1 - b_2 - b_3$ are accessed as a single 32-bit data), allowing to support a throughput of one 32-bit element/cycle, while reusing the same memories.

As shown in Figure 4.2, to allow for further scalability, the Flex-SFU parallelism can be tuned by increasing the number of instantiated clusters, namely $N_c$, to match the underlying VPU throughput. We design both the ADU and LTC memories featuring multiple read ports. A multi-ported memory design allows sharing breakpoints and coefficients among clusters, reducing the amount of replication and thus saving area. Note that, as VPUs are typically optimized for throughput, we design Flex-SFU exploiting pipelining, thus enabling a steady-state performance of $N_c \times$ 32-bit/cycle, while avoiding dead-locks by design.

### 4.4 Flex-SFU Approximation Methodology

We rely on a PWL approximation, defining the interpolated and steady function $\hat{f}(x)$ as:

$$f(x) \approx \hat{f}(x) = \begin{cases} 
  m_l(x - p_0) + v_0 & x \leq p_0 \\
  \frac{v_{i+1} - v_i}{p_{i+1} - p_i}(x - p_i) + v_i & p_i < x < p_{i+1}, \\
  0 < i < n - 1 \\
  m_r(x - p_{n-1}) + v_{n-1} & x \geq p_{n-1}
\end{cases}$$

with $n$ breakpoints $p_i$, $(n+1)$ linear segments, and $n$ function values at the breakpoints $v_i = \hat{f}(p_i)$. The most left and right segments are calculated with values $v_0$ and $v_{n+1}$, using slopes $m_l$ and $m_r$, while the inner segments of each breakpoint $p_i$ are linearly interpolated through its value $v_i$ and the following breakpoint-value pair $[p_{i+1}; v_{i+1}]$.

To find the breakpoint-value pairs, we start with uniformly distributed breakpoints and exact function values. We use the Adam optimizer [80] (with $lr=0.1$, momenta=(0.9, 0.999)) and the Plateau LR scheduler. We choose the MSE between the interpolated function $\hat{f}$ and the target function $f$ on the interval $[a, b]$ as the loss function:

$$\mathcal{L}_{[a,b]}(\hat{f}, f) = \frac{1}{b-a} \int_a^b \left( \hat{f}(x) - f(x) \right)^2 dx$$

\footnote{The inclusion of the following section within this thesis is made for the purpose of thoroughness and completeness, but it contains content that has been mainly investigated by Dr. Renzo Andri from the Zurich Huawei Research Center.}
Aiming to avoid stalls in sub-optimal local minima during the optimization process, we extend our optimization algorithm by removing breakpoints and reinserting them at a better location.

**Removal loss:** We define the removal loss $\ell_{rm}^i$ as the loss of the interpolated function if the breakpoint $p_i$ is removed. We then remove the breakpoint with the minimal removal loss, $p_{\text{remove}}$:

$$p_{\text{remove}} = \arg \min_{p_i} \ell_{rm}^i, \quad \ell_{rm}^i = \mathcal{L}_{[a,b]}(f(p_j,v_j)_{j \neq i}, f).$$

**Insertion loss:** On the other hand, we define the insertion loss $\ell_{ins}^i$ as the loss over the $i$-th segment, and insert a new breakpoint in the center of the segment with the highest insertion loss:

$$\left(\frac{p_{\text{insert}}}{v_{\text{insert}}}\right) = \arg \max \left(\frac{p_i + p_{i+1}}{v_i + v_{i+1}}\right) \ell_{ins}^i, \quad \ell_{ins}^i = (p_{i+1} - p_i)\mathcal{L}_{[p_i,p_{i+1}]}(f, f).$$

**Boundary condition:** All relevant activation functions converge outside the interpolation interval to a constant value or an asymptote. To avoid large errors outside of the interpolation interval, unless noted otherwise, we define boundary conditions for value and slope for the most left and the most right segments, such that they lie on the asymptote of the function:

$$m_l = \lim_{x \to -\infty} \frac{f(x)}{x}, \quad v_0 = m_l p_0 + \lim_{x \to -\infty} (f(x) - m_l x),$$
$$m_r = \lim_{x \to +\infty} \frac{f(x)}{x}, \quad v_{n-1} = m_r p_{n-1} + \lim_{x \to +\infty} (f(x) - m_r x)$$

For example, considering GELU, this resolves to $m_l = 0, v_0 = 0, m_r = 1, v_{n-1} = p_{n-1}$. Notably, $p_0$ and $p_{n-1}$ themselves are still learned. In this way, the interpolated function converges to the original function for values far from the boundary breakpoints. This comes at a small cost in error close to the boundary breakpoints.

