Low-Power Neural Network Accelerators with Custom Floating-Point Computation

Dissertation zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.) im Fach Elektrotechnik und Informationstechnik

YARIB NEVAREZ

1. Gutachter:	Prof. Dr. Alberto García-Ortiz
2. Gutachter:	Prof. Dr. X
Eingereicht am:	27.05.2022
Tag des Promotionskolloquiums:	DD.MM.2022

Universität Bremen

Acknowledgment

This work is funded by the *Consejo Nacional de Ciencia y Tecnologia – CONACYT* (the Mexican National Council for Science and Technology).

I would like to thank Prof. Dr. Alberto García-Ortiz, my Ph.D. supervisor, for his invaluable help in this process. He knows how to raise students to the level of independent researchers. I would also like to thank Prof. Dr. X for his guidance and for his time to review this work. Moreover, I would like to thank the members of the graduation committee for their review and suggestions. I would like to thank Mexico and Conacyt for financially supporting my PhD. Thanks to Germany for being my second home during my preparation. Special thanks to Tovalin, Julian Rosales, Fernando De la Torre, Ulises Ponce, Carlos Cruz, and Kai Müller for their inspiration.

I would also like to thank my colleagues Ardalan Najafi, Amir Najafi, Wanli Yu, Yanqiu Huang, Robert Schmidt, Yizhi Chen, Jinming Sun, and Andreas Beering. They are excellent human beans, professional, supportive, and brilliant minds. I also want to thank David Rotermund and Klaus Pawelzik (Institute for Theoretical Physics, University of Bremen) for their collaboration and guidance. A special thanks to all the students I have supervised during my Ph.D. for teaching me so much. I would like to thank Kerstin Janssen and Peter Lutzen for their support in the research department. I thank the University of Bremen, ITEM, and the Studierendenwerk for being virtuous institutions. It would not be possible to reach this point without your kind existence.

I would like to thank Atena Berhang and her family for their sweet and constant support. Likewise, I want to thank my parents for their effort and encouragement to practice virtue and universal love. I want to thank my brothers Kevin and Efren, great sages, my best friends. I also thank all my family members and friends for their support and good wishes during my doctorate.

I would like to thank Marcus Aurelius, Epictetus, Seneca, and Nezahualcóyotl for their mentorship and inspiration.

Yarib Nevarez

The Netherlands, May 2023.

Abstract

The use of Artificial Intelligence (AI) is entering a new era based on the use of ubiquitous connected devices. The sustainability of this transformation requires the adoption of design techniques that reconcile accurate results with cost-effective system architectures. As such, improving the efficiency of AI hardware engines as well as Machine Learning (ML) interoperability must be considered.

In the emerging era of Industry 4.0, ML algorithms yield the power of AI to massively ubiquitous Internet-of-Things (IoT) devices. Applications in this field become smarter and more profitable as the availability of big data gets expanded, driving evolution of many aspects in science, industry, and daily life. However, state-of-the-art ML algorithms, specially Spiking Neural Networks (SNNs) and Convolutional Neural Networks (CNNs), represent elevated computational and energy cost. Therefore, hardware efficiency is one of the major goals to innovate compute engines as they are the machinery of the future.

Energy, performance, and chip-area are the key design concerns in computer systems. Considering the intrinsic error resilience of ML algorithms, paradigms such as approximate computing come to the rescue by offering promising efficiency gains to assist the aforementioned design concerns. Approximation techniques are widely used in ML algorithms at the model-structure as well as at the hardware processing level. However, state-of-the-art methods do not sufficiently address accelerator designs for Artificial Neural Networks (ANNs), in particular with Floating-Point (FP) computation.

To sustain the continuous expansion of ML applications on cost-effective compute devices, approximate computing has the potential to gradually transform from a design alternative to an essential feature. This dissertation focuses on the investigation of design methodologies to exploit the intrinsic error resilience of ML algorithms to optimize high-quality FP inference in low-power embedded systems.

In the field of SNN, this dissertation presents a hardware design methodology for low-power inference of Spike-by-Spike (SbS) neural networks targeting embedded applications. This ML algorithm provides exceptional noise robustness and reduced complexity compared to conventional SNN with Leaky Integrate-and-Fire (LIF) mechanism. However, SbS networks represent a

memory footprint and a computational cost unsuitable for embedded applications. To address this problem, this research exploits the intrinsic error resilience of SbS to improve performance and to reduce hardware complexity. More precisely, it is proposed a vector dot-product module based on approximate computing with configurable quality using hybrid custom FP and logarithmic number representations.

In the field of CNN, this dissertation presents a hardware design methodology for low-power inference targeting sensor analytics applications. In this work, it is proposed the Hybrid-Float6 (HF6) quantization and its dedicated hardware processor. This quantization allows an optimized FP Multiply-Accumulate (MAC) hardware design by reducing the mantissa multiplication to a multiplexer-adder operation. Additionally, this design exploits the intrinsic error tolerance of neural networks to further reduce the hardware architecture with approximation on the FP subnormal number computation. For ML portability, the custom FP representation is wrapped in the standard format, which is automatically extracted by the proposed hardware. The hardware/software architecture is integrated with TensorFlow Lite to demonstrate portability and backward compatibility with industry standard ML frameworks.

The outcome of this dissertation aims to contribute to the rise of a sustainable next generation of energy efficient neural network processors with ML portability and high-accuracy as design requirements.

Contents

1. Introduction

1.1. Preamble

The use of AI/ML is entering a new era with ubiquitous connected devices to enhance intelligence, functionality, and connectivity in our increasingly digital world. However, AI/ML algorithms are computationally and energy intensive, this represents an important challenge for sustainability and realization. Therefore, this digital transformation requires hardware design methodologies that reconcile efficiency with inference quality, and adaptability.

Hardware acceleration for embedded devices is expected to drive further advancements in AI/ML technologies, paving the way for transformative applications in Edge AI, IoT, Federated Learning (FL), Industry 4.0, and smart cities. Interoperability of quantization approaches and model compression techniques are often favored in decentralized machine learning approaches and mission critical domains to ensure the safety and reliability of AI systems. In this line, the development of efficient compute engines remains a crucial enabler for unlocking the full potential of AI/ML technologies.

This section presents the preamble to investigate design methodologies for low-power hardware accelerators of AI/ML algorithms focusing on inference quality, scalability, versatility, and compatibility as design philosophy.

1.1.1. AI/ML in Industry 4.0

AI and ML play a crucial role in the context of Industry 4.0, which is characterized by the integration of digital technologies into manufacturing and industrial processes to create a more connected, intelligent, and automated environment.

Industry 4.0

Since the beginning of industrialization, technological leaps have led to paradigm shifts, now called "industrial revolutions": from mechanization, electrification, and later, digitalization (the so-called 3rd industrial revolution). Based on the advanced digitalization within factories, the combination of Internet technologies and future-oriented technologies in the field of "smart" things (machines and products) seems to result in a new fundamental paradigm shift in industrial production. Emerging from this future expectation, the term "Industry 4.0" was established for an expected "4th industrial revolution" [1].

Internet-of-Things in Industry

To build the emerging environment of Industry 4.0, disruptive technologies are required to handle autonomous communications between all industrial embedded computers throughout the factory and the Internet. Such technologies offer the potential to transform the industry along the entire production chain and stimulate productivity and overall economic growth [2]. These technologies include cloud computing, big data, and specially a new generation of IoT devices fused with Cyber-Physical Systems (CPS), safety-security, augmented reality, ML, and hardware accelerators [3].

Artificial Intelligence in Internet-of-Things

The continuous evolution of AI algorithms and IoT devices has not only made AI the major workload running on these embedded devices, but has transformed AI into the main approach for industrial solutions, especially in the rise of Industry 4.0 [3]. As a result, the term of IoT has also been redefined as AI of Things (AIoT) to emphasize the impact on this technology [4].

There is a clear motivation to run AI/ML algorithms on IoT devices because of [5]:

- **Feasibility of Mission-Critical with Real-Time Processing**. Deploying ML models directly on embedded devices, inference can be performed locally without relying on cloud or remote servers. This enables real-time decision-making and faster response times, which is critical for applications such as autonomous vehicles, industrial automation, and robotics.

- **Privacy and Security of Data**. Processing data locally on embedded devices helps preserve data privacy and security. Instead of transmitting sensitive data to external servers, the data stays inside the device, reducing the risk of data breaches or unauthorized access.

- **Offline Operation Capability and Robustness for Stressed Communication**. Embedded devices often work in environments where network connectivity may be limited or intermittent. Locally running ML models allow the device to continue to function and make intelligent decisions even when disconnected from the network.

1.1.2. Approximation in AI/ML

Based on the error tolerance in ML algorithms, a promising solution is approximate computing. This paradigm has been used in a wide range of applications to increase hardware efficiency [6]. For neural network applications, two main approximation strategies are used, namely network compression and classical approximate computing [7].

Network Compression and Quantization

Researchers focusing on embedded applications started lowering the precision of weights and activation maps to shrink the memory footprint of the large number of parameters representing

ANNs, a method known as network quantization. In this manner, reduced bit precision causes a small accuracy loss [8, 9, 10, 11]. In addition to quantization, network pruning reduces the model size by removing structural portions of the parameters and its associated computations [12, 13]. This method has been identified as an effective technique to improve the efficiency of CNN for applications with limited computational and energy budget [14, 15, 16]. These techniques leverage the intrinsic error-tolerance of neural networks, as well as their ability to recover from accuracy degradation while training.

Error Tolerance in AI/ML Algorithms

An algorithm can be regarded as error-tolerant or error-resilient when it provides a result with the required accuracy while utilizing processing components with a certain degree of inaccuracy. There are several reasons why an algorithm/application is tolerant of errors as discussed in [17]. These include noisy or redundant data of the algorithm, approximate or probabilistic computations within the algorithm, and a range of acceptable outcomes. This is the case of AI/ML models.

Approximate Computing

Approximate computing is a design paradigm that is able to tradeoff computation quality (e.g., accuracy) and computational efficiency (e.g., in run-time, chip-area, and/or energy) by exploiting the error resilience of applications/algorithms [18, 19]. Data redundancy of neural networks incorporate a certain degree of resilience against random external and internal perturbations; for instance, noisy inputs and random hardware errors. This property can be exploited in a cross-layer resilience approach [20]: by leveraging error tolerance at algorithmic-level, it can be allowed a certain degree of inaccuracies at the computing-level. This approach consists of designing processing elements that approximate their computation by employing cleverly modified algorithmic logic units [6].

Approximate computing techniques allow substantial enhancement in processing efficiency with moderated accuracy degradation. Some research papers have shown the feasibility of applying approximate computing to the inference stage of neural networks [21, 6, 22, 23, 24, 25]. Such techniques usually demonstrated small inference accuracy degradation, but significant enhancement in computational performance, chip-area, and energy consumption. Hence, by taking advantage of the intrinsic error-tolerance of neural networks, approximate computing is positioned as a promising approach for AI/ML computation on resource-limited devices.

1.2. Problem Statement

A fundamental problem for the rise of AI in Industry 4.0 is the fact that ML models, particularly CNNs and ANNs, are highly computational and data intensive. This brings significant challenges across the spectrum of computing hardware, specially in the scope of embedded systems [26]. The most deployed models and also some of the most computationally and energy expensive are for computer vision using CNNs. Compared to the conventional image processing methods, the accuracy of CNN has improved significantly that by 2015, a human can no longer beat a computer in image classification [5]. The early development of CNNs before 2016 mainly focused on accuracy improvement without considering computational costs. While accuracy of deep CNN for image classification improved 24% between 2012 and 2016, the demand on hardware resources increased more than 10×. Starting from 2017, significant attention was paid to improve hardware efficiency in terms of compute power, memory bandwidth, and power consumption, while maintaining accuracy at a similar level to human perception [26].

1.2.1. Power Dissipation

Consequently, the recent breakthroughs in AI/ML applications have brought significant advancements in neural network processors [27]. To bring the inference speed to an acceptable level, Application-Specific Integrated Circuit (ASIC) with Neural Processing Unit (NPU) are becoming ubiquitous in both embedded and general purpose computing. NPUs perform several tera operations per second in a confined area, as a consequence, they become subject to elevated on-chip power densities that rapidly result in excessive on-chip temperatures during operation [28]. Subsequently, the elevated power supply, physical dimensions, heat sink and air cooling requirements demand a balance between the benefits of ML against its environmental and financial costs. This outcome is expected on parallel computing techniques, yet unsustainable for resource-constrained devices. Therefore, radical changes to conventional computing are required in order to sustain and improve performance while satisfying energy and temperature constraints [18].

1.2.2. Aggressive Quantization

Furthermore, reducing the compute hardware with aggressive quantization such as binary [8], ternary [29], and mixed precision (2-bit activations and ternary weights) [30] typically incur significant accuracy degradation for very low precisions, especially for complex problems [31], such as: semantic segmentation, machine translation, language generation, playing agents, image/music generation, and medical applications.

While aggressive quantization can be beneficial for resource-constrained environments and non-critical applications, careful consideration and a more conservative approach are essential for ensuring the safety and reliability of AI/ML systems in mission-critical domains. Quantization techniques must be chosen wisely, keeping in mind the specific requirements and constraints of each application.

1.2.3. Interoperability

Aggressive quantization might not be supported by all hardware platforms. Custom hardware accelerators may have limitations on the precision they can handle effectively, limiting the compatibility and portability of aggressively quantized models. Aggressively quantized models may not be compatible with all frameworks, libraries, or AI platforms, limiting their interoperability and portability across different environments. In real-world deployment scenarios, there may be constraints and requirements that make aggressive quantization impractical, especially when high accuracy, compatibility, portability, and interoperability are necessary.

1.3. Working Hypothesis

Despite its benefits, quantization may lead to a trade-off between model accuracy and hardware efficiency. Aggressive quantization can cause a loss of model accuracy due to information loss from reduced precision. Therefore, finding the right balance between quantization levels and model accuracy becomes a critical challenge in developing efficient hardware accelerators for ML applications.

Introducing custom FP computation, with standard FP activation maps and significantly reduced FP representation for trainable parameters, will lead to improved hardware efficiency in ML accelerators, Quality of Result (QoR), and interoperability.

The hypothesis is built on the following key aspects:

- **Reduced Precision Requirement for Weights and Biases**. Neural network models can tolerate lower precision for weights and biases without a substantial decrease in model accuracy [32]. By significantly reducing the bit-width of weights and biases, the memory requirements and computational complexity of the accelerator can be significantly reduced.
- **FP Activation Maps**. Retaining higher precision in activation maps prevents significant accuracy degradation during inference.

- **Hardware Efficiency**. The mixed precision approach, combining standard FP activation maps and reduced bit-width for weights and biases, is expected to improve the hardware efficiency of ML accelerators [32]. The reduced bit-width for weights and biases will enable faster arithmetic operations and reduce memory bandwidth requirements, leading to more efficient processing in hardware accelerators.

- **Compatibility and Interoperability**. The hardware accelerator should be designed to efficiently handle the combination of different precision levels for weights, biases, and activation maps. Downgrading FP from higher precision to lower precision involves rounding or discarding the extra bits. Upgrading FP from lower precision to higher precision involves wrapping values in wider bit representations.

- **Quantization-Aware Training (QAT)**. This helps the model adapt to the mixed precision representation, ensuring that the reduced precision for weights and biases does not cause significant accuracy loss.

The dissertation would contribute to the field of hardware-efficient ML accelerators by exploring the trade-offs between FP precision levels for different hardware components and finding the right balance to achieve improved efficiency without compromising accuracy.

1.4. Research Objective

Develop novel design methodologies for low-power neural network accelerators with custom FP computation. This research objective encompasses several key aspects that are crucial for advancing the field of low-power neural network accelerators with custom FP computation:

- **Custom FP Representation**. Investigate novel custom FP representation optimized for neural network computations. This involves exploring different bit-widths, exponent formats, or non-standard FP representations tailored specifically for neural network workloads.

- **Low-Power Design Techniques**. Investigate low-power design techniques at various levels, including logic-level optimizations and architectural-level approaches to minimize power consumption.

- **Custom FP Arithmetic**. Investigate design and implementation of custom arithmetic units that efficiently perform the proposed custom FP computations. Special attention shall be given to reducing energy consumption during arithmetic operations.

- **Neural Network-Specific Optimization**. Investigate accelerator architectures specifically for neural network workloads in resource-constrained applications. Low-power techniques like pipe-lining, data quantization, and hardware-specific optimizations are explored to ensure efficiency and QoR.
- **Quantization and Precision Analysis**. Investigate and present comprehensive analysis of the effects of quantization and reduced precision on model accuracy. QAT and dynamic precision techniques to maintain or enhance accuracy with lower precision arithmetic.
- **Scalability and Versatility**. Investigate scalability techniques to handle a wide range of neural network models and sizes.
- **Comparison with Other Approaches**. Include comparisons with other low-power neural network accelerators to demonstrate the strengths and uniqueness of the proposed custom FP computation techniques.
- **Evaluation and Benchmarking**. Present evaluation and benchmarking against existing state-of-the-art, demonstrate advantages in terms of power efficiency and performance.
- **Real-World Applications**. The objective involves showcasing the practical applicability in real-world low-power and resource-constrained applications, such as sensor analytics.
- **Future Directions**. This dissertation shall provide valuable insights into future research directions for further optimizing low-power neural network accelerators with custom FP computation, potential integration with edge devices, and their impact on the broader field of machine learning hardware.

Overall, the research objective should address significant challenges in designing efficient hardware accelerators for neural networks, with a focus on reducing power consumption through custom FP computation and advancing the state-of-the-art in low-power neural network accelerators.

1.5. Scope

The scope of this research revolves around enabling energy-efficient and high-quality inference of SbS and CNN models on resource-constrained applications. This scope presents hardware design methodologies specifically tailored for SbS and CNN models on System-on-Chip (SoC) devices with limited computational resources, memory, and power constraints.

- **Spike-by-Spike Neural Networks**. SNNs offer advantageous robustness and the potential to achieve a power efficiency closer to that of the human brain. SNNs operate reliably using stochastic elements that are inherently non-reliable mechanisms [33]. This provides superior resistance against adversary attacks [34, 35]. Beside robustness, SNNs have further advantages like the possibility of a more efficient asynchronous parallelization and higher energy efficiency than conventional ANNs.

  The Spike-by-Spike model is on the less realistic side of the SNN scale of biological realism [36, 34]. Consequently, the hardware complexity of SbS network implementations is greatly reduced [37]. In spite of this, SbS still uses stochastic spikes as a means of transmitting information between populations of neurons and thus retains the advantageous robustness of SNNs. A significant research effort has been done in SNN accelerators, see e.g. [38, 7, 39, 40, 41, 42].

  However, hardware accelerators that focus on SbS have only been partially investigated so far [37]. Enhanced SbS accelerators will have a double impact. From scientific and application point of view, they will facilitate fundamental research for neuroscience [34, 43, 44] and contribute to the deployment of robust neural networks in small embedded systems [45].

- **Convolutional Neural Networks**. CNNs represent the essential building blocks in 2D pattern analytics. Sensor-based applications such as mechanical fault detection [46, 47], structural health monitoring [48], Human Activity Recognition (HAR) [49], hazardous gas detection [50] have been powered by CNN models in industry and academia. CNN models provide advantages such as local dependency, scale invariance, and noise resilience in analytics [22]. However, these models are computationally intensive and power-hungry. This is particularly challenging for low-power embedded applications, specially in the field of IoT. As a result, numerous commercial ASIC and Field-Programmable Gate Array (FPGA) accelerators have been proposed, these are targeting both High Performance Computing (HPC) for data-centers and embedded systems applications.

  However, most accelerators have been implemented to target mid- to high-range FPGAs for computationally intensive CNN models such as AlexNet, VGG-16, and ResNet-18. The main drawbacks of these implementations are power supply demands, physical dimensions, heat sink requirements, air cooling, and a resulting high price. In some cases, these implementations are not feasible for ubiquitous low-power/resource-constrained applications. Furthermore, reducing the compute hardware with aggressive quantization such as binary [8], ternary [29], and mixed precision (2-bit activations and ternary weights)

[30] typically incur significant accuracy degradation for very low precisions, especially for complex problems [31].

1.6. Contributions

This research produces hardware design methodologies for low-power hardware accelerators with custom FP computation that reconcile efficiency with inference quality, and compatibility. This work is demonstrated on SbS and CNN hardware accelerators on resource-constrained SoC FPGAs:

1.6.1. Accelerating Spike-by-Spike Neural Networks with Hybrid 8-bit Floating-Point and 4-bit Logarithmic Computation

1. An optimized FP MAC design with hybrid precision is presented. This design utilizes the IEEE 754 single-precision FP for feature maps and custom FP representation for weights. The custom FP precision does not implement a sign bit and has reduced exponent and mantissa bit widths. This design accumulates values in a denormalized manner and produces IEEE 754 single-precision results.This design not only upholds QoR but also enhances efficiency through reduced latency, diminished power dissipation, minimized hardware resources, and a smaller memory footprint.

2. A design exploration is presented focusing on input weight FP representation of 8 and 4-bit. Evaluation is reported across various facets such as run-time, accuracy degradation, hardware resource utilization, and power consumption. An embedded system architecture is presented to assess the performance of the custom FP MAC. In evaluations using a generative model based on Non-Negative Matrix Factorization (NNMF) (SbS model), a notable latency improvement of 20.5× was measured in comparison to an embedded Central Processing Unit (CPU) (ARM Cortex-A9 operating at 666 MHz). Notably, in a handwritten digit recognition task, an accuracy degradation of less than 0.5% was measured.

3. A noise tolerance plot is proposed as a quality monitor, which is intended to serve as an intuitive visual model. This model offers insights into the accuracy degradation of NNMF models (SbS networks) when subjected to custom FP computation.

4. The design presented for custom FP MAC is considered adaptable for use as a building block in other error-resilient applications, such as image/video processing.

1.6.2. Accelerating Convolutional Neural Networks with Hybrid 6-bit Floating-Point Computation

1. A 6-bit FP representation for weights is presented as HF6 quantization. An optimized hardware MAC is proposed, wherein mantissa multiplication is reduced to a multiplexer-adder operation. The intrinsic error tolerance of ANN is harnessed in this method to further refine the hardware design via approximation. To maintain model accuracy, a QAT approach is introduced. Notably, in certain instances, this method elevates accuracy, attributed to the regularization effect.

2. A custom hardware/software co-design framework for CNN sensor analytics applications has been developed, targeting resource-constrained SoC FPGAs. This architecture incorporates TensorFlow Lite.

3. A customizable Tensor Processor (TP) is presented as a proof of concept demonstration with HF6. In this design, accelerations of up to 48X are achieved by the *Conv2D* tensor operation in comparison to the CPU running TensorFlow Lite. This enhancement is realized without accuracy degradation employing QAT.

4. The potential of this approach is demonstrated with a CNN-regression model for anomaly localization in Structural Health Monitoring (SHM) based on Acoustic Emission (AE). A hardware design exploration is addressed evaluating accuracy, compute performance, hardware resource utilization, and energy consumption.

1.7. Publications

The outcome of this dissertation, including the collaborative works with our research partners is a list of publications including [45, 51, 52]. In the following, a complete list of the related publications are itemized.

Journal Articles

1. **Yarib Nevarez**, David Rotermund, Klaus R Pawelzik, and Alberto Garcia-Ortiz, "Accelerating Spike-by-Spike Neural Networks on FPGA With Hybrid Custom Floating-Point and Logarithmic Dot-Product Approximation," IEEE Access, vol. 9, pp. 80603–80620, May 2021, doi: 10.1109/ACCESS.2021.3085216.

2. **Yarib Nevarez**, Andreas Beering, Amir Najafi, Ardalan Najafi, Wanli Yu, Yizhi Chen, Karl-Ludwig Krieger, and Alberto Garcia-Ortiz, "CNN Sensor Analytics With Hybrid-Float6 Quantization on Low-Power Embedded FPGAs," IEEE Access, vol. 11, pp. 4852–4868, January 2023, doi: 10.1109/ACCESS.2023.3235866.

Conference Proceedings

3. **Yarib Nevarez**, Alberto Garcia-Ortiz, David Rotermund, and Klaus R Pawelzik, "Accelerator framework of spike-by-spike neural networks for inference and incremental learning in embedded systems," 2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST), Bremen, 2020, pp. 1–5, doi: 10.1109/MOCAST49295.2020.9200288.

4. Wanli Yu, Ardalan Najafi, **Yarib Nevarez**, Yanqiu Huang and Alberto Garcia-Ortiz, "TAAC: Task Allocation Meets Approximate Computing for Internet of Things," 2020 IEEE International Symposium on Circuits and Systems (ISCAS), Sevilla, 2020, pp. 1-5, doi: 10.1109/ISCAS45731.2020.9180895.

5. Amir Najafi, Ardalan Najafi, **Yarib Nevarez** and Alberto Garcia-Ortiz, "Learning-Based On-Chip Parallel Interconnect Delay Estimation," 2022 11th International Conference on Modern Circuits and Systems Technologies (MOCAST), Bremen, 2022, pp. 1–5, doi: 10.1109/MOCAST49295.2020.9200288.

6. Yizhi Chen, **Yarib Nevarez**, Zhonghai Lu, and Alberto Garcia-Ortiz, "Accelerating Non-Negative Matrix Factorization on Embedded FPGA with Hybrid Logarithmic Dot-Product Approximation," 2022 IEEE 15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Malaysia, 2022, pp. 239–246, doi: 10.1109/MCSoC57363.2022.00070.

7. Ardalan Najafi, Wanli Yu, **Yarib Nevarez**, Amir Najafi, Andreas Beering, Karl-Ludwig Krieger, and Alberto Garcia-Ortiz, "Acoustic Emission Source Localization using Approximate Discrete Wavelet Transform," 2023 12th International Conference on Modern Circuits and Systems Technologies (MOCAST), Bremen, 2023, pp. X–X, doi: XX.XXXX/MOCASTXXXXX.XXXX.XXXXXXX.

1.8. Dissertation Outline

This dissertation is organized in three main parts: an introduction, where the state of the art and related background are stated; a central core, where the proposed design methodologies and validation are presented; and a final part with the conclusion. More precisely:

I **Introduction**: Chapter 2 introduces the background related to SbS, CNN, and FP number representation.

II **Core**: the proposed hardware design methodologies for SbS and CNN accelerators are presented in Chapter 3 and Chapter 4, respectively.

III **Conclusions**: the final conclusions are presented in Chapter 5.

2. Background and Related Work

2.1. Introduction

At the heart of the AI revolution lie neural networks, which are computational models inspired by the human brain. These algorithms have demonstrated an unprecedented ability to discern patterns, extract insights, and make predictions from vast amounts of data, often matching or even surpassing human capabilities in certain domains [53, 54, 55, 56, 57].

However, the explosive growth and complexity of neural networks have also given rise to significant computational challenges. Deep neural networks, characterized by their multi-layered architectures, can contain millions, if not billions, of parameters. Training and deploying these models demand elevated amounts of computational power. Traditional CPU, while versatile, are not inherently optimized for the parallel and matrix-based computations that neural networks demand. This computational bottleneck not only impacts the speed and efficiency of neural network operations but also their energy consumption. This is a critical concern in our increasingly mobile and interconnected world.

Neural networks accelerators such as Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), specialized ASIC, and FPGA-based implementations have emerged as important players, providing the needed horsepower to drive neural computations swiftly and efficiently. As the quest for speed and efficiency continues, there is a growing interest in further refining these accelerators, particularly through custom numerical representations such as custom floating-point computation. This avenue promises a harmonious blend of performance and power efficiency, potentially announcing a new era for neural network engines.

This chapter delves into the world of neural networks, their indelible mark on modern computation, and the imperatives driving the development of dedicated hardware accelerators. Through this exploration, it is provided the background stage into low-power neural network accelerators leveraging custom floating-point computation a frontier at the nexus of AI potential and practicality.

2.2. Neural Networks

The concept of neural networks traces its roots back to the 1950s with the introduction of the perceptron by Frank Rosenblatt. The perceptron, a single-layer feedforward neural network, was among the first models capable of binary classifications [58]. However, the limitations of perceptrons, including their inability to solve non-linearly separable functions, led to diminished interest in neural networks until the backpropagation algorithm emerged in the 1980s [59].

2.2.1. Architecture

Neural networks are computational models designed to extract patterns, interpret data, and approximate complex functions. Their architecture comprises interconnected nodes (neurons) organized into layers. Each connection possesses a weight value, which is adjusted during training. The primary components include:

Layers

- **Input Layer**: Receives data. Given input data vector $\mathbf{x}$ of dimension d, the number of neurons is d.

$$\mathbf{x} = [x_1, x_2, ..., x_d]^T$$

- **Hidden Layer(s)**: Transform the input using weighted connections. The output h of a neuron in a hidden layer is:

$$h = f(\mathbf{w}^\top \cdot \mathbf{x} + b)$$

  where $\mathbf{w}$ is the weights vector, b is a bias, and f is an activation function.

- **Output Layer**: Produces the predictions. The architecture depends on the task (e.g., regression, classification).

Weights and Bias

For each neuron, input data is transformed using weights and biases, adjusted during training. The weighted sum for a neuron is:

$$z = \mathbf{w}^\top \cdot \mathbf{x} + b$$

Activation Functions

Introduce non-linearities, enabling neural networks to capture complex relationships. Common functions include:

- **Sigmoid**: The sigmoid function, denoted as $\sigma(z)$, is especially used in binary decision tasks. Mathematically, the sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

  Here, z represents the input to the function. The function outputs a value between 0 and 1, making it especially useful for models where the output is a probability.

The curve of the sigmoid function is S-shaped or sigmoidal. One of its properties is that its derivative (used in the backpropagation step of training neural networks) can be expressed in terms of the sigmoid function itself:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

However, the sigmoid function is not without drawbacks. For very large or very small values of z, the function becomes saturated, leading to small gradients and, consequently, slow convergence during training [60]. This phenomenon is often referred to as the "vanishing gradient" problem.

- **Tanh**: The hyperbolic tangent function, denoted as $\tanh(z)$, serves as an activation function in many neural network architectures. It scales and shifts the output of the sigmoid function to produce outputs in the range $[-1, 1]$. The mathematical expression for tanh is:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Alternatively, it can be expressed in terms of the sigmoid function, $\sigma(z)$, as:

$$\tanh(z) = 2\sigma(2z) - 1$$

The derivative of tanh, which is used during the backpropagation phase of neural network training, is given by:

$$\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z)$$

Compared to the sigmoid function, tanh is often preferred because its outputs are zero-centered, making it less likely to get stuck during training. However, it still suffers from the vanishing gradient problem for very large or very small values of z.

- **ReLU**: The Rectified Linear Unit, commonly referred to as ReLU, has become one of the default activation functions, particularly for deep learning architectures. Mathematically, the ReLU function is defined as:

$$f(z) = \max(0, z)$$

In essence, the function returns z if z is greater than or equal to zero, and returns zero otherwise. This can be visualized as a linear function that will output the input directly if it is positive; otherwise, it will output zero.

The gradient of the ReLU function is binary:

$$f'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

One noted benefit of the ReLU function is its computational efficiency, given that it only requires a simple thresholding at zero. This allows models to train faster and requires less computational resources compared to other activation functions like sigmoid or tanh.

However, a potential drawback of the ReLU function is that units can sometimes get "stuck" during training and cease updating, leading to what's known as "dying ReLUs." The dying ReLU phenomenon can be viewed as a specific type of the vanishing gradient problem, where ReLU neurons become non-responsive and consistently output a value of zero, regardless of the input they receive. This is due to the fact that for inputs less than 0, the gradient is 0, which can cause weights to not update during backpropagation. To counteract this, variants like Leaky ReLU [61] and Parametric ReLU [62] have been proposed.

- **Softmax**: In a neural network for multiclass classification tasks, the softmax function is a common choice for the activation function in the output layer. Given an input vector $\mathbf{z}$ of length K, representing the raw output (logits) of the K nodes in the output layer, the softmax function transforms these logits into a probability distribution over K classes. For each component i (where $i = 1, 2, \ldots, K$), the softmax function $S(\mathbf{z})$ is computed as:

$$S(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$$

where $S(\mathbf{z})_i$ represents the i-th component of the output vector $S(\mathbf{z})$, exp denotes the exponential function, and z_i is the i-th component of the input vector $\mathbf{z}$. After applying the softmax function, each component of $S(\mathbf{z})$ is in the interval $(0, 1)$, and the components sum to 1, allowing them to be interpreted as probabilities associated with each of the K classes. This makes the softmax function particularly useful for producing the final output in a neural network designed for classification tasks, as it ensures that the outputs are normalized and can be interpreted as class probabilities.

2.2.2. Training Process

Neural networks learn by adjusting weights and biases in response to training data. The goal is to minimize the difference between predictions and target values (the loss), often optimized using gradient-based methods.

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an iterative optimization algorithm used to minimize an objective function that is defined as a sum of differentiable functions. This is particularly well-suited for problems with a large number of training samples [63].

Given an objective function $J(\theta)$ which we aim to minimize, the objective is often defined as:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} J_i(\theta)$$

where $J_i(\theta)$ is the loss associated with the i^{th} training example and θ represents the parameters of the model.

The basic update rule for SGD is:

$$\theta \leftarrow \theta - \eta \nabla J_i(\theta)$$

where:

- η is the learning rate, a positive scalar determining the size of steps in the parameter space.
- $\nabla J_i(\theta)$ is the gradient of the loss with respect to the parameters for the i^{th} training example.

In each iteration, a single training example (x_i, y_i) is picked randomly, and the model parameters are updated with the gradient of the loss $J_i(\theta)$ with respect to that single example.

The iterative nature and use of only a single training example at each step make SGD computationally more efficient compared to batch gradient descent, especially for large datasets [63]. However, due to its stochastic nature, the trajectory of the parameters through the parameter space can be noisy, leading to a non-stable convergence to the minimum [64]. Variants and improvements, like momentum [65] or adaptive learning rates [66], have been introduced to combat this instability and accelerate convergence.

2.2.3. Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is a type of feedforward artificial neural network, consisting of multiple layers of interconnected neurons [67].

Key Components

- An **input layer** that receives the data. The number of neurons in this layer corresponds to the dimensionality of the input data.
- One or more **hidden layers** that transform the input data. Each neuron in a hidden layer computes a weighted sum of its inputs, adds a bias term, and then applies an activation function.
- An **output layer** that provides the final prediction or classification results. The number of neurons in the output layer and their activation functions are tailored to the specific task.

Mathematically, the output o_j of the j^{th} neuron in any layer can be defined as:

$$o_j = f\left(\sum_{i=1}^{N} w_{ij}x_i + b_j\right)$$

where:

- x_i is the output of the i^{th} neuron from the previous layer.
- w_{ij} is the weight associated with the connection between the i^{th} neuron from the previous layer and the j^{th} neuron of the current layer.
- b_j is the bias term for the j^{th} neuron.
- f is the activation function, which introduces non-linearity into the network.

The weights and biases of an MLP are adjusted during the training phase.

2.2.4. Convolutional Neural Networks

CNNs represent a specialized architecture in the deep learning domain, predominantly optimized for image and video processing tasks. Drawing inspiration from the human visual cortex structure and function, CNNs are adept at automatically and adaptively discerning spatial hierarchies and patterns inherent in visual data [68].

Key Components

- **Input Layer**: Responsible for ingesting raw pixel values from the image. The size is typically dictated by the resolution and depth (e.g., RGB channels) of the input.
- **Convolutional Layer**: At its core, the convolution operation involves sliding a filter over the input matrix to produce feature maps. Each filter aims to detect specific features, like edges or textures, in the input.
- **Activation Function**: Following the convolution operation, it is common to introduce non-linearity into the system. The ReLU is popularly employed in CNNs.
- **Pooling Layer**: To reduce the spatial dimensions and computational load, the pooling layer down-samples feature maps. Max pooling is a widely adopted strategy:

  $$Z = \max(W \times X) \tag{2.1}$$

  where W is the window applied to the input matrix X.
- **Fully Connected Layer**: This layer sees each neuron connected to every activation from the previous layer, effectively acting as a standard multi-layer perceptron. It linearizes the features extracted from preceding layers and approximates a function, which commonly is an image classification or regression task.
- **Output Layer**: Generates the final class scores, typically in a probabilistic form via a softmax function.

A relevant characteristic of CNNs is weight sharing, this substantially reduces the number of parameters, thus diminishing the risk of overfitting. Through their capacity to hierarchically discern two-dimensional patterns, CNNs have been relevant in furthering advancements in a variety of fields, spanning from sensor analytics to medical image analysis and autonomous vehicle vision systems.

Conv2D Tensor Operation

A convolutional layer aims to learn and extract feature representations from a given input. Each unit of a feature map is connected to a region of neighboring units on the input maps (from the previous layer). This neighborhood in the previous layer is known as the receptive field of such unit. A new feature map can be obtained by first convolving the input maps with a learned kernel and then applying a nonlinear elementwise activation function to the convolved results.

All spatial locations on the input maps share a kernel to generate a feature map. All feature maps are obtained by convolving several different kernels [69].

The 2D convolution process is performed by the *Conv2D* tensor operation, described in **Eq.** (2.2), where W is the convolution kernels (known as filters), b is the bias vector for the output feature maps, and h is the input tensor containing the feature maps [67]. $K \times L \times M$ is the receptive field size, $K \times L$ is the convolution kernel, and M is the number of input channels/feature maps. Mathematically, the *Conv2D* operator is defined as:

$$Conv2D\,(W, b, h)_{i,j,o} = \sum_{k,l,m}^{K,L,M} h_{(i+k,j+l,m)} W_{(o,k,l,m)} + b_o \tag{2.2}$$

Computational Cost of a Convolution Layer

The computational complexity of a convolution layer primarily depends on the spatial dimensions of the input and the kernel, the number of input and output channels, and the stride (slide step) with which the kernel is applied. For clarity, a list of definitions and concepts is presented:

Definitions

- W_i: Width of the input feature map.
- H_i: Height of the input feature map.
- D_i: Depth (number of channels) of the input feature map.
- W_k: Width of the kernel.
- H_k: Height of the kernel.
- D_k: Depth of the kernel. Typically, $D_k = D_i$.
- N_o: Number of output feature maps (number of kernels in the layer).
- S: Stride of the convolution.

Computations Per Output Element For each kernel position on the input feature map, the number of multiply and accumulate operations is defined by:

$$2 \times W_k \times H_k \times D_k \tag{2.3}$$

Output Dimensions Given the stride S, the dimensions of the output feature map are:

$$W_o = \frac{W_i - W_k}{S} + 1 \tag{2.4}$$

$$H_o = \frac{H_i - H_k}{S} + 1 \tag{2.5}$$

Total Computations For each output feature map, the total operations are:

$$W_o \times H_o \times 2 \times W_k \times H_k \times D_k \tag{2.6}$$

Given N_o output feature maps, the entire convolution layer's computational cost becomes:

$$N_o \times W_o \times H_o \times 2 \times W_k \times H_k \times D_k \tag{2.7}$$

This represents the number of multiply-accumulate operations in a convolution layer. Other considerations might include biases and employed activation functions, but the above calculation primarily signifies the computational burden of the convolution operation.

Error Tolerance in Convolution Layers

Deep neural networks, especially CNNs, have exhibited an advantageous property: they possess a significant degree of robustness to various perturbations in their computations, including reduced-precision arithmetic and the introduction of noise. This error tolerance characteristic of convolution layers has paved the way for various optimization techniques aiming to reduce the computational and storage overhead without sacrificing too much in performance [9].

Sources of Error Errors in convolution layers can arise from various sources:

- **Quantization:** Converting FP precision weights and activations to a lower bit-width representation.
- **Pruning:** Setting certain weights to zero to reduce the total number of weights.
- **Approximate Computing:** Techniques that purposefully introduce computational errors by simplifying compute hardware to improve power efficiency, area, and speed.

Error Compensating Mechanisms There are several hypotheses and mechanisms that explain the error tolerance:

- **Overparameterization:** Many deep models have more parameters than necessary for the task. This data redundancy can help the network adapt to small errors.
- **Re-training:** After introducing errors (like in quantization), the network can be fine-tuned to recover some of the lost performance.
- **Regularization Effect:** Some error introduction techniques, like quantization, can act as a form of regularization, potentially helping to prevent overfitting.

Exploiting Error Tolerance for Optimization Leveraging the error resilience of convolution layers can lead to several benefits:

- **Reduced Precision:** Weights and activations can be represented with fewer bits, leading to a reduced memory footprint and computation savings.
- **Energy Efficiency:** Approximate computing techniques can yield significant power savings.
- **Faster Computations:** Reduced precision arithmetic can be faster and allows for parallelism.

Understanding and harnessing the error tolerance properties of convolution layers present opportunities for designing more efficient and compact neural network implementations, especially vital for low-power devices and real-time applications. While errors can be introduced to an extent, it remains crucial to ensure that the network accuracy does not degrade beyond acceptable levels.

2.2.5. Spiking Neural Networks

SNNs offer an alternative paradigm in the neural computation landscape. Diverging from traditional ANNs, they emulate the temporal dynamics of biological neurons. Each neuron in an SNN accumulates input until a threshold is reached, upon which it emits a spike and resets. This behavior can be mathematically described by models such as the LIF [70]:

$$\tau_m \frac{du(t)}{dt} = -u(t) + RI(t) \tag{2.8}$$

where $u(t)$ is the neuron membrane potential, τ_m its time constant, R its resistance, and $I(t)$ the input current. A spike is emitted when $u(t)$ surpasses a threshold u_{th}, followed by a reset.

Potential advantages of SNNs include energy efficiency, especially on neuromorphic hardware, and competence for processing temporal data [71]. However, training SNNs presents challenges due to the non-differentiable nature of spikes [72]. Techniques range from gradient approximations for backpropagation [73], surrogate gradient methods [72], and unsupervised methods such as Spike-Timing-Dependent Plasticity (STDP) [74].