**Optimization strategy:** We initialize the Flex-SFU function interpolation with uniformly distributed breakpoints. Then we optimize with SGD until convergence. After this, we remove and insert one breakpoint as described above, and retrain with a lower learning rate. We reiterate until the removal and insertion points converge. Note that we perform this optimization for each function, and we substitute the layers within the DNN models without any retraining for ease of use.
Figure 4.3: Throughput of Flex-SFU, in terms of number of computed activations (i.e. \( GAct \)) per second, as a function of the input tensor size, accounting for different bit-widths \((b)\) and lookup table cluster (LTC) depths \((d)\)

### 4.5 Experimental Evaluation

We evaluate Flex-SFU exploring both stand-alone and end-to-end analyses. Specifically, Section 4.5.1 provides a performance, area, and power evaluation of Flex-SFU, while Section 4.5.2 analyzes the precision of the approximation methodology proposed in Section 4.4, comparing it with the SoA. Finally, Section 4.5.3 evaluates Flex-SFU on a commercial hardware accelerator, analyzing both end-to-end performance and accuracy on more than 700 DNN models, considering the most relevant activation functions over different LTC sizes.

#### 4.5.1 Performance, Power and Area Analyses

We implement the proposed hardware accelerator in RTL, and perform synthesis and PnR for a 28nm CMOS technology node. We evaluate several Flex-SFU configurations in terms of performance, area, and power, varying the number of segments from 4 to 64 while considering \(N_c = 1\) and a target frequency of 600 MHz. We then extend our analysis to multiple clusters to evaluate the proposed solution scalability.

**Single Cluster Evaluation:** Figure 4.3 shows the throughput of Flex-SFU, accounting for the time spent on both \(1d.\, bp()\), \(1d.\, cf()\), and \(exe.\, af()\), across input tensors ranging from 8 KB to 32 KB. All the analyzed Flex-SFU combinations reach the steady-state performance for input tensors larger than 256 32-bit data, gaining 0.6 GAct/s, 1.2 GAct/s, and
### 4.5. Experimental Evaluation

Table 4.1: Flex-SFU Characterization for $N_c = 1$ at $f = 600$ MHz in 28nm CMOS

<table>
<thead>
<tr>
<th>LTC Depth (i.e. # Segments)</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency [cycles]</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Power [mW]</td>
<td>1.4</td>
<td>1.7</td>
<td>2.2</td>
<td>2.8</td>
<td>3.7</td>
</tr>
<tr>
<td>ADU Area [%]</td>
<td>34.2%</td>
<td>41.2%</td>
<td>43.7%</td>
<td>46.0%</td>
<td>41.6%</td>
</tr>
<tr>
<td>LTC Area [%]</td>
<td>31.3%</td>
<td>34.9%</td>
<td>44.1%</td>
<td>46.6%</td>
<td>53.4%</td>
</tr>
<tr>
<td>Total Area [µm²]</td>
<td>2572.4</td>
<td>3593.0</td>
<td>5846.0</td>
<td>9791.3</td>
<td>14857.2</td>
</tr>
</tbody>
</table>

2.4 GAct/s in terms of throughput, considering 32-, 16-, and 8-bit data sizes, corresponding to an energy efficiency ranging from 158 GAct/s/W to 1722 GAct/s/W. The performance overhead introduced on small input tensors, due to the time spent on storing breakpoints and coefficients into the ADU and the LTC units, can be completely neglected by pre-executing the `ld.bp()`, `ld.cf()` instructions. Note that the throughput reported in Figure 4.3 saturates to 1 Act/cycle, 2 Act/cycle, and 4 Act/cycle for 32-, 16-, and 8-bit data sizes at 600 MHz, proving that Flex-SFU can reach the theoretical peak performance discussed in Section 4.3, accelerating complex DNN activation functions exploiting the same computation time typically required by simple operations like ReLU.

Table 4.1 details the characterization of Flex-SFU, obtained after the PnR step and considering from 4 to 64 segments, reporting a total power consumption ranging from 1.4 mW to 3.7 mW, and a total area requiring from 2572 µm² to 14857 µm². Flex-SFU latency increases by 1 clock cycle per doubling of the number of segments (i.e. for each additional ADU stage).

**Scalability Analysis:** Aiming to explore the actual scalability of the proposed solution in terms of area and performance, we evaluate Flex-SFU exploiting higher $N_c$ values.

Figure 4.4 proposes an area-based design space exploration of Flex-SFU, considering from 4 to 64 segments and $N_c$ ranging from 1 to 64. Specifically, Figure 4.4 explores the Flex-SFU area considering ADU and LTC memories either replicated on each cluster (i.e. featuring a single read port) or leveraging on multiple read ports, thus sharing the same memory on multiple adjacent clusters. As Figure 4.4 shows, exploiting multiple ports allows reducing the total area up to 63% with respect to the single port Flex-SFU implementation, still respecting the target timing constraints.