Spike-by-Spike Neural Networks

SbS is a spiking neural network based on a generative probabilistic model. It iteratively finds an estimate of its input probability distribution $p(s)$ (i.e. the probability of input node s to stochastically send a spike) by its latent variables via $r(s) = \sum_i h(i)W(s|i)$, where $\vec{h}$ is an inference population composed of a group of neurons that compete with each other. An Inference Population (IP) sees only the spikes s_t (i.e. the index identifying the input neuron s which generated that spike at time t produced by its input neurons, not the underlying input probability distribution $p(s)$ itself. By counting the spikes arriving at a group of SbS neurons, $p(s)$ is estimated by $\hat{p}(s) = 1/T \sum_t \delta_{s,s^t}$ after T spikes have been observed in total. The goal is to generate an internal representation $r(s)$ from the string of incoming spikes s_t such that the negative logarithm of the likelihood $L = C - \sum_\mu \sum_s \hat{p}_\mu(s) log\left(r_\mu(s)\right)$ is minimized. C is a constant which is independent of the internal representation $r_\mu(s)$ and μ denotes one input pattern from an ensemble of input patterns. Applying a multiplicative gradient descent method on L, an algorithm for iteratively updating $h_\mu(i)$ with every observed input spike s_t could be derived [34]:

$$h_\mu^{new}(i) = \frac{1}{1+\epsilon}\left(h_\mu(i) + \epsilon \frac{h_\mu(i)W(s_t|i)}{\sum_j h_\mu(j)W(s_t|j)}\right) \tag{2.9}$$

where ϵ is a parameter that also controls the strength of sparseness of the distribution of latent variables $h_\mu(i)$. Furthermore, L can also be used to derive online and batch learning rules for optimizing the weights $W(s|i)$. The interested reader is referred to [34] for a more detailed exposition.

From a practical point of view, SbS provides a mechanism to obtain a sparse representation of input patterns. Given a set of training samples $\{x_\eta\}$, it learns weights (W), that allow to express the input patterns as a linear sparse non-negative combination of features. During inference, it provides a mechanism for expressing each test input x_μ as $x_\mu \approx W\, h_\mu$ where all entries are non-negative.

The inference procedure consists in generating indices s_t distributed according to a categorical distribution of the input pattern $s_t \sim \text{Categorical}(x_\mu(0), x_\mu(1), .., x_\mu(N-1))$. Starting with a random h and executing iteratively **Eq.** (2.9) the SbS algorithms finds h_μ. The fundamental concept of SbS can be extended from vector to matrix inputs. In this case, the linear operation $W\, h_\mu$ can be replaced by a convolution to obtain a convolutional SbS layer. A detailed description of the SbS algorithm is presented in the Appendix A

Basic Network Overview SbS network models can be constructed in sequential layered structures [36]. Each layer consists of many IPs (represented by $\vec{h}$), while the communication between them is organized by a low bandwidth signal – the spikes.

The SbS layer update is summarized in Algorithm 1. This is an iterative algorithm, where the number of spikes are denoted as (N_{Spk}), which is the number of iterations. As a generative model, each iteration updates the internal representation (H) based on the input spikes (S_t^{in}). A basic SbS network architecture for handwritten digit classification (MNIST) is shown in **Fig.** 2.1 and **Tab.** 2.1. Each IP is an independent computational entity, this allows to design specialized hardware architectures that can be massively parallelized (see **Fig.** 2.2).

Algorithm 1: SbS layer update.

for $t \leftarrow 0$ **to** $N_{Spk} - 1$ **do**
  for $x \leftarrow 0, y \leftarrow 0$ **to** $N_X - 1, N_Y - 1$ **do**
    $S_t^{out}(x, y) \sim Categorical(H(x, y, :))$
    for $\Delta_X \leftarrow 0, \Delta_Y \leftarrow 0$ **to** $K_X - 1, K_Y - 1$ **do**
      $spk \leftarrow S_t^{in}(x + \Delta_X, y + \Delta_Y)$
      for $i \leftarrow 0$ **to** $N_H - 1$ **do**
        $\Delta h(i) \leftarrow H(x, y, i) \cdot W(\Delta_X, \Delta_Y, spk, i)$
        $r \leftarrow r + \Delta h(i)$
      end for
      for $i \leftarrow 0$ **to** $N_H - 1$ **do**
        $H^{new}(x, y, i) \leftarrow \frac{1}{1+\epsilon}\left(H(x, y, i) + \frac{\epsilon}{r}\Delta h(i)\right)$
      end for
    end for
  end for
end for

Computational Cost The number of MAC operations required for inference of an SbS layer is defined by $NOPS_{MAC} = N_{Spk} N_X N_Y K_X K_Y (3N_H + 2)$, where N_{Spk} is the number of spikes (iterations), $N_X N_Y$ is the size of the layer, $K_X K_Y$ is the size of the kernel for convolution/pooling, and N_H is the length of $\vec{h}$. The computational cost of SbS network models is higher compared to equivalent CNN models and lower compared to regular SNN models (e.g., LIF) [75].

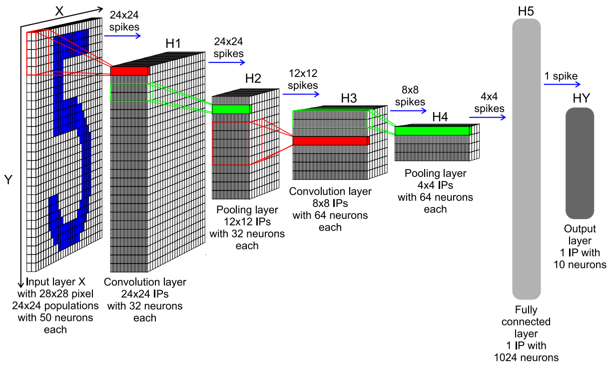

Figure 2.1.: SbS network architecture for handwritten digit classification task.

Table 2.1.: SbS network architecture for handwritten digit classification task.

	Layer size			**Kernel size**	
Layer (H^l)	N_X	N_Y	N_H	K_X	K_Y
Input (HX)	28	28	2	-	-
Convolution ($H1$)	24	24	32	5	5
Pooling ($H2$)	12	12	32	2	2
Convolution ($H3$)	8	8	64	5	5
Pooling ($H4$)	4	4	64	2	2
Fully connected ($H5$)	1	1	1024	4	4
Output (HY)	1	1	10	1	1

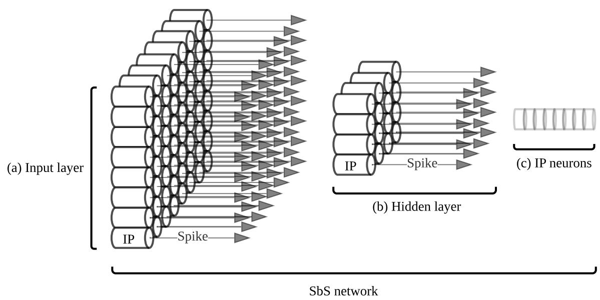

Figure 2.2.: SbS IPs as independent computational entities, (a) illustrates an input layer with a massive amount of IPs operating as independent computational entities, (b) shows a hidden layer with an arbitrary amount of IPs as independent computational entities, (c) exhibits a set of neurons grouped in an IP.

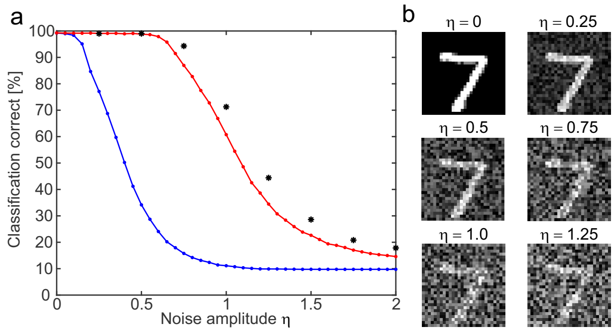

Figure 2.3.: (a) Performance classification of SbS NN versus equivalent CNN, and (b) example of the first pattern in the MNIST test data set with different amounts of positive additive uniformly distributed noise.

Error Tolerance To illustrate the error tolerance of SbS networks, it is presented a classification performance under positive additive uniformly distributed noise as external disturbance. **Fig.** 2.3 presents a comparison of the classification performance of an SbS network and a standard CNN, with the same amount of neurons per layer as well as the same layer structure. Both neural networks are trained for handwritten digit classification on MNIST dataset [76] (see [36] for details). The figure shows the correctness for the MNIST test set with its 10,000 patterns in dependency of the noise level for positive additive uniformly distributed noise. The blue curve shows the performance for the CNN, while the red curve shows the performance for the SbS network with 1200 spikes (iterations). Beginning with a noise level of 0.1, the respective performances are different with a p - level of at least 10^{-6} (tested with the Fisher exact test). Increasing the number of spikes per SbS population to 6000 (performance values shown as black stars), shows that more spikes can improve the performance under noise even more.

2.3. Neural Network Accelerators

Neural network accelerators are specialized hardware components or platforms designed to accelerate the computationally intensive tasks associated with neural networks. Their primary goal is to enhance performance, reduce power consumption, and provide real-time processing capabilities for AI/ML applications [77].

2.3.1. The Need for Accelerators

- **Compute Cost**: AI/ML models, especially CNNs and transformers, are characterized by their deep architectures. Each layer involves a large number of weights and activations. In the forward pass (inference), for each neuron, the input activations are multiplied by corresponding weights, and then all these products are accumulated to produce the neuron output. Similarly, during training, the backpropagation algorithm is also computation-costly.

- **Memory Cost**: AI/ML models, especially those with millions or even billions of parameters, require elevated memory footprint for model storage. During computation, the frequent access to weights, along with the need to read and write intermediate activations, can stress the memory bandwidth. Memory access is also energy-expensive, often more than the actual arithmetic operations.

  As illustration, the memory requirements for an MLP are:

  1. **Weights Storage**: Given a deep neural network with L layers, where each layer l has n_l neurons and receives input from n_{l-1} neurons, the number of weights (excluding biases) is:

  $$W = \sum_{l=2}^{L} n_l \times n_{l-1}$$

  2. **Activations Storage**: Activations need to be stored for the forward pass and are particularly crucial during training for the backpropagation process. For a given layer l, activations storage requirement is proportional to n_l, and the total for the entire network is:

  $$A = \sum_{l=1}^{L} n_l$$

  Considering both weights and activations, the memory access pattern becomes a bottleneck, especially when the model size exceeds the on-chip memory capacity, leading to frequent off-chip accesses which are both time and energy-consuming.

- **Real-time Requirements**: Many contemporary applications demand instantaneous or near-instantaneous processing due to their interactive or safety-critical nature. Hence, the computational backend supporting such applications, often driven by deep neural networks, must be optimized for low-latency and high-throughput to meet the real-time requirements.

- **Energy Efficiency**: Many modern devices, from smartphones to IoT sensors, operate on limited power sources such as batteries. For these devices, the power-hungry computations of AI/ML models can quickly drain the battery, limiting usability, applicability, and functionality. Given that neural network computations are becoming pervasive, even in these power-constrained devices, energy efficiency is of vital importance.

The proliferation of AI/ML applications in low-power devices imposes strict constraints on energy consumption. Factors driving this need include:

1. **Battery Capacity**: Most low-power devices rely on battery power. High energy consumption due to intensive computations can drastically reduce operational time between charges, affecting applicability.
2. **Form Factor and Heat Dissipation**: Smaller devices have smaller batteries and reduced space for cooling mechanisms, making them susceptible to overheating. Hence, energy-efficient computations are not only about battery longevity but also about device temperature, safety, and size.
3. **Operational Continuity in Low-Power Devices**: Many edge devices, such as sensors, are expected to operate continuously. These devices might be located in hard-to-reach places, making frequent battery replacements impractical. Thus, energy efficiency is crucial for operational viability and application feasibility.

Consequently, when deploying neural network models on such devices, it is essential to consider optimizations at both the software and hardware levels to ensure that the power budget is adhered without compromising performance.

2.3.2. Types of Accelerators

- **GPUs**: Historically designed for the purpose of graphics rendering, GPUs have architecture that naturally perform parallel computing. This parallelism is especially beneficial for neural network computations involving repetitive and simultaneous operations. As a result, GPUs have become a cornerstone for deep learning training and inference.

  The success of GPUs in the AI/ML domain is evident from the rise of GPU-optimized deep learning frameworks and the continuous evolution of GPU architectures tailored for neural network computations.

- **ASICs**: they can provide significant benefits in terms of power, performance, and area over general-purpose processors. One of the most notable ASICs designed for neural network computations is Google's TPU. Neuromorphic chips, like IBM's TrueNorth or Intel's Loihi, are ASICs designed to mimic the synapse-neuron connections in the human brain, potentially offering more efficient ways to handle neural network tasks, especially for real-time processing and low-power scenarios.

- **FPGAs**: FPGAs represent a bridge between general-purpose processors and ASICs in terms of adaptability and performance:

  1. **Reconfigurability**: Unlike ASICs, which are fixed in their functionality post-manufacture, FPGAs can be reprogrammed to adopt different logic functions. This means they can be tailored to accelerate specific neural network operations and then reconfigured for another task if needed.
  2. **Parallelism**: FPGAs excel in parallel processing, with their array of logic blocks and interconnects. Neural network computations, which often involve concurrent operations on data, can be accelerated by exploiting this parallelism.
  3. **Prototyping and Evolution**: Given their reconfigurable nature, FPGAs are excellent platforms for prototyping neural network architectures. Furthermore, in environments where neural network models evolve or get updated frequently, FPGAs can adapt without requiring new hardware.
  4. **Trade-offs**: While FPGAs offer flexibility, they might not achieve the same level of performance or energy efficiency as a highly-optimized ASIC for a specific task. However, their adaptability can outweigh this in certain scenarios.

  In the context of the rapidly evolving field of AI/ML, FPGAs provide a compelling balance of adaptability and performance, especially when agility in hardware is desired.

- **NPUs**: NPUs are dedicated hardware accelerators optimized for neural network computation. NPUs streamline both training and inference processes by focusing on operations and data flow patterns typically found in neural networks, resulting in enhanced energy efficiency and performance compared to general-purpose processors.

2.3.3. Design Considerations

- **Precision**: In neural network accelerator design, the precision of arithmetic operations plays a pivotal role. Employing reduced precision arithmetic offers dual advantages: it can markedly accelerate computations and simultaneously diminish power consumption. However, it is crucial to preserve a balance to ensure that the reduced precision does not compromise the accuracy and reliability of the neural network model.

- **Memory Hierarchy and Dataflow**: Memory hierarchy and dataflow are tightly coupled in the design of efficient neural network accelerators. Dataflow refers to the way data is passed and processed between different memory hierarchies and computation units. The choice of dataflow can dramatically impact the energy efficiency, latency, and throughput of the accelerator.

  For neural networks, especially deep learning models, the memory access pattern plays a significant role in determining overall performance. This is because fetching data (e.g., weights, activations) from memory often consumes more energy and time than the arithmetic computations themselves.

- **Scalability**: The capability of a neural network accelerator to extend its computational capacity to handle larger neural networks or to elevate existing hardware performance. This extension can be achieved by either increasing the resources within a single chip (vertical scaling) or by distributing the computation across multiple chips or processing units (horizontal scaling).

  As neural network models become more complex and demand more computational resources, it is crucial for accelerators to be scalable. This ensures that they can continue to provide accelerated performance for newer and larger models without requiring a complete redesign. A modular approach facilitates easy addition of processing units or modules to address scalability needs.

- **Flexibility**: Flexibility remains a cornerstone in the design of neural network accelerators. While tailoring hardware for specific tasks or models can yield substantial performance boosts, it is imperative that these accelerators retain the versatility to accommodate a diverse range of neural network models and operations. This ensures a balance between optimized performance and broad applicability, allowing for both efficiency and adaptability in ever-evolving AI/ML landscapes.

2.4. Precision

Conventional neural networks typically rely on regular FP arithmetic. However, to optimize computational speed, minimize memory footprint, and reduce energy consumption, hardware accelerators adopt lower precision formats. While this can speed up operations and reduce resource demands, there is a trade-off as reduced precision might affect the model accuracy. Therefore, balancing precision and performance is crucial, with techniques such as mixed-precision and dynamic/custom arithmetic being employed to navigate these trade-offs [78].

2.4.1. Fixed-Point

Fixed-point arithmetic represents numbers with a fixed number of digits before and after the decimal point, in contrast to floating-point where the decimal point can "float". Leveraging fixed-point arithmetic offers distinct advantages. Specifically, fixed-point operations are more resource-efficient, leading to expedited computations. Additionally, their inherent simplicity in arithmetic operations often translates to diminished power consumption.

However, while fixed-point arithmetic offers efficiencies in many neural network applications, there are situations where it may not be ideal:

1. **Training**: During training, the need to represent small weight updates and gradient values is critical. FP arithmetic is often preferred to ensure effective backpropagation, whereas fixed-point might impede convergence or acceptable model accuracy [79].

2. **High-Precision**: For tasks requiring acute precision, such as medical imaging or anomaly detection, fixed-point arithmetic might compromise prediction accuracy, especially when using fewer bits.

3. **Transfer Learning and Fine-Tuning**: In scenarios such as fine-tuning pre-trained models, small gradient values are crucial. The reduced precision of fixed-point might neglect these subtle updates.

4. **Normalizing and Batch Normalization**: Operations involving a diverse range of values, like normalization, might introduce significant quantization errors when using fixed-point representations [80].

5. **Recurrent Neural Networks (RNNs)**: RNNs, due to their sequential nature and sensitivity to numerical precision, can experience issues such as exploding or vanishing gradients with fixed-point arithmetic.

6. **Activation Functions with Exponential Ranges**: Functions such as softmax, which operate over a wide range, might be susceptible to quantization errors in a fixed-point context.

While quantization techniques continue to evolve, it remains essential to rigorously evaluate fixed-point representations in the above contexts to ensure desired performance and accuracy.

2.4.2. Floating-Point

FP arithmetic offers distinct advantages due to its capability to represent numbers with both high precision and a wide dynamic range. However, while it provides granular accuracy, it typically demands more computational resources and power compared to fixed-point arithmetic. The inherent robustness of neural networks to numerical perturbations implies a potential avenue for exploring custom reduced-precision FP arithmetic.

The representation of every numerical value, in any numerical system, is made of an integer and a fractional part. The border that delimits them is called the radix point. The fixed-point format for representing numeric values derives its name from the fact that in this format, the base point is fixed at a certain position. For integer numbers, this position is at the right of the least significant digit.

In scientific computation, it is often necessary to represent very large and very small values. This is difficult to achieve using the fixed-point format because the bit size required to maintain both the desired precision and the desired range are very large. In such situations, FP formats are used to represent real numbers. Each FP number can be divided into three fields: sign S, exponent E, and mantissa M. Using the binary number system, it is possible to represent any FP number as:

$$(-1)^S \times 1.M \times 2^{E-B} \tag{2.10}$$

In FP representations the exponent is biased. This bias depends on the bit size of the exponent field. This exponent bias is defined by **Eq.** (2.11), where E_{size} is the exponent bit size.

$$B = 2^{E_{size}-1} - 1 \tag{2.11}$$

There is a natural trade-off between small bit size requiring fewer hardware resources and larger bit size providing higher precision. Within a given total bit size, it is possible to assign various combinations of sizes to the exponent and mantissa fields, with wider exponents resulting in a higher range and wider mantissa resulting in better precision.

The most widely used format for FP arithmetic is the IEEE 754 standard [81]. The IEEE single-precision format (32-bit) is expressed by **Eq.** (2.10) with $B = 127$, 8 bits for the exponent and 23 bits for the mantissa, see **Fig.** 2.4(a). In FP formats, the numbers are normalized, the leading one is an implicit bit, and only the fractional part is explicitly stored in the mantissa

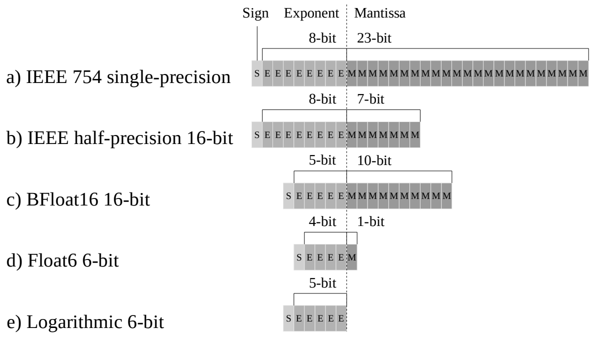

Figure 2.4.: Floating-point number representation.

field.

Reduced bit size than those specified in the IEEE 754 standard are often sufficient to provide the desired precision. Reduced designs require fewer hardware resources enabling low-power implementations. In custom hardware designs, it is possible to customize the FP format implemented. In later sections, the term EaMb is used to denote FP formats, where a and b are the exponent and mantissa bit size, respectively. For example, E4M1 means 4-bit exponent and 1-bit mantissa, see **Fig.** 2.4(d).

There are three special definitions in IEEE 754 standard. The first is subnormal numbers when $E = 0$, then **Eq.** (2.10) is modified to **Eq.** (2.12). Infinity and Not a Number (NaN) are the other two special cases but are not used in our work.

$$(-1)^S \times 0.M \times 2^{1-B} \tag{2.12}$$

Transitioning a neural network model from high-precision to lower-precision computations presents a set of challenges. One of the most critical of these challenges lies in preserving the accuracy of the model while it operates under conditions of reduced numerical precision. This balance requires thoughtful consideration and strategic implementation to ensure that the benefits of computational efficiency do not come at the cost of significant degradation in model performance.

2.4.3. Post-Training Quantization

In Post-Training Quantization (PTQ), a neural network, once fully trained, undergoes a conversion process wherein its FP weights and activations are mapped to a lower precision [82].

Given the full precision weights W of a neural network, they can be quantized to a lower precision using the quantization function:

$$Q(W) = \text{round}\left(\frac{W}{\Delta}\right) \times \Delta \tag{2.13}$$

where Δ is a quantization step size, often derived from the range of the weights or activations and the target precision.

One challenge of PTQ is to ensure minimal loss of accuracy after the conversion. Techniques such as fine-tuning the quantized model, applying regularization during initial training, or using advanced quantization schemes (e.g., mixed precision) can aid in preserving the model accuracy.

While PTQ offers the benefit of simplicity, the resultant quantized model might not be as robust or accurate as models trained with QAT techniques. However, for many applications, especially those on resource-constrained devices, the slight trade-off in accuracy is often outweighed by the benefits in computational efficiency, memory usage, and power consumption.

2.4.4. Quantization-Aware Training

QAT is a technique developed to optimize neural networks for deployment on platforms with limited numerical precision. This quantization maps weights to a lower precision $Q(W)$, similarly as in PTQ. Moreover, QAT integrates this process into the training phase, ensuring that the resultant model is resilient to potential accuracy degradation [82]. Such adaptations are relevant when targeting resource-constrained devices or specialized hardware accelerators that rely on reduced precision for improved computational efficiency and accuracy.

For neural networks with custom reduced FP formats, QAT exhibits even greater versatility. Such custom formats, often denoted as $FP_{M,E}$, allocate specific bit-lengths for the mantissa M and the exponent E. The quantization function for such a format can potentially take the form:

$$Q_{FP}(W) = \text{round}\,(M) \times 2^{\text{round}(E)} \tag{2.14}$$

Where M and E are derived from the mantissa and exponent parts of W, respectively. During training, each forward pass applies the aforementioned quantization, simulating the operational conditions of the reduced precision. The backward pass, essential for gradient-based optimization, is conducted with a higher precision. If ∇W symbolizes the computed gradients, the weight update rule in gradient descent is typically:

$$W_{t+1} = W_t - \eta \nabla W \tag{2.15}$$

Here, η represents the learning rate. However, with QAT, the model remains cognizant of $Q(W)$ or $Q_{FP}(W)$ during forward computations, a factor that influences gradient dynamics. Post-QAT, calibration using a validation dataset aids in refining scaling factors or biases, this improves the model for optimal performance in its intended deployment environment. Assessments against a full-precision baseline are important to ensure that the trade-offs in precision do not compromise the model efficacy.

2.5. Dataflow Taxonomy

Dataflow taxonomy refers to the classification of various schemes that determine how data (weights, activations, and partial results) moves through the accelerator during computation. The way data is moved and reused can have an impact on the energy efficiency and throughput of the accelerator. These strategies aim to maximize performance and energy efficiency by optimizing data movement, which is often more energy-intensive than computation itself. When choosing or designing a dataflow, it is essential to consider the specific neural network workload, the memory hierarchy, and the architectural details of the hardware to ensure an optimal match [77].

Weight Stationary (WS)

In this dataflow, weights are kept stationary in the processing elements. As different input activations come in, they are multiplied with these stationary weights. This approach maximizes the reuse of weights, which can be beneficial when processing a large number of activations, such as during a convolution operation. Characteristics:

Fixed: Weights
Moving: Activations, Partial Sums

Output Stationary (OS)

Here, the partial results (output activations) are kept stationary. Weights and input activations move through the processing elements and the partial sums are accumulated in place. This scheme tries to maximize the reuse of the output from the computation, which is beneficial when a given output is the result of multiple accumulations. Characteristics:

Fixed: Partial Sums
Moving: Weights, Activations

Input Stationary (IS)

In this scheme, input activations remain stationary, while weights move through and are multiplied with these stationary activations. This can be beneficial when a single activation is used in multiple computations. Characteristics:

Fixed: Activations
Moving: Weights, Partial Sums

No Local Reuse (NLR)

As the name suggests, in this dataflow, there is minimal local data reuse. All data types-weights, activations, and partial sums-move through the processing elements. This is often not as efficient in terms of energy consumption since there is a lack of reuse; however, it is simpler in terms of control and design. Characteristic:

Moving: Weights, Activations, Partial Sums

Row Stationary (RS)

This is a specialized dataflow developed for systolic array architectures. In RS, a row of the systolic array holds the activations stationary while weights are propagated horizontally and partial sums propagate vertically.

2.6. Multiply-Accumulate Unit

The dot-product, a foundational operation in neural network computation, involves a series of multiplications followed by their summation. This directly maps onto a series of MAC operations. The MAC operation is foundational in digital signal processing and, it is central to the architecture and efficiency of neural network accelerators. Formally, the MAC operation can be expressed as:

$$\mathrm{ACC} = \mathrm{ACC} + (A \times B) \tag{2.16}$$

Where:

- ACC: Represents an accumulator accumulating the results of the products.
- A and B: Operands subjected to multiplication.

In neural computations, a substantial number of MAC operations are executed. To elucidate, consider the convolutional layer in a CNN. This layer predominantly requires MAC operations to compute the weighted sum of inputs and respective weights. If we denote $x[i]$ as a data array or vector of input values and $w[i]$ as the weights, an output y for a specified filter and input position can be delineated as:

$$y = \sum_i x[i] \cdot w[i] \tag{2.17}$$

This summation is fundamentally a sequence of MAC operations.

2.6.1. Design Considerations

Several considerations come into play to ensure optimal performance, efficiency, and versatility of MAC hardware modules.

1. **Computational Efficiency**: Contemporary neural network models, particularly those under the deep learning paradigm, necessitate executing billions of MAC operations. An well engineered hardware MAC unit can amplify the computation speed. The number of clock cycles it takes to complete a MAC operation can impact the overall performance. High performance is often achieved using parallelism techniques, pipelining, and array-based hardware architectures.

2. **Power Dynamics**: The magnitude of MAC operations in neural networks means that even marginal inefficiencies can escalate into significant power consumption. A carefully designed MAC unit, potentially integrating advanced techniques such as quantization or approximation, can mitigate power exigencies.

3. **Precision Dynamics**: Neural networks, in certain architectures, can accommodate reduced precision. However, the essence of a dynamic MAC unit design is to navigate the balance between computational efficiency and precision.

4. **Scalability Factors**: The ever-evolving domain of neural networks is marked by models growing in complexity and depth. A MAC unit, based on modular design principles, can be seamlessly scaled across extensive accelerator architectures, serving to the spectrum of models.

5. **Adaptive Flexibility**: The dynamism inherent in neural network architectures necessitates a MAC unit enriched with the capacity to adapt to diverse operations, varied data typologies (e.g., floating-point, fixed-point), and a range of hybrid/custom precisions.

The MAC operation holds a pivotal role in the field of neural network accelerator design. The MAC design directly influences the accelerator performance, power efficiency, and overall effectiveness.

2.7. Related Work

For efficient neural network computation, two main optimization strategies are used, namely network compression and classical approximate computing [7]. Researchers focusing on low-power embedded applications started lowering the precision of weights and activation maps to compress the memory footprint of the large number of parameters representing ANNs, a method known as network compression or quantization. This practice takes advantage of the intrinsic error-tolerance of neural networks, as well as their ability to compensate for approximation while training. In this way, reduced bit precision causes a small accuracy loss [8, 9, 10, 11].

In hardware development, Weight Quantization (WQ) has shown up to 2× improvement in energy consumption with an accuracy degradation of less than 1% [83, 84]. Some advanced quantization methods yield to Binary Neural Networks (BNNs) allowing the use of Logical Exclusive Non-Disjunctions (XNORs) instead of the conventional costly MACs [11]. In [85], Sun et al. report an accuracy of 98.43% on handwritten digit classification (MNIST) with a simple BNN. Hence, quantization is a powerful tool for improving the energy efficiency and memory requirements of ANN accelerators, with limited accuracy degradation.

In addition to quantization, network pruning reduces the model size by removing structural portions of the parameters and its associated computations [12, 13]. This method has been identified as an effective technique to improve the efficiency of Deep Neural Network (DNN) for applications with limited computational budget [14, 15, 16].

2.7.1. Low-Power Spiking Neural Network Accelerators

These methods can be used for SNNs as well. In [86], Rathi et al. report up to 3.1× improvement in energy consumption with an accuracy loss of around 3%. Weight quantization allows the designer to realize a trade-off between the accuracy of the SNN application and efficiency of resources. Approximate computing can also be applied at the neuron level, where irrelevant units are deactivated to reduce the computation cost of the SNNs [87]. This computation skipping can be applied randomly on synapses, training ANNs with stochastic synapses improves generalization, resulting in a better accuracy [88, 89]. Such methods are compatible with SNNs and have been tested both during training [90, 91] and operation [92], and even to define the connectivity between layers [93, 94]. Implementations of spiking neuromorphic systems in FPGA [95] and hardware [96] demonstrated that synaptic stochasticity allows to increase the final accuracy of the networks while reducing memory footprint.

Quantization is therefore a powerful technique to improve energy efficiency and memory requirements of ANN and SNN accelerators, with small accuracy degradation. However, this approach requires QAT methods that, in some cases, are problematic or even inaccessible, particularly in emerging deep SNN algorithms [97].

Classical Approximate Computing

Approximate computing has been used in a wide range of applications to increase the computational efficiency in hardware [6]. This approach consists of designing processing elements that approximate their computation by employing modified algorithmic logic units [6]. In [98], Kim et al. have shown SNNs using carry skip adders achieving 2.4× latency enhancement and 43% more energy efficiency, with an accuracy degradation of 0.97% on a handwritten digit classification task (MNIST). Therefore, approximate computing provides important enhancement in energy efficiency and processing speed.

However, as the complexity of the dataset increases, as well as the depth of the network topology, such as ResNet [99] on ImageNet [100], the accuracy degradation becomes more important and may not be negligible anymore [11], especially for critical applications such as autonomous driving. Therefore, it is not certain that network compression techniques and approximate computing are suitable for all applications.

Spike-by-Spike Neural Network Accelerators

Rotermund et al. demonstrated the feasibility of a neuromorphic SbS IP on a Xilinx Virtex 6 FPGA [37]. It provides a massively parallel architecture, optimized to reduce memory access and suitable for ASIC implementations. Nonetheless, this design is considerably resource-demanding if implemented as a full SbS network in today's embedded technology.

2.7.2. Low-Power Convolutional Neural Network Accelerators

In the literature we find plenty of hardware architectures for CNN accelerators implemented in FPGA. Most of the research work implements fixed-point quantization, and very limited research focuses on FP. Moreover, to the best of my knowledge, there is no research work related to FP inference for low-power embedded applications.

Hybrid Custom Floating-Point

In [32], Liangzhen Lai et al. proposed a mixed data representation with floating-point for weights and fixed-point for activations. This work demonstrated on SqueezeNet, AlexNet, GoogLeNet, and VGG-16 that 8-bit floating-point quantization (4-bit exponent and 3-bit mantissa) results in constant negligible accuracy degradation. Similarly, in [101], Sean O. Settle et al. presented an 8-bit FP quantization scheme, which needs an extra inference batch to compensate for quantization errors. However, [32] and [101] did not present a dedicated hardware architecture.

In [102], Xiaocong Lian et al. proposed a hardware accelerator with optimized block floating-point (BFP). In this design the activations and weights are represented by 16-bit and 8-bit FP formats, respectively. This design is demonstrated on Xilinx VC709 evaluation board. This implementation achieves throughput and power efficiency of 760.83 GOP/s and 82.88 GOP/s/W, respectively. However, this design is not suitable for low-power resource-constrained embedded FPGAs.

Low-Precision Floating-Point

In [103], Chunsheng Mei et al. presented a hardware accelerator for VGG16 model using half-precision FP (16-bit). This design is demonstrated on Xilinx Virtex-7 (XC7VX690T) with PCIe interface. This implementation achieves throughput and power efficiency of 202.8 GFLOP/s and 18.72 GFLOP/s/W, respectively. In [104], Chen Wu et al. proposed a low-precision (8-bit) floating-point (LPFP) quantization method for FPGA-based acceleration. This design is demonstrated on Xilinx Kintex 7 and Ultrascale/Ultrascale+. This implementation achieves throughput and power efficiency of 1086.8 GOP/s and 115.4 GOP/s/W, respectively. While these research works focus on low-precision FP, they are not suitable for inference in low-power devices.

Low-Power

Two research papers have been reported hardware accelerators targeting XC7Z007S. This is the smallest and most inexpensive device from Zynq-7000 SoC family. In [105], Paolo Meloni et al. presented a CNN inference accelerator for compact and cost-optimized devices. However, this implementation uses fixed-point to process light-weight CNN architectures with a power efficiency between 2.49 to 2.98 GOPS/s/W.

In [106], Chang Gao et al. presented EdgeDRNN, a RNN accelerator for edge inference. This implementation adopts the SNN inspired delta network algorithm to exploit temporal sparsity in RNNs.

3. Low-Power Spike-by-Spike Neural Network Accelerator: Hybrid 8-bit Floating-Point and 4-bit Logarithmic Computation

3.1. Introduction

The exponential improvement in computing performance and the availability of large amounts of data are boosting the use of AI applications in our daily lives. Among the various algorithms developed over the years, neural networks have demonstrated remarkable performance in a variety of image, video, audio, and text analytics [107, 108].

Historically, ANNs can be classified into three different generations [109]: the first one is represented by the classical McCulloch and Pitts neuron model using discrete binary values as outputs; the second one is represented by more complex architectures as MLP and CNN using continuous activation functions; while the third generation is represented by SNN using spikes as means for information exchange between groups of neurons. Although the AI research is currently dominated by DNNs from the second generation, the SNNs belonging to the third generation are receiving considerable attention [41, 34, 109, 110].

SNNs offer advantageous robustness and the potential to achieve power efficiency closer to that of the human brain. SNNs operate reliably using stochastic elements that are inherently non-reliable mechanisms [33]. This provides superior resistance against adversary attacks [34, 35]. Beside robustness, SNNs have further advantages like the possibility of a more efficient asynchronous parallelization and higher energy efficiency than DNNs. For example, Loihi [42], a SNN developed by Intel, can solve LASSO optimization problems with an over three orders of magnitude better energy-delay product than conventional approaches. These advantages are motivating large research programs by major companies (e.g., Intel [42] and IBM [40]) as well as pan-european projects in the domain of spiking neural networks [41].

SNNs emulate the real behavior of neurons in different levels of detail. The more detailed the biological part is emulated, the greater the computational complexity [75, 111]. For example, LIF is a widely used model; however, this model is relatively more complex for emulation in low-power embedded applications.

Alternatively, the SbS neural network is a remarkable model for its reduced complexity, which is on the less realistic side of the SNN scale of biological realism [36, 34]. Consequently, the hardware complexity of SbS network implementations is reduced [45, 37]. In spite of this, SbS still uses stochastic spikes as a means of transmitting information between populations of neurons and thus retains the advantageous robustness of SNNs.

The conceptual model in SbS (see Chapter 2 for a review) differs fundamentally from conventional ANNs since (a) the building blocks of the network are IPs which are an optimized generative representation with non-negative values, (b) time progresses from one spike to the next, preserving the property of stochastically firing neurons, and (c) a network has only a small number of parameters, which is a noise-robust stochastic version of NNMF. The SbS network is placed between non-spiking Neural Networks (NNs) and stochastically spiking NNs, which offers advantages from both structures [36]. On one hand, the SbS model incorporates the inherent robustness of SNNs, which gives the possibility of more efficient asynchronous parallelization and resilience against disturbances based on the synaptic stochasticity; on the other hand, the SbS model incorporates the regular flow of information from CNNs.

As computational demanding algorithms, CNNs and SNNs in particular, must be addressed by specialized hardware architectures. A significant research effort has been performed in SNN accelerators, see e.g. [38, 7, 39, 40, 41, 42]. However, hardware accelerators that focus on SbS have only been partially investigated so far [37]. Enhancing SbS accelerators will contribute to the deployment of robust neural networks in resource-constrained devices [45, 34, 43, 44].

A central point that can be optimized in current SbS accelerators is the use of approximation techniques. Most SNN models use FP numerical representation, which imposes high complexity of the required circuits for FP operations. Quantization has the potential to improve computational performance; however, this solution is often accompanied by quantization-aware training methods that, in some cases, are problematic or even inaccessible, particularly in deep SNN algorithms [97].

As an alternative, based on the relaxed need for fully precise or deterministic computation of neural networks, approximate computing techniques allow substantial enhancement in processing efficiency with moderated accuracy degradation. Some research papers have shown the feasibility of applying approximate computing to the inference stage of neural networks [21, 24, 23, 22]. Such techniques usually demonstrated small inference accuracy degradation, but significant enhancement in computational performance, chip-area, and energy consumption. Hence, by taking advantage of the intrinsic error-tolerance of neural networks, approximate computing is positioned as a promising approach for inference on resource-limited devices.

In this chapter, it is presented an accelerator for SbS neural networks with a dot-product hardware design based on approximate computing with hybrid custom FP and logarithmic number representation. This hardware unit has a quality configurable scheme based on the exponent and mantissa bit-size of the synaptic-weight vector. **Fig.** 3.1 illustrates the dot-product hardware module with standard FP (IEEE 754) arithmetic, and our approach with hybrid custom FP as well as logarithmic approximation. As a design parameter, the mantissa bit-width of the weight vector provides a tunable knob to trade-off between efficiency and QoR [112, 6]. Since the lower-order bits have smaller significance than the higher-order bits, bit-truncation strategy represents a minor impact on QoR [113, 114]. Further on, the mantissa bits can be completely removed in order to use only the exponent of a FP representation. This configuration becomes a logarithmic representation, which consequently leads to significant architectural-level optimizations using only hardware adders and shifters for dot-product approximation. Moreover, since approximations and noise have qualitatively the same effect [115], it is proposed the noise tolerance plot as an intuitive visual measure to provide insights into the quality degradation and resilience budget of SbS networks under approximation effects.

The main contributions presented in this chapter are as follows:

- A hardware module for dot-product approximation. To perform the sum of pairwise products of two vectors, this hardware module has the following three design features: (1) the pairwise product is approximated by adding integer exponents and multiplying truncated mantissas, and the sum of products is done by accumulating denormalized integer products with barrel shifters, this increases computational throughput; (2) the synaptic weight vector uses either reduced custom FP or logarithmic representation, this reduces memory footprint; and (3) the neuron vector uses either standard or custom FP representation, this preserves QoR and overall inference accuracy.

- A hardware design exploration with the proposed dot-product approximation using synaptic weight vectors with custom FP and logarithmic representation as shown in **Fig.** 3.1. It is presented the inference run-time, accuracy degradation, resource utilization and power dissipation. Experimental results demonstrate 20.5× run-time enhancement versus embedded CPU (ARM Cortex-A9 at 666 MHz), and less than 0.5% of accuracy degradation without retraining on a handwritten digit recognition task (MNIST). This machine learning task simply provides a proof of concept to demonstrate the feasibility of our approximation technique for SbS neural network accelerators.

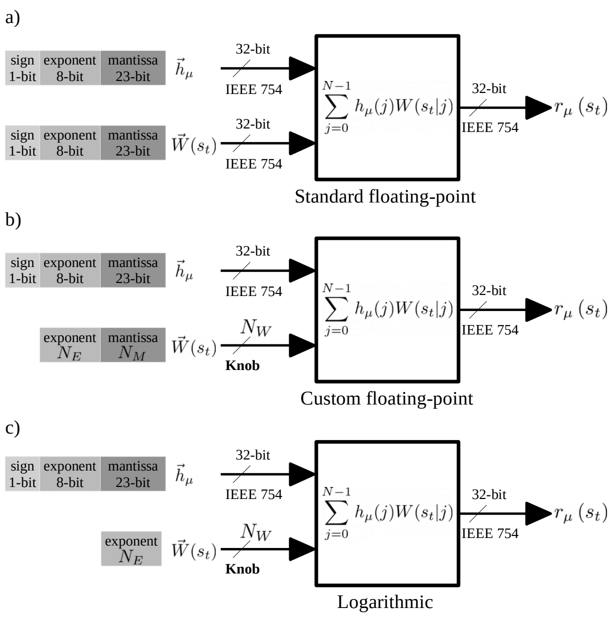

Figure 3.1.: Dot-product hardware module with (a) standard floating-point (IEEE 754) arithmetic, (b) hybrid custom floating-point approximation, and (c) hybrid logarithmic approximation.

- A noise tolerance plot is proposed as quality monitor, which serves as an intuitive visual model to provide insights into the accuracy degradation and noise resilience-budget of SbS networks under approximate processing effects.

- The present design for dot-product approximation is adaptable as a building block for other error resilient applications (e.g., image/video processing).

To promote the research on SbS networks, the design exploration framework is made available to the public as an open-source project at https://github.com/YaribNevarez/sbs-framework.git

3.2. System Design

In this section, it is presented a hardware architecture composed of specialized heterogeneous Processing Units (PUs) with hybrid custom floating-point and logarithmic dot-product approximation. This approach represents an advantageous design for error resilient applications in resource-constrained devices due to the reduced hardware utilization and memory footprint. Furthermore, the proposed approach allows the implementation of stationary synaptic weight matrices as internal accelerator storage based on the reduced memory footprint.

Regarding the software architecture, this is structured as a layered object-oriented application framework written in the C programming language. This offers a comprehensive high level embedded software Application Programming Interface (API) that allows the construction of scalable sequential SbS networks with configurable hardware acceleration. Conceptually this design is modular, reusable, and extensible. The overall structure is depicted in **Fig.** 3.2.