Performance-wise, our scalability analysis reports a linear performance increase with the $N_c$ increasing. For example our evaluation reports a steady-state throughput of 38.3 GAct/s, 76.7 GAct/s, and 153.4 GAct/s considering $N_c$ equal to 64, corresponding to 63.9 Act/cycle, 127.8 Act/cycle, and 255.7 Act/cycle for 32-, 16-, and 8-bit data sizes at 600 MHz. These results clearly show that Flex-SFU can scale its performance linearly by
4.5. EXPERIMENTAL EVALUATION

Figure 4.4: Area scalability analysis, accounting for several numbers of segments, exploring the area improvements of memories exploiting either a single read port (SP) or multiple read ports (i.e. equal to the number of clusters $N_c$).

Increasing the number of clusters, as using 64 clusters allows gaining $63.9 \times$ more performance than the single cluster configuration analyzed in Figure 4.3.

To investigate the area and power impact of Flex-SFU on high-performance VPUs, we perform a back-of-the-envelope integration of Flex-SFU into the RISC-V VPU proposed by Perotti et al. in [122], composed of 4 lanes and supporting a maximum data size of 64-bit. Our evaluation, considering four Flex-SFU instances (i.e. one instance per lane) featuring $N_c = 2$ (i.e. supporting from $1 \times 64$-bit to $8 \times 8$-bit elements/cycle), shows that Flex-SFU only accounts for 2.2%, 3.5% and 5.9% of the total area for a LTC depth of 8, 16 and 32 elements, respectively, while consuming from 0.5% to 0.8% of the total power.

### 4.5.2 Function Approximation Precision Analysis

In Figure 4.5, we investigate MSE and maximum absolute error (MAE) of the most representative activation functions. We select the interpolation interval within $[-10, 0.1]$ for Exp, and within $[-8, 8]$ for the other functions. The boundary breakpoints lie on the functions asymptote to reduce the error outside the interpolation interval. We interpolate Exp for negative values to be used in Softmax, typically requiring exponentiation implemented...
4.5. EXPERIMENTAL EVALUATION

Figure 4.5: Error analysis for a set of activation functions, considering from 4 to 64 breakpoints. Interpolation intervals are in the range [-10, 0.1] for the exponential function (Exp), and in the range [-8, 8] for the other functions.

with (vector-wide) maximum subtraction \(i.e. \exp(x_i - \max_j x_j)\). As detailed in Figure 4.5, the approximation precision of the analyzed functions averagely improves MSE and MAE by \(15.9 \times\) and \(3.8 \times\) per doubling of the number of breakpoints. Moreover, all the interpolations featuring more than 16 breakpoints reach a MSE lower than 1 Float16 unit of least precision (ULP), defined as the single-bit error at a base of 1.

In Table 4.2, we compare Flex-SFU with other PWL interpolation methods, considering the same interpolation range and number of breakpoints. Following most of the previous works, \[62, 79, 84, 93, 95\], we evaluate Flex-SFU exploiting the average absolute error (AAE) metric, squaring it \(i.e. \text{sq-} \text{AAE}\) to match the same MSE order of magnitude. Furthermore, we compare with the equivalent number of breakpoints of previous works exploiting symmetry \[19, 84\]. As Table 4.2 shows, our method outperforms all the other PWL approaches, by a factor ranging from \(2.3 \times\) to \(88.4 \times\), with an average of \(22.3 \times\).

The method proposed by Gonzalez et al. \[62\] exploits a second-order piecewise but not-steady interpolation, averagely achieving \(4.3 \times\) better MSEs than Flex-SFU on Tanh, Sigmoid, and SiLU. Although Flex-SFU can be easily extended to support a second-order interpolation, second-order approximations feature high area overheads, requiring to double the number of VPU MADD units to guarantee the same throughput of the proposed solution, as well as larger LUTs able to store an additional interpolation coefficient. We extend the function approximation order discussion in Section 4.6.1.
Table 4.2: Comparison of our MSE-optimized method with other PWL Interpolation Methods with the same number of breakpoints and range