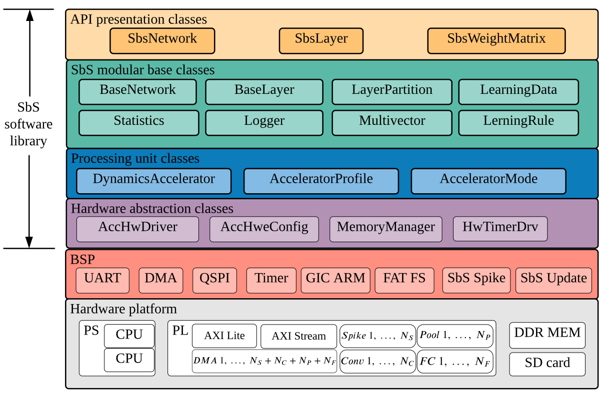

Figure 3.2.: System-level overview of the embedded software architecture.

3.2.1. Hardware Architecture

As a hardware/software co-design, the system architecture is an embedded CPU+FPGA-based platform, where the acceleration of SbS network computation is based on asynchronous[1] execution of parallel heterogeneous processing units: *Spike* (input layer), *Conv* (convolution), *Pool* (pooling), and *FC* (fully connected). **Fig.** 3.3 illustrates the system overview as a scalable structure. For hyperparameter configuration, each PU uses AXI-Lite interface. For data transfer, each PU uses AXI-Stream interfaces via Direct Memory Access (DMA) allowing data movement with high transfer rate. Each PU asserts an interrupt flag once the job or transaction is complete. This interrupt event is handled by the embedded CPU to collect results and start a new transaction.

The hardware architecture can resize its resource utilization by changing the number of PU instances prior to the hardware synthesis, this provides scalability with a good trade-off between area and throughput. The dedicated PUs for *Conv* and *FC* implement the proposed dot-product approximation as a system component. The PUs are written in System C using Xilinx Vivado High-Level Synthesis (HLS). In this research, we illustrate the integration of the approximate dot-product component on the *Conv* PU.

3.2.2. Conv Processing Unit

This hardware module computes the dynamics of the IP defined by **Eq.** (2.9) and offers two modes of operation: *configuration* and *computation*.

[1]The system is synchronous at the circuit level, but the execution is asynchronous in terms of jobs.

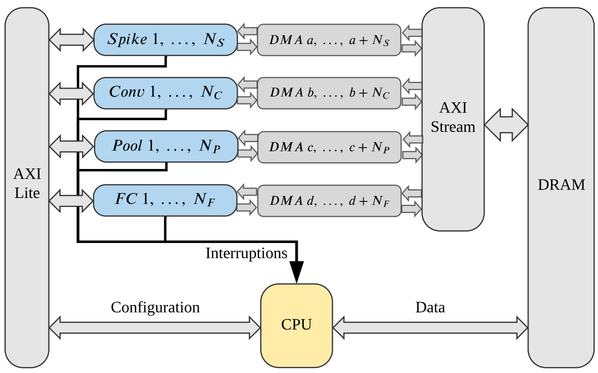

Figure 3.3.: System-level hardware architecture with scalable number of heterogeneous PUs: *Spike*, *Conv*, *Pool*, and *FC*

Configuration Mode

In this mode of operation, the PU receives and stores in on-chip memory (BRAM) the hyperparameters to compute the IP dynamics: ϵ as the epsilon, N as the length of $\vec{h}_\mu \in \mathbb{R}^N$, $K \in \mathbb{N}$ as the size of the convolution kernel, and $H \in \mathbb{N}$ as the number of IPs to process per transaction. H is the number of IPs forming a layer or a partition.

Additionally, the processing unit also stores in on-chip memory (BRAM) the synaptic weight matrix using a number representation with a reduced memory footprint. Fundamentally, the synaptic weight matrix is defined by $W \in \mathbb{R}^{K\times K\times M\times N}$ with $0 \leq W(s_t|j) \leq 1$ and $\sum_{s_t=0}^{M-1} W(s_t|j) = 1$ [36]. Hence, W employs only positive normalized real numbers. Therefore, W is deployed using a reduced floating-point or logarithmic representation as follows:

- Custom floating-point representation. In this case, W is deployed with a reduced floating-point representation using the designer defined bit-width for the exponent and for the mantissa. For example, 4-bit exponent, 1-bit mantissa; as a result: 5-bit custom floating-point. The proposed method to determine the required bit-width is described in Section 3.2.3.

- Logarithmic representation. In this case, the synaptic weight matrix is $W \in \mathbb{N}^{K\times K\times M\times N}$ with positive natural numbers. Since $0 \leq W(s_t|j) \leq 1$ and $\sum_{s_t=0}^{M-1} W(s_t|j) = 1$, W has only negative values in the logarithmic domain. Hence, the sign bit is omitted, and the values are represented as natural numbers. Therefore, W is deployed with a representation using the necessary bit-width for the exponent according to the given application. For example, 4-bit exponent. The method to determine the required bit-width is described in Section 3.2.3.

In order to deploy different SbS models, the *Conv* processing units can load different hyperparameters and synaptic weight matrices as required via the embedded software.

Computation Mode

In this mode of operation, the PU executes a transaction to process a group of IPs using the previously given hyperparameters and synaptic weight matrix. This process operates in six stages as shown in **Fig.** 3.4. In the first two stages, the PU receives $\vec{h}_\mu \in \mathbb{R}^N$, then the PU calculates the emitted spike and stores it in $S^{new} \in \mathbb{N}^H$ (output spike vector). From the third to the fifth stage, the PU receives $S_t \in \mathbb{N}^{K\times K}$ (input spike matrix), then it computes the update dynamics, and then it dispatches $\vec{h}_\mu^{new} \in \mathbb{R}^N$ (updated IP). This process repeats for H number of loops (for each IP of the layer or partition). Finally, S^{new} is dispatched.

The computation of the update dynamics (see **Fig.** 3.4(d)) operates in two stages or hardware modules: *dot-product* and *neuron update*. First, the *dot-product* module calculates the sum of pairwise products of $\vec{h}_\mu$ and $\vec{W}(s_t)$, each pairwise product is stored as intermediate results. Subsequently, the *neuron update* module calculates **Eq.** (2.9) reusing parameters and previous intermediate results.

The calculation of the dot-product of **Eq.** (2.9) represents a considerable computational cost using standard floating-point in non-quantized network models. Fortunately, the pair product of $h_\mu(j)$ and $W(s_t|j)$ was defined by us as an approximable factor in the dot-product of **Eq.** (2.9). In the following section, we focus on an optimized dot-product hardware design based on approximate computing.

3.2.3. Dot-Product Hardware Module

The dot-product hardware module is part of an application-specific architecture optimized to approximate the dot-product of arbitrary length vectors, see **Eq.** (3.1). For quality configurability, we parameterized the mantissa bit-width of $\vec{W}(s_t)$, which provides a tunable trade-off between resource utilization and QoR. Since the lower-order bits have smaller significance than the higher-

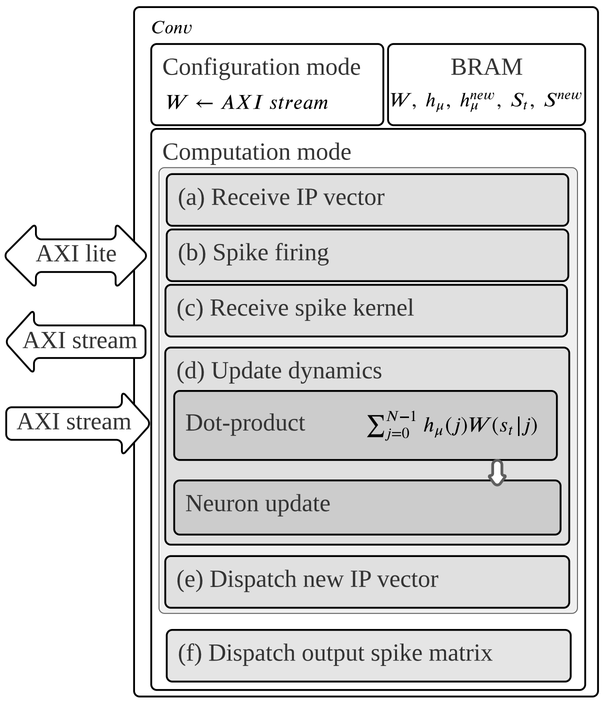

Figure 3.4.: The *Conv* processing unit and its six stages: (a) receive IP vector, (b) spike firing, (c) receive spike kernel, (d) update dynamics, (e) dispatch new IP vector, (f) dispatch output spike matrix.

order bits, removing them may have only a minor impact on QoR. We designate this as hybrid custom floating-point approximation (see **Fig.** 3.1(b)).

$$r_{\mu}(s_t) = \sum_{j=0}^{N-1} h_{\mu}(j) W(s_t|j) \tag{3.1}$$

Further on, we remove the mantissa bits completely in order to use only the exponent of a floating-point representation. Hence, the worst-case quality and yet the most efficient configuration becomes a logarithmic representation. Consequently, this structure leads to advantageous architectural optimizations using only adders and barrel shifters for dot-product approximation in hardware. We designate this as hybrid logarithmic approximation (see **Fig.** 3.1(c)).

In order to determine the required bit-width for the number representation, we use **Eq.** (3.2), **Eq.** (3.3), and **Eq.** (3.4).

$$E_{\min} = \log_2(\min_{\forall i}(W(i))) \tag{3.2}$$

$$N_E = \lceil \log_2(|E_{\min}|) \rceil \tag{3.3}$$

$$N_W = N_E + N_M \tag{3.4}$$

The **Eq.** (3.2) obtains the exponent of the minimum entry value in the synaptic weight matrix. Since $0 \leq W(s_t|j) \leq 1$ and $\sum_{s_t=0}^{M-1} W(s_t|j) = 1$, W has only negative values in the logarithmic domain; the smallest value is expressed by the biggest negative exponent ($E_{\min}$). Then, the **Eq.** (3.3) obtains the necessary bit-width to represent the exponent (N_E). Finally, we obtain the total bit-width by incorporating both exponent and mantissa bit-widths in **Eq.** (3.4). N_M denotes the mantissa bit-width, this is a knob parameter that is tuned by the designer to trade-off between resource utilization and QoR. The bit-width concept is illustrated in **Fig.** 3.1.

In this section, we will present three pipelined hardware modules with standard floating-point (IEEE 754) computation, hybrid custom floating-point approximation, and hybrid logarithmic approximation.

Dot-Product with Standard Floating-Point Computation

The hardware module to calculate the dot-product with standard floating-point computation is shown in **Fig.** 3.5. This diagram presents the hardware blocks and their clock cycle schedule. This module loads both $h_\mu(j)$ and $W(s|j)$ from BRAM, then the PU executes the pairwise product (**Fig.** 3.5(c)) and accumulation (**Fig.** 3.5(d)). Intermediate results of $h_\mu(j)W(s_t|j)$ are stored in BRAM for reuse in the neuron update stage. The latency in clock cycles of this hardware module is defined by **Eq.** (3.5), where N is the vector length of the dot-product. This equation is obtained from the general pipelined hardware latency formula: $L = (N-1)\,II + IL$, where II is the initiation interval (**Fig.** 3.5(a)), and IL is the iteration latency (**Fig.** 3.5(b)). Both II and IL are obtained from the high-level synthesis analysis. The equation for the latency with standard 32-bit floating-point is:

$$L_{f32} = 10N + 9 \tag{3.5}$$

In this design, the high-level synthesis tool infers computational blocks with considerable latency cost for standard floating-point. In the case of floating-point multiplication (**Fig.** 3.5(c)), the synthesis infers a hardware block with a latency cost of 5 clock cycles. This block executes addition of exponents, multiplication of mantissas, and mantissa correction (when needed). Moreover, in the case of floating-point addition (3.5(d)), the synthesis infers a hardware block with a latency cost of 9 clock cycles. Seemingly, this block executes alignment of mantissas, addition, and correction (when needed). Therefore, the use of standard floating-point results in high computational cost, this represents unnecessary overhead in error tolerant applications.

Dot-Product with Hybrid Custom Floating-Point and Logarithmic Computation

The hardware module to calculate dot-product with hybrid custom floating-point approximation is shown in **Fig.** 3.6. In this design, $h_\mu(j)$ uses standard 32-bit floating-point number representation, and $W(s|j)$ uses a positive reduced custom floating-point number representation, where the mantissa bit width is the quality configurability knob. This parameter is tuned by the designer to trade-off between QoR and resource utilization, thus, energy consumption.

As the most efficient setup, by completely truncating the mantissa of $W(s|j)$ leads to a slightly different hardware architecture using only adders and shifters, which computes the dot-product with hybrid logarithmic approximation. This is shown in **Fig.** 3.7.

Additionally, the exponent bit-width of $W(s|j)$ is a design parameter for efficient resource utilization and it is defined based on the application and deployment needs.

The hybrid custom floating-point and logarithmic approximation designs work in three phases:

Computation, *Threshold-test*, and *Result normalization*.

- Phase I, *Computation*:
  This phase approximates the magnitude of the dot-product in a denormalized representation. This is calculated in two iterative steps over each vector element: *pairwise product* and *accumulation*. *Pairwise product* is executed either in hybrid custom floating-point or hybrid logarithmic approximation described below.
  - Pairwise product.
    - Hybrid custom floating-point approximation. As shown in **Fig.** 3.6(c) in dark-gray, the pairwise product is approximated by adding exponents and multiplying mantissas of $W(s|i)$ and $h_\mu(i)$. If the mantissa multiplication results in an overflow, then it is corrected by increasing the exponent and shifting the resulting mantissa by one position to the right. Then, as intermediate result, $h_\mu(j)W(s_t|j)$ is stored for future reuse in the neuron update calculation. In this design, the pairwise product has a latency of 5 clock cycles.
    - Hybrid logarithmic approximation. As shown in **Fig.** 3.7(c) in dark-gray, the pairwise product is approximated by adding $W(s|i)$ to the exponent of $h_\mu(i)$,

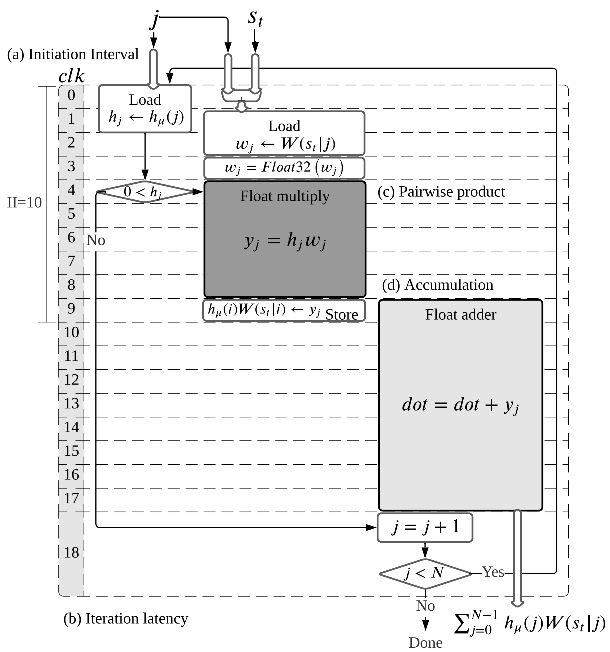

Figure 3.5.: Dot-product hardware module with standard floating-point (IEEE 754) computation, (a) exhibits the initiation interval of 10 clock cycles, (b) presents the iteration latency of 19 clock cycles, (c) shows the pairwise product block in dark-gray, and (d) illustrates the accumulation block in light-gray.

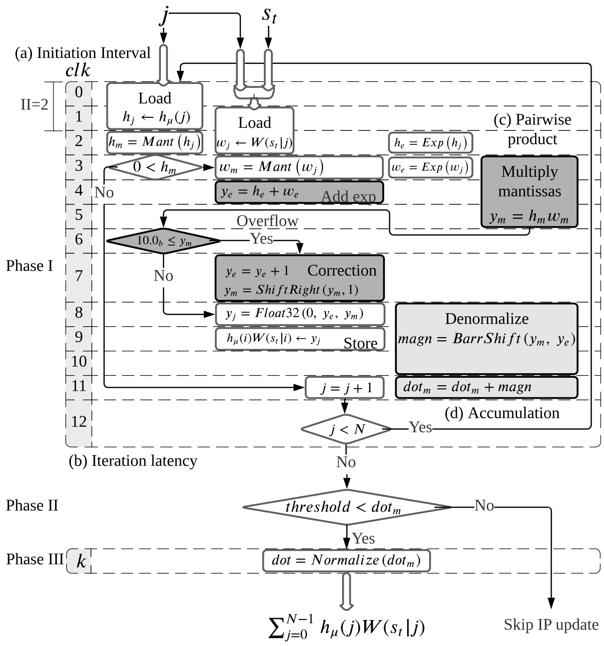

Figure 3.6.: Dot-product hardware module with hybrid custom floating-point approximation, (a) exhibits the initiation interval of 2 clock cycles, (b) presents the iteration latency of 13 clock cycles, (c) shows the pairwise product blocks in dark-gray, and (d) illustrates the accumulation blocks in light-gray.

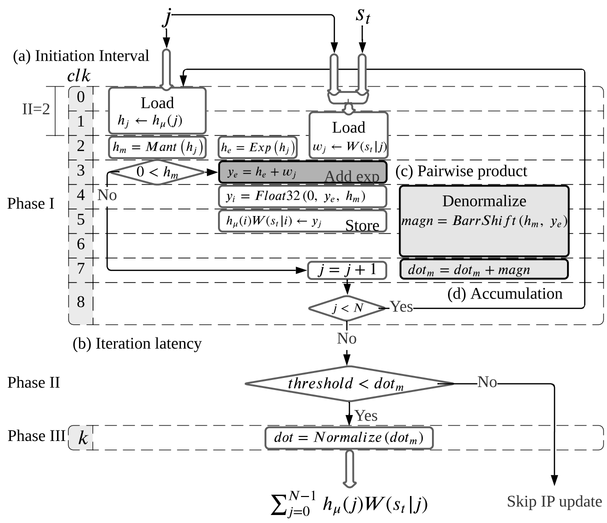

Figure 3.7.: Dot-product hardware module with hybrid logarithmic approximation, (a) exhibits the initiation interval of 2 clock cycles, (b) presents the iteration latency of 9 clock cycles, (c) shows the pairwise product block in dark-gray, and (d) illustrates the accumulation blocks in light-gray.

since the values of $W(s|j)$ are represented in the logarithmic domain and $h_\mu(j)$ in standard floating-point. In this design, the pairwise product has a latency of one clock cycle.

– Accumulation. As shown in both **Fig.** 3.6(d) and **Fig.** 3.7(d) in light-gray, first, it is obtained the denormalized representation of $h_\mu(j)W(s_t|j)$ by shifting its mantissa using its exponent as shifting parameter (barrel shifter). Then, this denormalized representation is accumulated to obtain the approximated magnitude of the dot-product.

The process of pairwise product and accumulation iterates over each element of the vectors. The computation latency is given by **Eq.** (3.6) for hybrid custom floating-point, and **Eq.** (3.7) for hybrid logarithmic, where N is the length of the vectors. Both pipelined hardware modules have the same throughput, since both have two clock cycles as initiation interval.

$$L_{custom} = 2N + 11 \tag{3.6}$$

$$L_{log} = 2N + 7 \tag{3.7}$$

- Phase II, *Threshold-test*:
  The accumulated denormalized magnitude is tested to be above of a predefined threshold, it must be above zero, since the dot-product is the denominator in **Eq.** (2.9). If passing the threshold, then the next phase is executed. Otherwise the rest of update dynamics is skipped. The threshold-test takes one clock cycle.

- Phase III, *Result-normalization*:
  In this phase, the dot-product is normalized to obtain the exponent and mantissa in order to convert it to standard floating-point for later use in the neuron update. The normalization is obtained by shifting the approximated dot-product magnitude in a loop until it is in the form of a normalized mantissa where the iteration count represents the exponent of the dot-product. Each iteration takes one clock cycle.

The total latency of the hardware module with hybrid custom floating-point and hybrid logarithmic approximation is the accumulated latency of the three phases.

The proposed architectures with approximation approach exceeds the performance of the design with standard floating-point. This performance enhancement is achieved by decomposing the floating-point computation into an advantageous handling of exponent and mantissa using intermediate accumulation in a denormalized representation and only one final normalization.

3.3. Experimental Results

The proposed architecture is demonstrated on a Xilinx Zynq-7020. This device integrates a dual ARM Cortex-A9 based Processing System (PS) and Programmable Logic (PL) equivalent to Xilinx Artix-7 (FPGA) in a single chip [116]. The Zynq-7020 architecture conveniently maps the custom logic and software in the PL and PS respectively as an embedded system.

In this platform, the proposed hardware architecture is implemented to deploy the SbS network structure shown in 2.1 for handwritten digit classification task for MNIST data set. The SbS model is trained using standard floating-point. Matlab software is used for this SbS network implementation. The resulting synaptic weight matrices are deployed on the embedded system as binary files stored in a micro SD memory card. In the embedded software, the SbS network is built as a sequential model using the API from the SbS embedded software framework [45]. This API allows to configure the computational workload of the neural network, this can be distributed among the hardware processing units and the embedded CPU.