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Error sq-AAE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Approximation Range</td>
</tr>
<tr>
<td>[84]</td>
<td>[-8, 8]</td>
</tr>
<tr>
<td>[93]</td>
<td>[-3.5, 3.5]</td>
</tr>
<tr>
<td>[95]</td>
<td>[-8, 8]</td>
</tr>
<tr>
<td>[79]</td>
<td>[1/64, 4]</td>
</tr>
<tr>
<td>[19]</td>
<td>[-4, 4]</td>
</tr>
<tr>
<td>[84]</td>
<td>[-8, 8]</td>
</tr>
<tr>
<td>[93]</td>
<td>[-7, 7]</td>
</tr>
<tr>
<td>[93]</td>
<td>[-7, 7]</td>
</tr>
<tr>
<td>[79]</td>
<td>[-8, 8]</td>
</tr>
<tr>
<td>[79]</td>
<td>[1/64, 4]</td>
</tr>
<tr>
<td>[19]</td>
<td>[-4, 4]</td>
</tr>
<tr>
<td>[95] GeLU</td>
<td>[-8, 8]</td>
</tr>
</tbody>
</table>

* SoA reports the average absolute error (AAE).
‡ Numbers in MSE.
‡‡ Uses symmetry to halve the number of segments.

4.5.3 End-to-End Evaluation

We evaluate Flex-SFU on a commercial Huawei Ascend 310P AI processor [97], exploiting a benchmark suite targeting 628 computer vision and 150 NLP networks from PyTorch Image Models (TIMM) and Hugging Face, respectively. This accelerator represents an ideal candidate to demonstrate the benefits that Flex-SFU can provide to SoA DNN accelerators, as it hosts a specialized matrix multiplication unit computing up to 4096 MAC/cycle, and processes the DNN activation functions on a general-purpose high-performance VPU.

To perform our performance evaluation, we convert each benchmark suite model from Pytorch 1.11 [121] to ONNX 1.12 [22] with opset version 13, and we replace each activation function of the resulting model graph with a custom ONNX operator, implementing a set of instructions supported by the Huawei Ascend ISA, and whose latency and throughput match the Flex-SFU metrics presented in Section 4.5.1. Then, we compile both the baseline and the Flex-SFU-enhanced ONNX models for the Ascend AI processor with Ascend Tensor Compiler (ATC) v5.1 and run them on the target accelerator to extract and compare their end-to-end inference run time. In our evaluation, we compute each model using all 8 cores of the Ascend 310P AI processor in parallel with batch size equal to 1, considering
4.5. EXPERIMENTAL EVALUATION

Figure 4.6: End-to-end model zoo performance evaluation, performed on eight Huawei Ascend 710 AI processor running in parallel. Different colors highlight the most frequently used activation function of each network.

Figure 4.6 summarizes the execution time improvements of the proposed benchmark suite when exploiting Flex-SFU, highlighting the reference family and most frequent activation function of each model. We obtained comparable performance results for batch sizes equal to 16, 32, and 128. As Figure 4.6 shows, Flex-SFU matches the performance of models primarily relying on lightweight activation functions (i.e. ReLU, Leaky ReLU), not introducing any overhead in their computation, and greatly improves the execution time of networks relying on more complex activation functions. Specifically, including the models based on ReLU, whose baseline execution time matches the Flex-SFU performance, Flex-SFU allows gaining 17.3%, 17.9%, 29.0%, and 45.1% performance on ResNets, Vision Transformers, NLP Transformers, and EfficientNets models, while reaching 2.1× more performance on DarkNets models. Overall, Flex-SFU reaches 22.8% better performance on the considered model zoo computation, improving the execution time of models relying on complex activation functions by 35.7% on average, and reaching a performance peak of 3.3× on the computation of resnext26ts.

We evaluate the accuracy impact of Flex-SFU for DNNs in the TIMM database on the ImageNet dataset [43] by comparing the top-1 accuracy on the validation set between the
reference model and one where the activations are replaced with *Flex-SFU*. We also evaluate the top-1 accuracy exploiting uniform PWL to further compare *Flex-SFU* with the SoA approaches, exploiting uniform distance between *breakpoints*. Figure 4.7 shows the cumulative distribution of TIMM models as a function of the accuracy drop with respect to the baseline FP32 accuracy, considering either *Flex-SFU* or a uniform interpolation. As Figure 4.7 shows, *Flex-SFU* outperforms the approximation based on uniform interpolation for all the considered *breakpoints* configurations, ranging from 4 to 64. Specifically, for all the considered configurations, the uniform interpolation roughly requires double of the *breakpoints* to perform as *Flex-SFU*. Indeed 80.8%, 96.6%, and 99.7% of the TIMM models show less than 0.3% accuracy drop when computed with *Flex-SFU* featuring 8, 16, and 32 *breakpoints*. On the other hand, the uniform interpolation only allows 57.8%, 78.4%, and 92.7% of the models to feature less than 0.3% accuracy drop, considering the same number of *breakpoints*. *Flex-SFU* performs almost losslessly on the 64 *breakpoints* configuration, showing a maximum accuracy drop of 0.04% on the computation of *swinv2_large_window12_192_22k*, while uniform interpolation exploiting the same configuration shows higher drops on 6% of the models, with a maximum accuracy drop of 1.9% on the computation of *mobilenetv3_small_075*.