For the evaluation of this approach, it is presented a design exploration by reviewing the computational latency, inference accuracy, resource utilization, and power dissipation. First, the performance of the embedded CPU is taken as benchmark, and then repeat the measurements on hardware processing units with standard floating-point computation. Afterwards, the dot-product architecture is evaluated addressing a design exploration with hybrid custom floating-point approximation, as well as the hybrid logarithmic approximation. Finally, a discussion of results is presented.

3.3.1. Performance Benchmark

Benchmark on Embedded CPU

The performance of the CPU for SbS network inference is examined. In this case, the embedded software builds the SbS network as a sequential model mapping the entire computation to the CPU (ARM Cortex-A9) at 666 MHz and a power dissipation of 1.658 W.

The SbS network computation on the CPU reaches a latency of 34.28 ms per spike with accuracy of 99.3 % correct classification on the 10, 000 image test set with 1000 spikes. The latency and schedule of the SbS network computation are displayed in **Tab.** 3.1 and **Fig.** 3.8, respectively.

Table 3.1.: Computation on embedded CPU.

Layer	Latency (ms)
HX_IN	1.184
H1_CONV	4.865
H2_POOL	3.656
H3_CONV	20.643
H4_POOL	0.828
H5_FC	3.099
HY_OUT	0.004
TOTAL	34.279

Benchmark on Processing Units with Standard Floating-Point Computation

The system architecture shown in **Fig.** 3.9 is implemented to benchmark the computation on hardware PUs with standard floating-point. The embedded software builds the SbS network as a sequential model and delegates the network computation to the hardware processing units at 200 MHz as clock frequency.

The layers of the neural network with the most neurons are partitioned for asynchronous parallel processing. Since *H2_POOL* and *H3_CONV* are the layers with the most neurons, the computational workload is distributed between two PUs for each one of these layers. The output layer *HY_OUT* is fully processed by the CPU, since it is the layer with fewest neurons. The

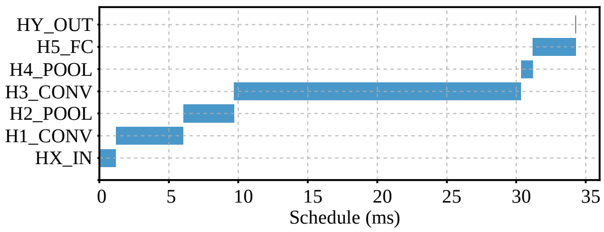

Figure 3.8.: Computation on embedded CPU.

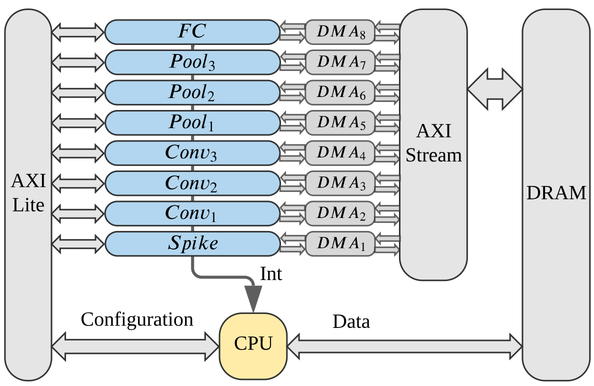

Figure 3.9.: System overview of the top-level architecture with 8 processing units.

Table 3.2.: Performance of processing units with standard floating-point (IEEE 754) computation.

Hardware mapping		**Computation schedule (ms)**			
Layer	**PU**	t_s	t_{CPU}	t_{PU}	t_f
HX_IN	Spike	0	0.056	0.370	0.426
H1_CONV	Conv1	0.058	0.598	2.002	2.658
H2_POOL	Pool1	0.658	0.126	1.091	1.875
	Pool2	0.785	0.125	1.075	1.985
H3_CONV	Conv2	0.911	0.280	3.183	4.374
	Conv3	1.193	0.279	3.176	4.648
H4_POOL	Pool3	1.473	0.037	0.481	1.991
H5_FC	FC	1.512	0.101	1.118	2.731
HY_OUT	CPU	1.615	0.004	0	1.619

hardware mapping and the computation schedule of this deployment are displayed in **Tab.** 3.2 and **Fig.** 3.10, respectively.

In the computation schedule, the following terms are defined as follows: $t_s(n)$ as the start time for the processing of the neural network layer (as a compute node) $n \in L$ where L represents the set of layers; $t_{CPU}(n)$ as the CPU preprocessing time; $t_{PU}(n)$ as the PU latency; and $t_f(n)$ as the finish time. For data preparation, $t_{CPU}(n)$ is the duration in which the CPU writes a DRAM buffer with $\vec{h}_\mu$ (vector of neuron latent variables) of the current processing layer and S_t (input spike matrix) from its preceding layer. This buffer is streamed to the PU via DMA.

The total execution time of the CPU is defined by **Eq.** (3.8). In a cyclic spiking inference, the execution time of the network computation is the longest path among the processing units including the CPU. This is denoted as the latency of an spike cycle and it is defined by **Eq.** (3.10). The total execution time of the network computation is the last finish time (t_f) in the schedule defined by **Eq.** (3.11).

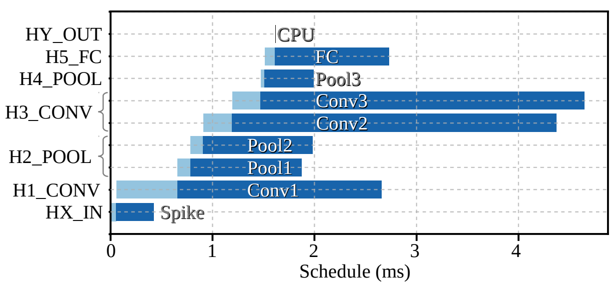

Figure 3.10.: Performance of processing units with standard floating-point (IEEE 754) computation.

$$T_{CPU} = \sum_{n \in L} t_{CPU}(n) \tag{3.8}$$

$$T_{PU} = \max_{n \in L}(t_{PU}(n)) \tag{3.9}$$

$$T_{SC} = \begin{cases} T_{PU}, & \text{if } T_{CPU} \leq T_{PU} \\ T_{CPU}, & \text{otherwise} \end{cases} \tag{3.10}$$

$$T_f = \max_{n \in L}(t_f(n)) \tag{3.11}$$

Using standard floating-point requires a high computational cost. As the largest layer, the computational workload of *H3_CONV* is evenly partitioned among two PUs: *Conv2* and *Conv3*. However, in the cyclic schedule, *Conv2* causes the performance bottleneck as shown in **Fig.** 3.11. In this case, the CPU awaits for *Conv2* to finish the computation of the previous cycle in order to start the current computation cycle. In contrast, as the smallest layer, the computational workload of *HY_OUT* is fully processed by the CPU. **Tab.** 3.2 and **Fig.** 3.10 show 4 μs as the

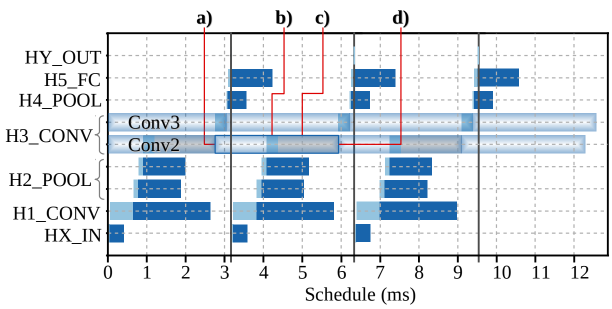

Figure 3.11.: Performance bottleneck of cyclic computation on processing units with standard floating-point (IEEE 754) arithmetic, (a) exhibits the starting of t_{PU} of *Conv2* on a previous computation cycle, (b) presents t_{CPU} of *Conv2* on the current computation cycle, (c) shows the CPU waiting time (in gray color) for *Conv2* as a busy resource (awaiting for *Conv2* interruption), and (d) illustrates the t_f from the previous computation cycle, the starting of t_{PU} on the current computation cycle (*Conv2* interruption on completion, and start current computation cycle).

processing latency of *HY_OUT*. This latency is negligible compared to the overall performance assessment. Accelerating *HY_OUT* would yield a negligible gain. Moreover, assigning a dedicated hardware PU to *HY_OUT* would add unprofitable data transfer and hardware interruption handling overheads.

Applying **Eq.** (3.10), it is obtained a latency of 3.18 ms per spike cycle. This deployment achieves an accuracy of 98.98% correct classification on the 10, 000 image test set with 1000 spikes.

The post-implementation resource utilization and power dissipation are shown in **Tab.** 3.3. Each *Conv* PU instantiates an on-chip stationary weight matrix of 52, 000 entries, wish is sufficient to store $W \in \mathbb{R}^{5\times5\times2\times32}$ and $W \in \mathbb{R}^{5\times5\times32\times64}$ for *H1_CONV* and *H3_CONV*, respectively. In order to reduce BRAM utilization, we use a custom floating-point representation composed of 4-bit exponent and 4-bit mantissa (bit sign is omitted). Each 8-bit entry is promoted to its standard floating-point representation for the dot-product computation. The method to find the appropriate bit-width parameters for custom floating-point representation is presented in Section 3.3.2.

Table 3.3.: Resource utilization and power dissipation of processing units with standard floating-point (IEEE 754) computation.

PU	LUT	FF	DSP	BRAM 18K	Power (mW)
Spike	2,640	4,903	2	2	38
Conv	2,765	4,366	19	37	89
Pool	2,273	3,762	5	3	59
FC	2,649	4,189	8	9	66

The implementation of dot-product with standard floating-point arithmetic (IEEE 754) utilizes proprietary Xilinx multiplier and adder floating-point operator cores. Vivado HLS implements floating-point arithmetic operations by mapping them onto Xilinx LogiCORE IP cores, these floating-point operator cores are instantiated in the resultant Register-Transfer Level (RTL)[117]. In this case, the implementation of the dot-product with the standard floating-point computation reuses the multiplier and adder cores already instantiated and used in other computation sections of *Conv* and *FC* processing units. The post-implementation resource utilization and power dissipation of the floating-point operator cores are shown in **Tab.** 4.3.

Table 3.4.: Resource utilization and power dissipation of multiplier and adder floating-point (IEEE 754) operator cores.

Core operation	DSP	FF	LUT	Latency (clk)	Power (mW)
Multiplier	3	151	325	4	7
Adder	2	324	424	8	6

Benchmark on Noise Tolerance Plot

The purpose of the proposed noise tolerance plot is to serve as an intuitive visual model used to provide insights into accuracy degradation under approximate processing effects. This plot reveals inherent error resilience, and hence, approximation resilience. As an application-specific quality metric, this plot offers an effective method to estimate the overall quality degradation of the SbS network under different approximate processing effects, since both approximations and noise have qualitatively the same effect [115].

In order to experimentally obtain the noise tolerance plot, the inference accuracy of the neural network with increasing number of spikes is measured. The measurements are retaken with uniformly distributed noise applied on the input. The levels of the noise amplitude are gradually ascended until accuracy degradation is detected. **Fig.** 3.12 demonstrates this method using 100 input samples.

As benchmark, the tolerance plot in **Fig.** 3.12 revels accuracy degradation having 50% noise and convergence with 400 spikes. In this case, the given SbS network with precise processing demonstrates its inherent error resilience, hence, the resilience for approximate processing.

3.3.2. Design Exploration with Hybrid Custom Floating-Point and Logarithmic Computation

In this section, it is presented a design exploration to evaluate the proposed approach for SbS neural network inference using hybrid custom floating-point and logarithmic approximation. First, the synaptic weight matrix of each layer is examined in order to determine the minimum requirements for numeric representation and memory storage. Second, the proposed dot-product architecture is implemented using the minimal floating-point and logarithmic representation as design parameters. Finally, it is presented an evaluation of the overall performance, inference accuracy, resource utilization, and power dissipation.

Parameters for Numeric Representation of Synaptic Weight Matrix

The parameters for numerical representation of the synaptic weight matrices is obtained from their $\log_2$-histograms presented in **Fig.** 3.13. These histograms show the distribution of synaptic weight values in each matrix. The histograms show that the minimum integer exponent value is -13. Hence, applying **Eq.** (3.2) and **Eq.** (3.3) to the given SbS network, results $E_{\min} = -13$ and $N_E = 4$, respectively. Therefore, 4-bits are used for the absolute binary representation of the exponents.

For quality configurability, the mantissa bit-width is a knob parameter that is tuned by the designer. This procedure leverages the builtin error-tolerance of neural networks and performs a trade-off between resource utilization and QoR. In the following subsection, a case study is presented with 1-bit mantissa. This corresponds to the custom floating-point approximation.

Design Exploration for Dot-product with Hybrid Custom Floating-Point Computation

For this design exploration, a custom floating-point representation is composed of 4-bit exponent and 1-bit mantissa. This format is used for the synaptic weight vector on the proposed dot-product architecture. Each *Conv* PU instantiates an on-chip stationary weight matrix for 52, 000 entries of

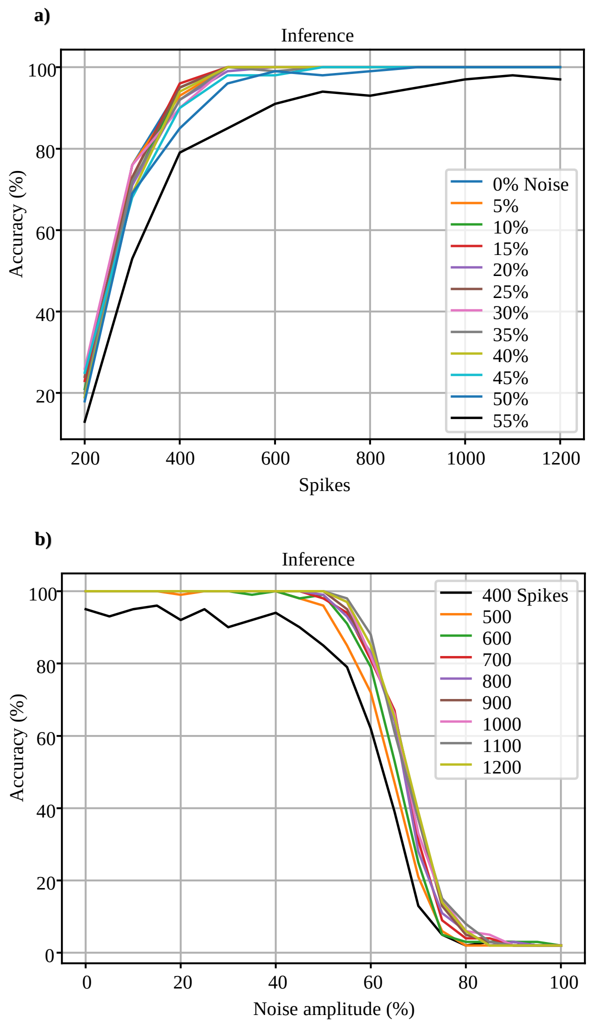

Figure 3.12.: Noise tolerance on hardware PU with standard floating-point (IEEE 754) computation (benchmark/reference), (a) exhibits accuracy degradation applying 50% of noise amplitude, and (b) illustrates convergence of inference with 400 spikes.

5-bit. The available memory size is large enough to store $W \in \mathbb{R}^{5\times5\times2\times32}$ and $W \in \mathbb{R}^{5\times5\times32\times64}$ for *H1_CONV* and *H3_CONV*, respectively. The same dot-product architecture is implemented in the processing unit of the fully connected layer (*FC*). However, due to lack of BRAM resources, this PU can not instantiate on-chip stationary synaptic weight matrix. Instead, *FC* receives the $\vec{W}(s_t)$ (weight vectors) during operation as well as $\vec{h}_\mu$ and S_t. The hardware mapping and the computation schedule of this implementation are displayed in **Tab.** 3.6 and **Fig.** 3.14.

As shown in the computation schedule in **Tab.** 3.6 and **Fig.** 3.14, this implementation presents a maximum hardware PU latency of 1.30 ms according to **Eq.** (3.9), and CPU latency of 1.67 ms. Therefore, applying **Eq.** (3.10), the total latency is 1.67 ms per spike cycle as shown in **Fig.** 3.14. In this case, the cyclic bottleneck in each SbS spike is in the CPU performance.

This configuration achieves an accuracy of 98.97% correct classification on the 10,000 image test set with 1000 spikes. This indicates an accuracy degradation of 0.33%. To monitoring output quality, the noise tolerance plot in **Fig.** 3.15 revels accuracy degradation for noise higher than 50% on the input images, and convergence of inference with 400 spikes. Thus, the

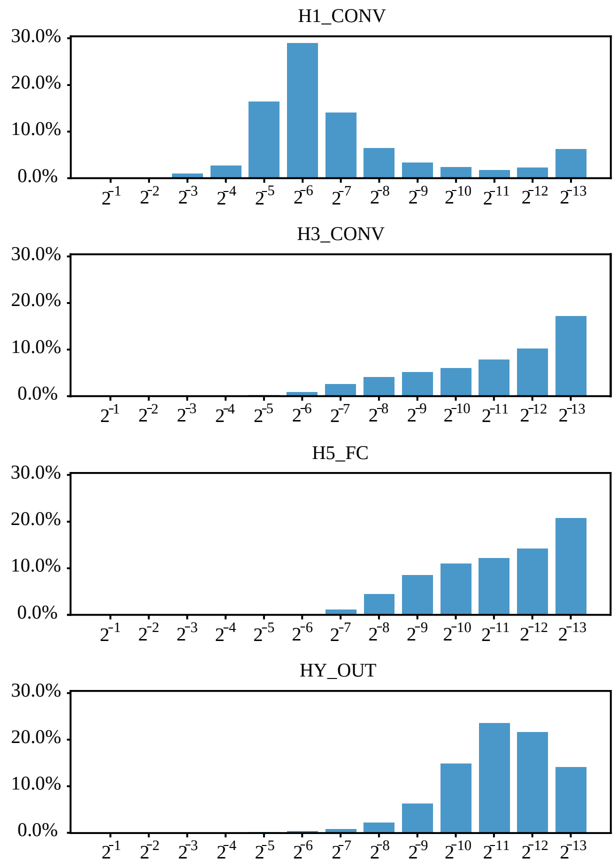

Figure 3.13.: $\log_2$-histogram of each synaptic weight matrix showing the percentage of matrix elements with given integer exponent.

particular SbS network implementation under approximate processing effects demonstrates a minimal impact on the overall accuracy. This reveals an inherent error resilience, and hence, remaining approximation budget.

The post-implementation resource utilization and power dissipation of this design are shown in **Tab.** 3.5.

Table 3.5.: Resource utilization and power dissipation of processing units with hybrid custom floating-point approximation.

PU	LUT	FF	DSP	BRAM 18K	Power (mW)
Conv	3,139	4,850	19	25	82
FC	3,265	5,188	8	9	66

Design Exploration for Dot-Product whit Hybrid Logarithmic Computation

For this design, 4-bit integer exponent are used for logarithmic representation of the synaptic weight matrix. Each *Conv* processing unit implements the proposed dot-product architecture including an on-chip stationary weight matrix for 52, 000 entries of 4-bit integer each one to store $W \in \mathbb{N}^{5\times5\times2\times32}$ and $W \in \mathbb{N}^{5\times5\times32\times64}$ for *H1_CONV* and *H3_CONV*, respectively. The same

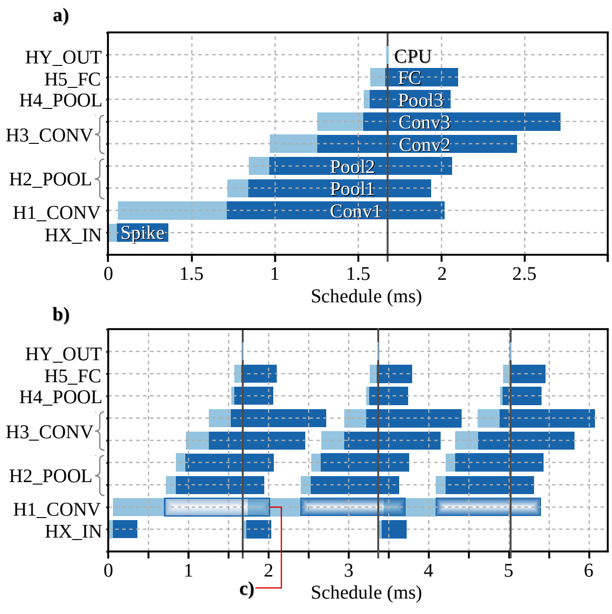

Figure 3.14.: Performance on processing units with hybrid custom floating-point approximation, (a) exhibits computation schedule, (b) presents cyclic computation schedule, and (c) shows the performance of *Conv2* from a previous computation cycle during the preprocessing of *H1_CONV* on the current computation cycle without bottleneck.

Table 3.6.: Performance of hardware processing units with hybrid custom floating-point approximation.

Hardware mapping		**Computation schedule (ms)**			
Layer	**PU**	t_s	t_{CPU}	t_{PU}	t_f
HX_IN	Spike	0	0.055	0.307	0.362
H1_CONV	Conv1	0.057	0.654	1.309	2.020
H2_POOL	Pool1	0.713	0.131	1.098	1.942
	Pool2	0.845	0.125	1.098	2.068
H3_CONV	Conv2	0.972	0.285	1.199	2.456
	Conv3	1.258	0.279	1.184	2.721
H4_POOL	Pool3	1.538	0.037	0.484	2.059
H5_FC	FC	1.577	0.091	0.438	2.106
HY_OUT	CPU	1.669	0.004	0	1.673

Table 3.7.: Performance of hardware processing units with hybrid logarithmic approximation.

Hardware mapping		**Computation schedule (ms)**			
Layer	**PU**	t_s	t_{CPU}	t_{PU}	t_f
HX_IN	Spike	0	0.055	0.264	0.319
H1_CONV	Conv1	0.057	0.655	1.271	1.983
H2_POOL	Pool1	0.714	0.130	1.074	1.918
	Pool2	0.845	0.126	1.106	2.077
H3_CONV	Conv2	0.973	0.285	1.179	2.437
	Conv3	1.258	0.278	1.176	2.712
H4_POOL	Pool3	1.538	0.037	0.488	2.063
H5_FC	FC	1.577	0.091	0.388	2.056
HY_OUT	CPU	1.669	0.004	0	1.673

dot-product architecture is implemented in the *FC* processing unit without stationary synaptic weight matrix. The hardware assignment and the computation schedule of this implementation are displayed in **Tab.** 3.7 and **Fig.** 3.16.

As shown in the computation schedule in **Tab.** 3.7 and **Fig.** 3.16, this implementation presents a maximum hardware PU latency of 1.27 ms (according to **Eq.** (3.9)), and CPU latency of 1.67 ms. Therefore, applying **Eq.** (3.10), gives 1.67 ms as latency per spike cycle as shown in **Fig.** 3.16. In this case, the cyclic bottleneck is in the CPU performance.

This quality configuration achieves an accuracy of 98.84% correct classification on the 10,000 image test set with 1000 spikes. This indicates an accuracy degradation of 0.46%. To monitor output quality, the noise tolerance plot in **Fig.** 3.17 revels accuracy degradation having 40% noise on the input images, and convergence of inference with 600 spikes. The particular SbS network implementation under approximate processing demonstrates a minor impact on the overall accuracy. As the most efficient setup and yet the worst-case quality configuration, this exhibits remaining budget for further approximate processing approaches.

The post-implementation resource utilization and power dissipation are shown in **Tab.** 3.8.

Table 3.8.: Resource utilization and power dissipation of processing units with hybrid logarithmic approximation.

PU	LUT	FF	DSP	BRAM 18K	Power (mW)
Conv	3,086	4,804	19	21	78
FC	3,046	4,873	8	8	66

3.3.3. Results and Discussion

As benchmark, the SbS network inference on embedded CPU using standard 32-bit floating-point achieves an accuracy of 99.3% with a latency of $T_{SC} = 34.28ms$. As a second reference point,

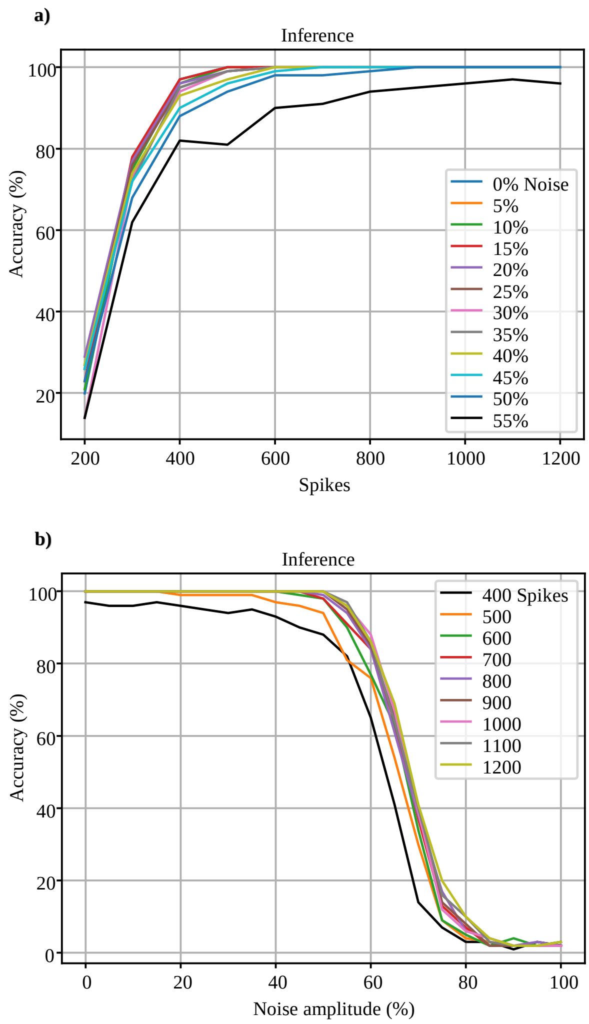

Figure 3.15.: Noise tolerance on hardware PU with custom floating-point approximation, (a) exhibits accuracy degradation applying 50% of noise amplitude, and (b) illustrates convergence of inference with 400 spikes.

the network simulation on hardware processing units with standard floating-point achieves an accuracy of 98.98% with a latency $T_{SC} = 3.18ms$. As result, this design get 10.7× latency enhancement and an accuracy degradation of 0.32%. The tolerance plot in **Fig.** 3.12 reveals accuracy degradation having 50% noise on the input images, and convergence of inference with 400 spikes. In this case, the SbS network deployment with precise computing proves extraordinary inherent error resilience, and hence, this represents a great potential for approximate processing.

As a demonstration of the proposed dot-product architecture, the SbS network inference on hardware PUs with synaptic representation using 5-bit custom floating-point (4-bit exponent, 1-bit mantissa) and 4-bit logarithmic (4-bit exponent) achieve 20.5× latency enhancement and accuracy of 98.97% and 98.84%, respectively. This results in accuracy degradation of 0.33% and 0.46%, respectively. To monitor output quality, the noise tolerance plot in **Fig.** 3.15 and **Fig.** 3.17 reveal accuracy degradation when having 50% and 40% noise on the input images, and convergence of inference with 400 and 600 spikes, respectively. Therefore, the design exploration under the proposed approximate computing approach indicates sufficient inherent error resilience for further or more aggressive approximation approaches.

Regarding resource utilization and power dissipation with the proposed approach, *Conv* processing units have a 43.24% reduction of BRAM, and 12.35% of improvement in energy efficiency over the standard floating-point implementation. However, the proposed approach

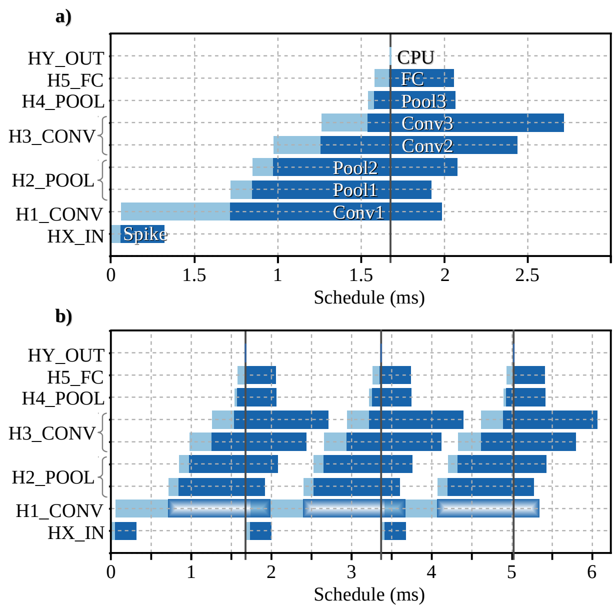

Figure 3.16.: Performance of processing units with hybrid logarithmic approximation, (a) exhibits computation schedule, and (b) illustrates cyclic computation schedule.

does not reuse the available floating-point operator cores instantiated from other computational sections (see **Tab.** 4.3). Therefore, the logic required for the dot-product must be implemented, which is reflected as additional utilization of Look-up Table (LUT) and Flip-Flop (FF) resources. The experimental results of the design exploration are summarized in **Tab.** 3.9. The platform implementations are summarized in **Tab.** 3.10, and their power dissipation breakdowns are presented in **Fig.** 3.18.

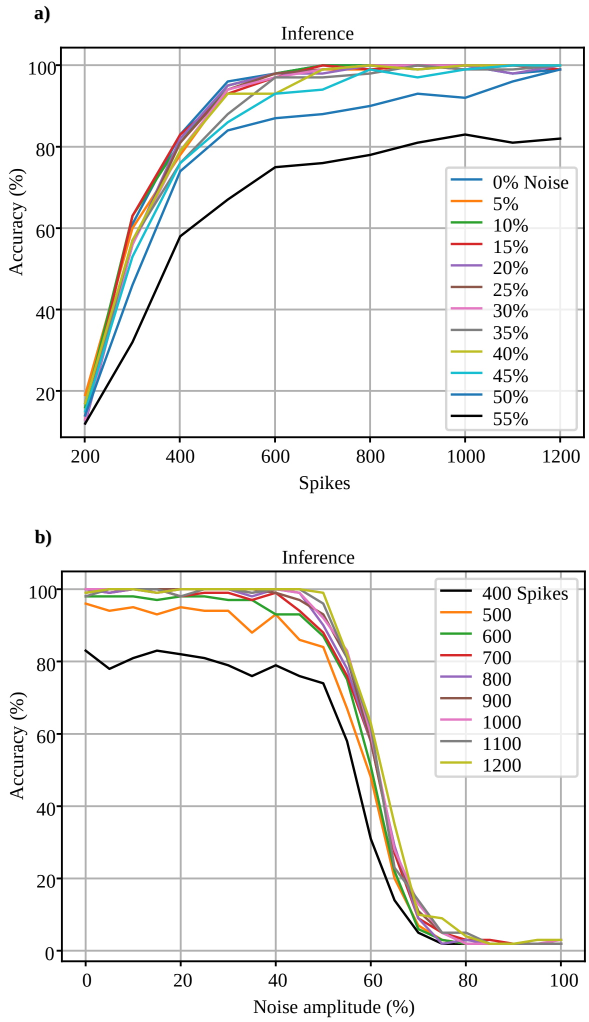

Figure 3.17.: Noise tolerance on hardware PU with hybrid logarithmic approximation, (a) exhibits accuracy degradation applying 40% of noise amplitude, (b) illustrates convergence of inference with 600 spikes.

Table 3.9.: Experimental results.

Dot-product	PU	Post-implementation resource utilization				Power (mW)	Latency		Accuracy (%)[e]	
		LUT	FF	DSP	BRAM 18K		(ms)	Gain[d]	Noise 0%	50%
Standard FP[a]	Conv	2,765	4,366	19	37	89	3.183	10.77x	98.98	98.63
	FC	2,649	4,189	8	9	66				
Hybrid custom FP[b]	Conv	3,139	4,850	19	25	82	1.673	20.49x	98.97	98.47
	FC	3,265	5,188	8	9	66				
Hybrid log[c]	Conv	3,086	4,804	19	21	78	1.673	20.49x	98.84	95.22
	FC	3,046	4,873	8	8	66				

[a] Reference with standard floating-point arithmetic (IEEE 754).
[b] Synaptic weight with number representation composed of 4-bit exponent and 1-bit mantissa.
[c] Synaptic weight with number representation composed of 4-bit exponent.
[d] Acceleration with respect to the computation on embedded CPU (ARM Cortex-A9 at 666 MHz) with latency $T_{SC} = 34.28ms$.
[e] Accuracy on 10,000 image test set with 1000 spikes.

Table 3.10.: Platform implementations.

Platform implementation	Post-implementation resource utilization				Power (W)	Clk (MHz)	Latency		Accu (%)[f]
	LUT	FF	DSP	BRAM 18K			(ms)	Gain[e]	
[45][a]	42,740	57,118	49	92	2.519	250	4.65	7.4x	99.02
This work (standard FP)[b]	39,514	56,036	82	180	2.420	200	3.18	10.7x	98.98
This work (hybrid custom FP)[c]	42,021	58,759	82	156	2.369	200	1.67	20.5x	98.97
This work (hybrid log)[d]	41,060	57,862	82	148	2.324	200	1.67	20.5x	98.84

[a] Reference architecture with homogeneous AUs using standard floating-point arithmetic (IEEE 754).
[b] Reference architecture with specialized heterogeneous PUs using standard floating-point arithmetic (IEEE 754).
[c] Proposed architecture with specialized heterogeneous PUs using synaptic weight with number representation composed of 4-bit exponent and 1-bit mantissa.
[d] Proposed architecture with specialized heterogeneous PUs using synaptic weight with number representation composed of 4-bit exponent.
[e] Acceleration with respect to the computation on embedded CPU (ARM Cortex-A9 at 666 MHz) with latency $T_{SC} = 34.28ms$.
[f] Accuracy on 10,000 image test set with 1000 spikes.

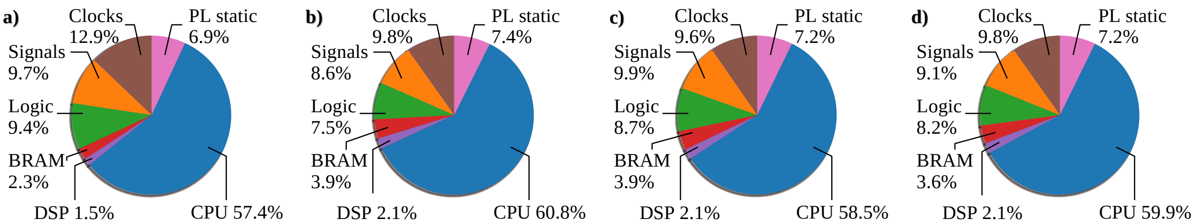

Figure 3.18.: Power dissipation breakdown of platform implementations, (a) [45] architecture with homogeneous AUs using standard floating-point arithmetic (IEEE 754), (b) reference architecture with specialized heterogeneous PUs using standard floating-point arithmetic (IEEE 754), (c) proposed architecture with hybrid custom floating-point approximation, and (d) proposed architecture with hybrid logarithmic approximation.

3.4. Conclusions

This chapter presents an accelerator for SbS neural networks with a dot-product functional unit based on approximate computing that combines the advantages of custom floating-point and logarithmic representations. This approach reduces computational latency, memory footprint, and power dissipation while preserving accuracy. For output quality monitoring, noise tolerance plots are proposed as an intuitive visual measure to provide insights into the accuracy degradation of SbS networks under different approximate processing effects. This plot revels inherent error resilience, hence, the possibilities for approximate processing.

The proposed approach is demonstrated with a design exploration flow on a Xilinx Zynq-7020 with a deployment of SbS network for MNIST classification task. This implementation achieves up to 20.5× latency enhancement, 8× weight memory footprint reduction, and 12.35% of energy efficiency improvement over the standard floating-point hardware implementation, this deployment incurs in less than 0.5% of accuracy degradation. Furthermore, with noise amplitude of 50% added on the input images, the SbS network presents an accuracy degradation of less than 5%. To monitor the inference quality, the resulting noise tolerance plots demonstrate a sufficient QoR for minimal impact on the overall accuracy of the neural network under the effects of this approximation technique. These results suggest available room for further or more aggressive approximate processing approaches.

In summary, based on the relaxed need for fully accurate or deterministic computation of neural networks, approximate computing techniques allow substantial enhancement in processing efficiency with moderated accuracy degradation.

4. Low-Power Conv2D Tensor Accelerator: Hybrid 6-bit Floating-Point Computation

4.1. Introduction

There is a growing demand for sensor analytics based on ML algorithms. Industry 4.0 and smart city infrastructure leverage AI solutions to increase productivity and adaptability [118]. These solutions are powered by advances in ML, compute engines, and big data. Therefore, enhancement of these should be considered for research, as they are the machinery of the future.

CNNs represent the essential building blocks in 2D pattern analytics. Sensor-based applications such as mechanical fault diagnosis [46, 47], structural health monitoring [48], human activity recognition [49], hazardous gas detection [50] have been powered by CNN models in industry and academia. CNN-based models, as one of the main types of ANN, have been widely used in sensor analytics with automatic learning from sensor data [119, 120, 121, 122]. In this context, CNN models are applied for automatic feature learning, usually, from 1D time series as well as for 2D time-frequency spectrograms. CNN models provide advantages such as local dependency, scale invariance, and noise resilience in analytics [22]. However, CNN models are computationally intensive and power-hungry. This is particularly challenging for low-power embedded applications, such as in the IoT field.

For ML inference, dedicated hardware architectures are typically used to enhance compute performance and power efficiency. In terms of computational throughput, GPUs offer the highest performance; in terms of power efficiency, ASIC and FPGA solutions are more energy efficient [123]. As a result, numerous commercial ASIC and FPGA accelerators have been proposed, targeting both HPC for data-centers and embedded systems applications.

However, most FPGA accelerators have been implemented to target mid- to high-range FPGAs for computationally intensive CNN models such as AlexNet, VGG-16, and ResNet-18. The main drawbacks of these implementations are power supply demands, physical dimensions, heat sink requirements, air cooling, and high price. In some cases, these implementations are not feasible for ubiquitous low-power/resource-constrained applications.

To reduce hardware there are two types of research [104]: the first one is deep compression including weight pruning, weight quantization, and compression storage [9, 124]; the second type of research corresponds to a more efficient data representation, also known as custom quantization for dedicated hardware implementation. In this group, hardware implementations with customized 8-bit floating-point computation have been proposed [103, 104, 102]. However, these architectures are inadequate for embedded applications, the target devices are high-end FPGAs and PCIe devices.

Reducing the compute hardware with more aggressive quantization such as binary [8], ternary [29], and mixed precision (2-bit activations and ternary weights) [30] typically incur significant accuracy degradation for very low precisions, especially for complex problems that require precision [31].

In this chapter, it is presented the Hybrid-Float6 quantization and its dedicated hardware design. In this approach, feature maps are represented by a standard FP number representation and trainable parameters by 6-bit FP. To preserve accuracy, a QAT method is proposed. For ML compatibility/portability, the 6-bit FP can be wrapped into the standard FP number representation. It is presented a parameterized tensor processor implementing a pipelined vector dot-product with HF6. The proposed hardware extracts the 6-bit representation automatically from the standard FP format and performs the computation. The 6-bit FP representation uses 4-bit

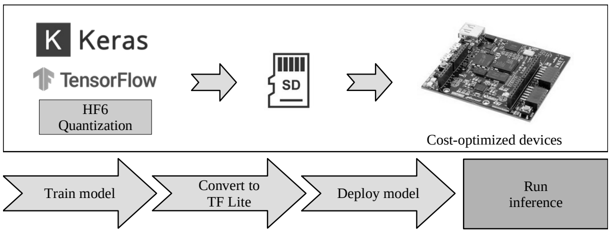

Figure 4.1.: The workflow of our approach on embedded FPGAs.

exponent and 1-bit mantissa. This approach enables optimizations in MAC design by reducing the mantissa multiplication to a multiplexer-adder operation. Moreover, the intrinsic error tolerance of ANN is leveraged to further reduce the hardware design with approximation. This approach reduces latency, resource utilization, and power dissipation. The embedded hardware/software architecture is integrated with TensorFlow Lite using delegate interface to accelerate *Conv2D* tensor operations. We evaluate the applicability of our approach with a CNN-regression model and hardware design exploration for sensor analytics of SHM for anomaly localization. The embedded hardware/software framework is demonstrated on XC7Z007S, this is the smallest and most inexpensive Zynq SoC device, see **Fig.** 4.1. To the best of my knowledge, this is the first research addressing 6-bit floating-point quantization on CNN models and its dedicated hardware design.

The main contributions presented in this chapter are as follows:

1. The Hybrid-Float6 quantization and its dedicated hardware design. It is proposed an optimized hardware MAC by reducing the mantissa multiplication to a multiplexer-adder operation. The intrinsic error tolerance of ANN is exploited to further reduce the hardware design with approximation. To preserve model accuracy, it is presented a quantization-aware training method, which provides regularization effects.

2. A custom hardware/software co-design framework for low-power analytics on resource-constrained embedded FPGA. TensorFlow Lite micro is integrated in this framework.

3. A customizable tensor processor as a dedicated hardware for HF6. This design computes *Conv2D* tensor operations employing a pipelined vector dot-product with parametrized on-chip memory utilization. For exploration purposes, the compute engine can be synthesized with the proposed HF6 hardware or with Xilinx LogiCORE IPs (for standard floating-point).

4. The potential of this approach is demonstrated with a CNN-regression model for anomaly localization in SHM based on AE. A hardware design exploration is presented evaluat-

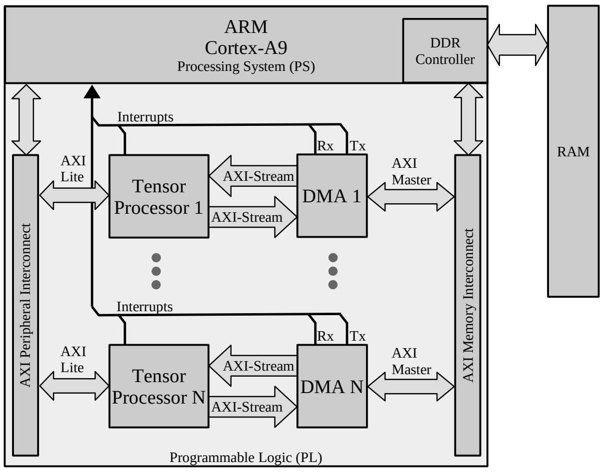

Figure 4.2.: Base embedded system architecture.

ing inference accuracy, compute performance, hardware resource utilization, and energy consumption.