We note that networks using SiLU are the most sensitive to approximation. For ex-
ample, to feature an accuracy drop smaller than 0.17%, mobilevit and halonet50ts require 32 breakpoints, while lambda_resnet50ts and mixer_b16_224_miil require 16 breakpoints. Hardswish is the second most sensitive activation function, with lcnet and mobilenetv3_small requiring 32 breakpoints, and hardcorenas, fbnet, and mobilenetv3_large requiring 16 breakpoints to show losses smaller than 0.15%. Finally, GELU-based sebotnet33ts_256, mixer and crossvit achieve lossless accuracy drops with 16 breakpoints.

4.6 Discussion

4.6.1 Function Approximation Order Trade-Offs

While this chapter primarily focuses on Flex-SFU performing PWL approximations, its hardware microarchitecture can be configured to exploit higher approximation orders. For instance, each LTC cell can be set to accommodate three segment coefficients instead of two, enabling a second-order approximation for the analyzed activation functions. When transitioning from a first-order to a second-order interpolation with 4 to 64 segments, the Flex-SFU area increases from 25.8% to 42.9%, respectively. Correspondingly, power consumption rises from 12.5% to 25%. However, it should be noted that achieving the same throughput with a second-order approximation necessitates doubling the number of MADD units compared to a first-order approximation, resulting in a larger overall area footprint. Therefore, the optimal design decision depends on the number of arithmetic functional units in each VPU lane. In cases where the target VPU features only one arithmetic functional unit per lane, Flex-SFU can be operated in either a high-throughput mode using a first-order approximation, where new data is generated on every clock cycle, or a high-precision mode employing a second-order approximation with enhanced precision but operating at half the throughput. The desired execution mode can be dynamically selected at runtime based on the target network accuracy and performance requirements. We are currently extending the approximation methodology described in Section 4.4 to support second-order approximations of the considered activation functions. This effort aims to establish a better understanding of the area overhead and precision improvements associated with higher-order approximations.

4.7 Summary

Modern DNN workloads increasingly rely on activation functions consisting of computationally complex operations. This poses a challenge to current accelerators optimized for convolutions and matrix-matrix multiplications.
We proposed Flex-SFU, a scalable hardware accelerator for DNN activation functions on VPUs based on a novel design supporting non-uniform breakpoints locations, and performing 8-, 16-, and 32-bit computations based on both fixed-point and floating-point data formats. Flex-SFU exploit a novel LUT address decoder exploiting binary-tree structure, supporting multiple data formats and non-uniform segment lengths. Our evaluation shows that Flex-SFU achieves on average $22.3 \times$ better MSE compared to previous PWL interpolation approaches. The evaluation with more than 700 computer vision and NLP models shows that Flex-SFU can, on average, improve the end-to-end performance of SoA AI hardware accelerators by 35.7%, achieving up to $3.3 \times$ speedup with negligible impact in the models accuracy, and only introducing an area and power overhead of 5.9% and 0.8% relative to the baseline VPU.
Chapter 5
Summary and Conclusion

In the past decade, the advancement of DNNs has achieved unprecedented progress, enabling numerous application domains to leverage deep learning for enhanced outcome quality. However, this remarkable progress comes with inherent challenges due to the large parameter sets and extensive computational requirements of DNNs. These challenges pose significant obstacles for modern computing systems across various domains, spanning from energy- and latency-constrained edge and mobile devices to HPC and cloud accelerators optimized for high throughput.

In this thesis, we address the aforementioned challenges by exploring several research paths. Firstly, we investigate a novel mathematical technique, called binary segmentation, capable of reducing the arithmetic complexity of linear algebra computations based on narrow integers, and we propose a hardware architecture, called Bison-e to efficiently exploit binary segmentation on edge CPU architectures. We show that Bison-e significantly enhances the performance of linear algebra kernels operating on narrow integers, and requires low area and energy costs. Building upon this, we introduce a novel HW-SW co-designed architecture, called Mix-GEMM, that leverages binary segmentation to accelerate quantized DNNs on edge CPUs by performing SIMD operations exploiting the off-the-shelf processor FUs, enabling computations with arbitrary precision among narrow integers. Mix-GEMM extends SoA matrix-matrix multiplication approaches to narrow integers, exhibiting scalable performance with decreasing computation data sizes. Finally, we examine potential computational bottlenecks of large-scale DNN accelerators, and we propose Flex-SFU as a solution to accelerate complex activation functions. Flex-SFU exploits a PWL approximation approach, and features non-uniform segmentation and multiple data types to achieve high accuracies in approximating DNNs activation functions.
5.1 Overview of the Main Results

The main results and contributions can be summarized as follows.

- **Bison-e**: We show that binary segmentation can reduce the arithmetic complexity of the inner-product kernel computation from $3 \times$ to $19 \times$ and the linear convolution kernel computation from $2.5 \times$ to $90.5 \times$ on 64-bit CPUs, considering computations from 8-bit to 1-bit, respectively. Our experimental evaluation reveals that the proposed microarchitecture achieves performance improvements ranging from $4.7 \times$ to $19.3 \times$ compared to the baseline scalar processor, when computing the AlexNet and VGG-16 DNNs, as well as comparable or higher energy efficiency than an edge VPU for the same tasks. Remarkably, Bison-e achieves these advancements while utilizing less than 0.07% of the total SoC area and 0.04% of the total power consumption;

- **Mix-GEMM**: We accelerate the computation of DNNs by proposing an HW-SW co-designed architecture performing matrix multiplications, and achieving from $4.1 \times$ to $10.2 \times$ better performance than the baseline processor performing the same kernel at 8-bit precision. Notably, Mix-GEMM achieves computation rates ranging from 4.8 GOPS to 13.6 GOPS for representative CNNs such as AlexNet, VGG-16, ResNet-18, MobileNet-V1, RegNet-x-400mf, and EfficientNet-B0, while exhibiting energy efficiency ranging from 524.3 GOPS/W to 1.3 TOPS/W. In comparison to a commercial RISC-V processor running OpenBLAS with FP32 precision, Mix-GEMM achieves performance gains ranging from $5.7 \times$ to $15.1 \times$. It also outperforms a commercial Arm core utilizing SIMD computations and a SoA matrix-matrix multiplication library for 8-bit computations by up to $2.6 \times$, all while occupying just 1% of the underlying RISC-V SoC;

- **Flex-SFU**: We propose a hardware accelerator for complex activation functions, that outperforms previous works exploiting PWL approximations by an average factor of $22.3 \times$. We show that the integration of the Flex-SFU microarchitecture in VPU incurs modest area and power overheads of 5.9% and 0.8%, respectively. By employing Flex-SFU, the execution time of significant DNNs can be improved by up to $3.3 \times$, with an average improvement of 35.7% across more than 600 computer vision and NLP models. Notably, this is achieved while maintaining a maximum end-to-end accuracy drop of merely 0.04% when considering 64 breakpoints in comparison to the FP32 baseline.
5.2 Outlook

In the following, we provide an overview of the research directions we consider promising within the context of the contributions discussed in this thesis.

Exploring Binary Segmentation in Heterogeneous Architectures
Although the DSE methodology proposed in Section 2.3 can be easily extended to any hardware architecture, this thesis explores the usage of the binary segmentation technique on CPUs. However, we believe its features can also be applied to heterogeneous architectures, such as GPUs and FPGAs. In particular, FPGAs-based accelerators represent a good candidate for binary segmentation, as they are particularly effective for computations based on narrow-integers and fixed-point data, and exploit DSP units whose bitwidth has a fixed size. For example, the latest Xilinx Ultrascale and Versal architecture [56,91] feature MAC units of $27 \times 18$ and $27 \times 24$ respectively, which is clearly oversized for quantized DNNs computations exploiting bitwidths equal or lower than 8 bits. As a result, novel dataflow accelerators exploiting binary segmentation as a main computational pillar can be explored to enhance DNN computations on FPGAs.

Expand Mix-GEMM Data Level Parallelism
Modern edge and mobile CPU architectures feature SIMD units to improve their efficiency in compute-intensive applications. Although we show that the proposed solution can achieve comparable or higher performance than general-purpose SIMD units (see Section 3.5), Mix-GEMM can leverage on the off-the-shelf MAC units of the underlying processor to further improve the application throughput, while requiring minimal modifications and area overhead with respect to the implementation described in Section 3.3.2. Vectorizing the memory operations performed by the proposed µ-engine GEMM software library, as well as the custom operations discussed in Section 3.3.1 represent a promising research direction, as it would enable significant performance improvements with respect to the current Mix-GEMM implementation.

Explore Higher Activation Function Approximation Orders
As detailed in Section 4.6.1, Flex-SFU can be configured to support higher interpolation orders. Therefore, exploring the impact of such approximation orders in the context of DNNs inference would allow increasing the design space of Flex-SFU in terms of precision, performance, and area. Our preliminary results in this direction show that exploiting a second-order approximation for the activation functions considered in Flex-SFU would
allow improving the MSE and MAE errors by roughly $10 \times$ when compared to a first-order approximation considering the same number of breakpoints, and would allow achieving precisions closed to the baseline FP32 data format for activation functions featuring second-order degree such as Hardswish.

**DNNs Training with PWL-based Activation Functions**

The analysis conducted in Section 4.5.3 primarily focuses on the accuracy evaluation of DNNs inference. However, it is important to note that the target systems considered in Chapter 4 could potentially derive benefits from the application of Flex-SFU not only during inference but also in the training phase. Therefore, it is worthwhile to explore the impact of utilizing PWL-based activation functions for DNNs training, which represents an intriguing avenue for future investigation. This analysis should encompass an examination of how the approximation of activation functions can affect the final model accuracy, as well as an exploration of whether the adoption of Flex-SFU influences the learning curve, specifically the number of epochs required to achieve the target accuracy.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Accuracy over time evolution of 800+ computer vision DNNs [7].</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Accuracy against number of operations and parameters for 800+ computer vision DNNs from [7].</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Activation functions distribution by year of model publication, extracted from 700+ SoA AI models of the TIMM and Hugging Face collections.</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Gantt chart describing the activities held during the Ph.D. timeline, including Ph.D. technical activities, papers/patent preparations, engineering activities, and internship periods.</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Examples of IP (a) and LC (b) kernel computations via binary segmentation, with clustering widths of 7-bit and 5-bit, respectively.</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Maximum input-cluster size achievable on 32-bit and 64-bit architectures, for data sizes ranging from 1-bit to 16-bit. The input-cluster size is defined as the number of elements that can be packed in a single register, following the binary segmentation constraints.</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Arithmetic complexity reduction when computing IP (a) and LC (b) kernels on 64-bit architectures, accounting for multiplications and additions.</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Amount of time spent in Pre-Processing, Processing and Post-Processing phases of the IP (a) and LC (b) kernels computed via binary segmentation on 64-bit architectures.</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Bison-e block diagram.</td>
<td>25</td>
</tr>
<tr>
<td>2.6</td>
<td>Improved overlap-add using Bison-e. Different colors represent different outer loop iterations.</td>
<td>29</td>
</tr>
</tbody>
</table>
Figure 2.7: Execution time of IP (a) and LC (b) kernels relying on a naïve implementation or binary segmentation. The three binary segmentation implementations either rely on software exploiting the standard RISC-V ISA (RV64IM), leverage on bit-manipulation instructions (bit-manip), or exploit Bison-e.

Figure 2.8: Speed-ups (a, c), and energy efficiency (b, d) of the AlexNet and the VGG-16 CNNs with respect to the scalar implementation, exploiting either Bison-e or the VPU.

Figure 2.9: Approximate string matching kernel speed-up exploiting either Bison-e (dark-green bars) or the VPU (light-green bars) with respect to the scalar implementation, featuring a 4 (a) and a 256 (b) letters alphabet.

Figure 2.10: Layout of the DRAC SoC including Bison-e, highlighted in green and circled in black, for the 65nm technology. This design, also including the VPU, a PLL, an ADC and several peripherals controllers, is ready for fabrication.

Figure 3.1: Data flow and allocation of the proposed macro-kernel and µ-kernel procedures, built upon the BLIS implementation of DGEMM. Note that both $kca$ and $kua$ are twice as $kcb$ and $kub$, indicating that the data size of $A$ is two times larger than the one of $B$.

Figure 3.2: Representation of three activation-weight configurations. Each µ-vector holds a different number of elements, depending on the element data size. Different colors represent different µ-engine execution cycles (i.e., different selected sub-µvectors).

Figure 3.3: µ-engine architecture, integrated in the processor FU. Different colors represent compute or memory unit details, according to Figure 3.4 (green, pink, blue, orange, and grey), or Figure 3.1 (red and yellow).

Figure 3.4: Example of inner-product computation (i.e., $4 \times 3 + 7 \times 2 + 3 \times 0 + 6 \times 1 = 32$) evaluated via binary segmentation through a pipelined approach. Each color represents a step required by binary segmentation to compute the inner-product. Each tick depicts the pipeline status over time.

Figure 3.5: Speed-up of Mix-GEMM over the baseline BLIS-based DGEMM algorithm on square input matrices. Configurations sharing the activations data size ($a$) are represented with the same color, with different line patterns differentiating the weights data size ($w$).
3.6 Performance vs. accuracy Pareto frontier for the selected CNNs. Labels represent activations and weights data sizes (a and w), respectively. We measure the performance of the quantized network exploiting Mix-GEMM, while the FP32 performance is measured exploiting OpenBLAS running on the SiFive U740 processor.

3.7 Post-PnR layout, targeting the Global Foundries 22nm FDSOI technology node, of the SoC integrating the proposed µ-engine (highlighted in green).

3.8 Mix-GEMM workflow diagram considering training and inference.

4.1 PWL approximation (left y-axis) and squared error (right y-axis) of GELU, exploiting uniform and non-uniform (i.e. Flex-SFU) interpolations, and considering 5 breakpoints (i.e. 4 segments) in the [-2,2] input range.

4.2 Flex-SFU hardware architecture, considering a PWL approximation, integrated into the main VPU as an additional functional unit, considering Look-up table clusters (LTCs) capable of storing 8 segment coefficients. Memory cell superscripts represent breakpoints and coefficients IDs, while memory cell subscripts represent data slices.

4.3 Throughput of Flex-SFU, in terms of number of computed activations (i.e. GAct) per second, as a function of the input tensor size, accounting for different bit-widths (b) and lookup table cluster (LTC) depths (d).

4.4 Area scalability analysis, accounting for several numbers of segments, exploring the area improvements of memories exploiting either a single read port (SP) or multiple read ports (i.e. equal to the number of clusters Nc).

4.5 Error analysis for a set of activation functions, considering from 4 to 64 breakpoints. Interpolation intervals are in the range [-10, 0.1] for the exponential function (Exp), and in the range [-8, 8] for the other functions.

4.6 End-to-end model zoo performance evaluation, performed on eight Huawei Ascend 710 AI processor running in parallel. Different colors highlight the most frequently used activation function of each network.

4.7 Cumulative distribution of End-to-End accuracy drop over 600+ DNNs of TIMM [154], considering both uniform and non-uniform PWL interpolations.
List of Tables

2.1 *Bison-e* Control parameters list .................................................. 26
2.2 Overview of the *Bison-e* custom instructions ................................. 27
2.3 *Bison-e* Area and Power Consumption .......................................... 36

3.1 *Mix-GEMM* optimal parameters obtained in the DSE. ...................... 55
3.2 *µ-engine* Area Breakdown ................................................................. 62
3.3 Comparison with state-of-the-art: performance and efficiency ranges ordered according to the supported data sizes (*e.g.*, 8b – 2b). Results gathered from published papers. ....................................................... 65

4.1 *Flex-SFU* Characterization for $N_c = 1$ at $f = 600$ MHz in 28nm CMOS . 81
4.2 Comparison of our MSE-optimized method with other PWL Interpolation Methods with the same number of *breakpoints* and range .................. 84
List of Algorithms

1. IP exploiting binary segmentation and bit-manipulation instructions. . . . . 22
2. LC exploiting binary segmentation. . . . . . . . . . . . . . . . . . . . . . 23
3. Pseudo-code of the IP kernel using Bison-e. . . . . . . . . . . . . . . . . . . . . 28
4. Mix-GEMM pseudo-algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
List of Abbreviations

AAE  average absolute error.

AccMem  accumulator memory.

ADU  address decoding unit.

AI  artificial intelligence.

ATC  Ascend Tensor Compiler.

BLAS  basic linear algebra subprogram.

BLIS  BLAS-like library instantiation software.

BST  binary search tree.

CNN  convolutional neural network.

CPU  central processing unit.

DCU  data conversion unit.

DFT  discrete Fourier transform.

DFU  data filtering unit.

DGEMM  double-precision general matrix multiplication.

DNN  deep neural network.

DSE  design space exploration.

DSU  data selection unit.
**EPI**  European Processor Initiative.

**FP32**  floating-point 32.

**FPGA**  field programmable gate array.

**FU**  functional unit.

**GCD**  greatest common divisor.

**GELU**  Gaussian Error Linear unit.

**GEMM**  general matrix multiplication.

**GPU**  graphics processing unit.

**HPC**  high-performance computing.

**HPEC**  high-performance edge computing.

**IoT**  internet-of-things.

**IP**  inner product.

**ISA**  instruction set architecture.

**KMP**  knuth-morris-pratt.

**LC**  linear convolution.

**LTC**  lookup table cluster.

**LUT**  lookup table.

**MAC**  multiply-accumulate.

**MADD**  multiply-add.

**MAE**  maximum absolute error.

**MCU**  microcontroller unit.

**MSB**  most significant bit.
MSE  mean squared error.
MXU  matrix-multiply unit.
NLP  natural language Processing.
PMU  performance monitoring unit.
PnR  place-and-route.
PTQ  post-training quantization.
PWL  piecewise linear.
QAT  quantization-aware training.
QNN  quantized neural network.
RAM  random access memory.
ReLU Rectified Linear Unit.
RF   register file.
RTL  register transfer level.
RVV  RISC-V vector extension.
SFU  special function unit.
SGD  stochastic gradient descent.
SiLU Sigmoid Linear unit.
SIMD single instruction multiple data.
SoA  state-of-the-art.
SoC  system-on-chip.
TPU  tensor processing unit.
ULP  unit of least precision.
VPU  vector processing unit.


Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2022, p. 56–69.