This work is available to the community as an open-source project at https://github.com/YaribNevarez/tensorflow-lite-fpga-delegate.git.

4.2. System Design

The system design is a hardware/software co-design framework for low-power ML analytics. This architecture allows design exploration for dedicated hardware in embedded systems. For ML compatibility, the proposed framework integrates TensorFlow Lite micro.

4.2.1. Base embedded system architecture

The embedded system architecture consists of a cooperative hardware-software platform. See **Fig.** 4.2. The embedded CPU delegates low-level compute-bound tensor operations to the TPs. The TPs employ AXI-Lite interface for configuration and AXI-Stream interfaces via DMA for data movement from off-chip memory. Each TP and DMA pair asserts interrupt flags once its compute job/transaction completes. Interrupt events are handled by the embedded CPU to use the results and to start a new compute job/transaction. The hardware architecture can vary its resource utilization by customizing the TPs prior to the hardware synthesis.

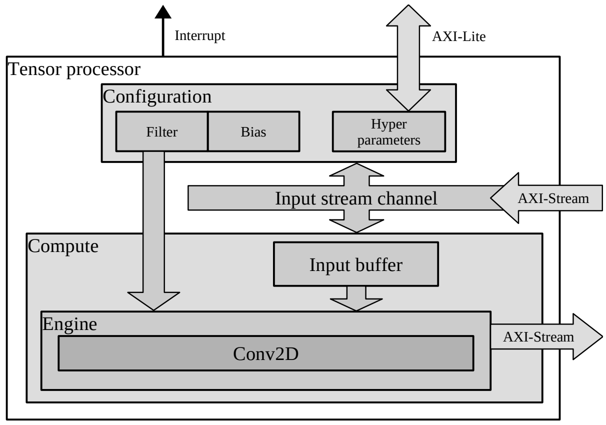

Figure 4.3.: High level hardware architecture of the proposed tensor processor.

4.2.2. Tensor processor

The TP is a dedicated hardware module to compute tensor operations. This implements high performance communication with AXI-Stream, direct CPU communication with AXI-Lite, and on-chip storage utilizing BRAM. This hardware architecture is implemented with HLS. The tensor operations are implemented based on the C++ TensorFlow Lite micro kernels. See **Fig.** 4.3. This research focuses on the *Conv2D* tensor operation that computes 2D convolution layers.

Modes of operation

The TP has two modes of operation: *configuration* and *execution*.

- In *configuration* mode, the TP receives the hyperparameters of the tensor operation: stride, dilation, padding, offset, activation, depth-multiplier, input shape, filter shape, bias shape, and output shape. Afterwards, in the same data stream, the TP receives filter and bias tensors. These are locally stored in BRAM for data re-usage. The filter and bias tensors are transferred using standard FP format wrapping the 6-bit FP representation, which is extracted by the TP for local on-chip storage.

- In *execution* mode, the TP executes the tensor operation according to the hyperparameters given in the configuration mode. During execution, the input and output tensors are moved via DMA.

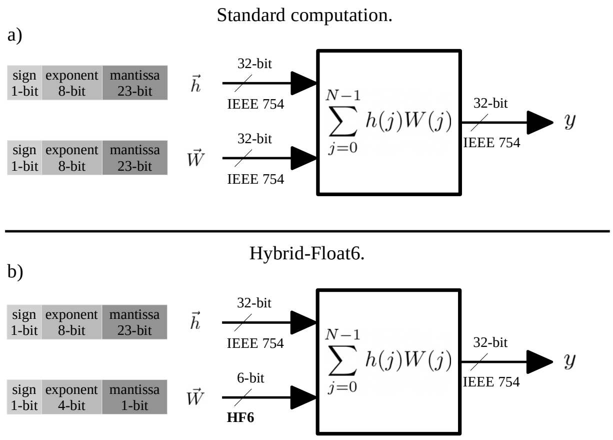

Figure 4.4.: Dot-product hardware module with (a) standard floating-point and (b) Hybrid-Float6.

Dot-product with hybrid floating-point

It is implemented the floating-point computation adopting the dot-product with hybrid custom floating-point [51]. The hardware dot-product is illustrated in **Fig.** 4.4 and **Fig.** 4.5(a). This design instantiates a HF6 MAC with an internal accumulator register of 64-bit fixed-point with 23-bit fraction. During operation, the feature map and filter values are extracted from on-chip memory (BRAM). Both values have to be different than zero to enable the MAC operation. The result is biased by accumulating a denormalized bias value. Since the bias is stored with 6-bit FP, its fractional part has to be aligned with the 23-bit fraction of the accumulator, see **Fig.** 4.5(b). The ReLu activation is applied to the accumulator and its result is normalized to convert it to IEEE 754 standard FP, see **Fig.** 4.5(c).

Rather than a parallelized hardware structure, this approach is a pipelined hardware design suitable for resource-limited devices. The latency in clock cycles of this hardware module is defined by **Eq.** (4.1), where N is the length for the vector dot-product. This latency equation is obtained from the general pipelined hardware latency formula: $L = (N-1)\,II + IL$, where II is the initiation interval, and IL is the iteration latency. Both II and IL are obtained from the HLS results. Both the exponent and mantissa bit widths of the filter and bias are set to 4-bit exponent and 1-bit mantissa (E4M1), which corresponds to float6 quantization.

$$L_{hf} = N + 7 \tag{4.1}$$

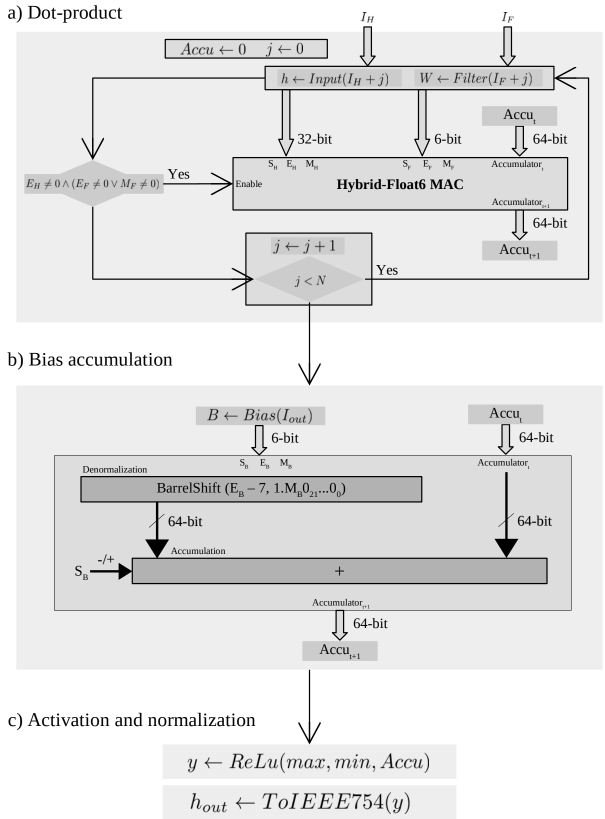

Figure 4.5.: (a) Dot-product hardware module with Hybrid-Float6 MAC, (b) bias accumulation, (c) activation and normalization to IEEE754.

Multiply-Accumulate

The multiply-accumulate operation calculates the product of two numbers and adds the result to an accumulator. In FP arithmetics, the area of a hardware multiplier scales with the bit size of the mantissas. In the case of HF6, the 6-bit FP representation allows a reduced hardware multiplicator for mantissas. The 1-bit mantissa enables optimized MAC implementations by reducing the mantissa multiplication to a multiplexed addition, see **Fig.** 4.6. This MAC produces denormalized results, which are accumulated in a fixed-point accumulator. This approach reduces latency, energy consumption, and hardware area/resource utilization.

Special cases, such as Infinity and NaN, are not considered in this design for simplicity, since they are not expected for CNN inference. For the subnormal case, the element-wise multiplication is disabled when having a zero entry and approximated when having subnormal mantissa. The feature map values are considered zero when the exponent is zero ($E_H = 0$). The filter values are considered zero when both exponent and mantissa are zero ($E_F = 0 \wedge M_F = 0$). See **Fig.** 4.5(a). In the 6-bit FP, the 1-bit mantissa has one subnormal case, which is handled as a normalized

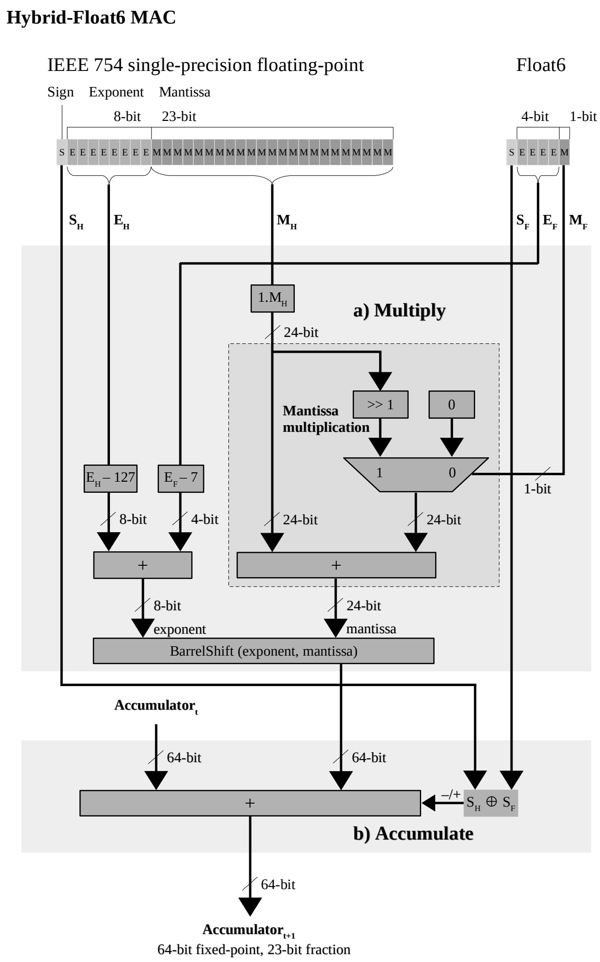

Figure 4.6.: Hybrid-Float6 multiply-accumulate hardware design.

case. This exploits the intrinsic error tolerance to reduce the hardware design.

The approximation error is defined by the difference between **Eq.** (2.10) and **Eq.** (2.12) when $E = 0$ and $M = 2^{-1}$. The result defines the error as $e = 2^{-B-1}$. Then, from **Eq.** (2.11) with $E_{size} = 4$, gives $B = 7$. Hence, $e = 3.9\text{e}{-3}$. This error is produced when having the subnormal case $E = 0$ and $M = 2^{-1}$, which corresponds to the value $\pm 7.8\text{e}{-3}$ deviated to $\pm 1.17\text{e}{-2}$. This approximation leverages the intrinsic error tolerance of CNN to reduce hardware resource utilization and energy consumption [22].

On-chip memory utilization

The total on-chip memory utilization on the TP is defined by **Eq.** (4.2), where TP_B and V_M represent the tensor buffers required for *Conv* operation and local registers required for the logic, respectively. **Eq.** (4.3) defines the tensor buffers, where $Input_M$ is the *input buffer*, $Filter_M$ is the *filter buffer*, $Bias_M$ is the *bias buffer*. These on-chip memory buffers are defined in bits.

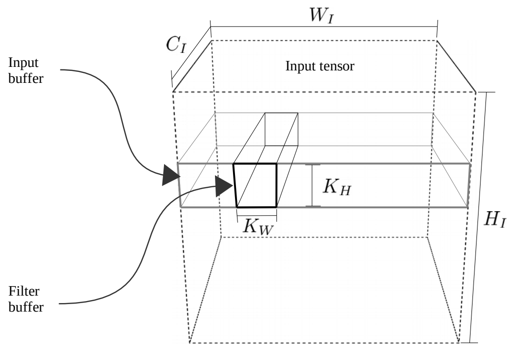

Figure 4.7.: Design parameters for on-chip memory buffers on the TP.

Fig. 4.7 illustrates the convolution operation utilizing the on-chip memory buffers.

$$TP_M = TP_B + V_M \tag{4.2}$$

$$TP_B = Input_M + Filter_M + Bias_M \tag{4.3}$$

The memory utilization of *input buffer* is defined by **Eq.** (4.4), where K_H is the height of the convolution kernel, W_I is the width of the input tensor (input feature maps), C_I is the number of input channels, and $BitSize_I$ is the bit size representation used by the input tensor.

$$Input_M = K_H W_I C_I BitSize_I \tag{4.4}$$

The memory utilization of *filter buffer* is defined by **Eq.** (4.5), where K_W and K_H are the width and height of the convolution kernel, respectively; C_I and C_O are the number of input and output channels, respectively; and $BitSize_F$ is the bit size representation used by filter values.

$$Filter_M = C_I K_W K_H C_O BitSize_F \tag{4.5}$$

The memory utilization of *bias buffer* is defined by **Eq.** (4.6), where C_O is the number of output channels, and $BitSize_B$ is the bit size representation used by bias values.

$$Bias_M = C_O BitSize_B \tag{4.6}$$

As a design trade-off, **Eq.** (4.7) defines the capacity of output channels based on given design parameters. The total on-chip memory TP_M determines the TP storage capacity.

$$C_O = \frac{TP_M - V_M - K_H W_I C_I BitSize_I}{C_I K_W K_H BitSize_F + BitSize_B} \tag{4.7}$$

The floating-point formats implemented in the TP are defined by $BitSize_F$, $BitSize_B$ and $BitSize_I$. The HF6 defines 6-bit for $BitSize_F$ and $BitSize_B$, and 32-bit for $BitSize_I$. These are design parameters defined before hardware synthesis. This allows fine control of BRAM utilization, which is suitable for resource-limited devices.

4.2.3. Training Method

The training method consists of two separate stages: (1) training with iterative early stop and (2) quantization-aware training.

Training with Iterative Early Stop

To achieve better performance on CNN-regression models, it implemented a training procedure with iterative early stop cycle. This allows to reach better local minima. This process consists of four steps:

1. A model is obtained with an initial training with standard early stop monitoring.
2. The model is iteratively re-trained with standard early stop. This process iteratively restarts the moving averages of the optimizer to search for better local minima.
3. In case of a better local minimum, the model is saved and used as a base for subsequent search iterations, otherwise it is a discarded search.
4. The cyclic process stops automatically with a given number of searches without a better local minimum, this is denoted as stop patience. This allows to set a maximum number of unsuccessful search trials before the stop.

This method is described in **Algorithm** 2.

Quantization-Aware Training

The QAT method is integrated into the training process, this operates as a callback on each mini-batch update. The quantization is applied on the trainable parameters of convolution layers. This method is implemented on the ML framework (TensorFlow/Keras), see **Algorithm** 3.

The quantization method uses rounding strategy to reduce the FP representation. This maps the full precision FP values to the closest representable 6-bit FP values, see **Algorithm** 4. This method quantizes the filter and bias tensors of the convolution layers. The exponent bit size plays a more predominant influence on the model accuracy than the mantissa bit size. In [32], Lai et al. demonstrated that 4-bit exponent and X-bit mantissa is adequate and consistent across different networks (SqueezeNet, AlexNet, GoogLeNet, VGG-16). In this research, the FP representation with 4-bit exponent and 1-bit mantissa is investigated.

Algorithm 2: Training with iterative early stop cycle.

input: $MODEL$ as the input model.
input: D_{train} as the training data set.
input: D_{val} as the validation data set.
input: N_I as the stop patience for iterative training cycle.
input: N_E as the early stop patience (epochs) for training.
input: B_{size} as the mini-batch size.
output: $MODEL$ as the full-precision output model.

```
Train(MODEL, D_train, D_val, N_E, B_size)
mse_i ← Evaluate(MODEL, D_val) // Benchmark
n_I ← 0
while n_I < N_I do
   // Iterative early stop cycle
   Train(MODEL, D_train, D_val, N_E, B_size)
   mse_v ← Evaluate(MODEL, D_val)
   if mse_v < mse_i then
      Update(MODEL)
      mse_i ← mse_v
   else
      MODEL ← LoadPreviousWeights()
      n_I ← n_I + 1
   end if
end while
```

4.2.4. Embedded software architecture

The software architecture is a layered object-oriented application framework written in C++, see **Fig.** 4.8 and **Fig.** 4.9. Description of the software layers is as follows:

- *Application*: As the highest level of abstraction, this software layer implements the analytics application, this invokes the ML library.

Algorithm 3: OnMiniBatchUpdate_Callback.

input: $MODEL$ as the full-precision input model.
input: E_{size} as the target exponent bits size.
input: M_{size} as the target mantissa bits size.
input: D_{train} as the training data set.
input: D_{val} as the validation data set.
input: N_{ep} as the number of epochs.
input: B_{size} as the mini-batch size.
output: $MODEL$ as the quantized output model.

```
// Quantize
MODEL ← Algorithm 4(MODEL, E_size, M_size)
if 1 < epoch then
    // Update model after first epoch
    mse_v ← Evaluate(MODEL, D_val)
    if mse_v < mse_i then
        Update(MODEL)
        mse_i ← mse_v
    end if
end if
```

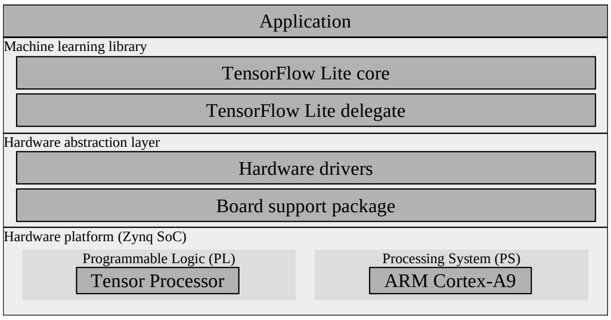

Figure 4.8.: High level embedded software architecture.

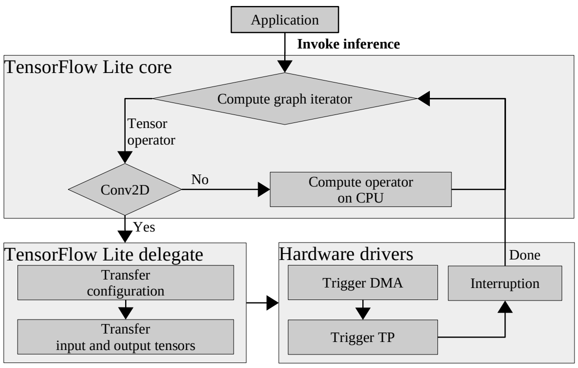

Figure 4.9.: Software flowchart.

Algorithm 4: Custom floating-point quantization.

input: $MODEL$ as the CNN.
input: E_{size} as the target exponent bit size.
input: M_{size} as the target mantissa bits size.
input: $STDM_{size}$ as the IEEE 754 mantissa bit size.
output: $MODEL$ as the quantized CNN.

```
for layer in MODEL do
  if layer is Conv2D or SeparableConv2D then
    filter, bias ← GetWeights(layer)
    for x in filter and bias do
      sign ← GetSign(x)
      exp ← GetExponent(x)
      fullexp ← 2^(E_size−1) − 1 // Get full range value
      cman ← GetCustomMantissa(x, M_size)
      leftman ← GetLeftoverMantissa(x, M_size)
      if exp < −fullexp then
        x ← 0
      else if exp > fullexp then
        x ← (−1)^sign · 2^fullexp · (1 + (1 − 2^(−Msize)))
      else
        if 2^(STDM_size−M_size−1) − 1 < leftman then
          cman ← cman + 1 // Above halfway
          if 2^(M_size) − 1 < cman then
            cman ← 0 // Correct mantissa overflow
            exp ← exp + 1
          end if
        end if
        // Build custom quantized floating-point value
        x ← (−1)^sign · 2^exp · (1 + cman · 2^(−M_size))
      end if
    end for
    SetWeights(layer, filter, bias)
  end if
end for
```

- *Machine learning library*: This software layer offers a comprehensive high level API for ML inference. This layer consist of TensorFlow Lite micro, this is modified to implement the delegate software interfaces for the proposed hardware accelerator.

- *Hardware abstraction layer*: This layer consist of the hardware drivers used in the TFLite delegate interfaces. This software layer handles initialization and runtime operation of the TP and DMA.

4.3. Experimental Results

This section presents experimental results using a low-power/low-cost sensor analytics application. A CNN-regression model is proposed to predict x- y- coordinates of acoustic emissions based on piezoelectric vibrations. Quantitative and qualitative aspects of the analytics are compared using floating-point 32-bit, fixed-point 8-bit, Hybrid-Logarithmic 6-bit, and Hybrid-Float6.

To demonstrate the proposed concept, the CNN model is deployed in the smallest Zynq SoC FPGA device for low-power inference. The performance of the TP synthesized with standard FP (using Xilinx LogiCORE IPs) and Hybrid-Float6 design.

4.3.1. Sensor Analytics Application

The analytics model is designed to predict x- y- coordinates of acoustic emissions on a metal plate. The metal plate is in the presence of noise disturbance to simulate realistic conditions. This subsection presents the structure for experimental setup, data sets, and the CNN-regression model.

Experimental Setup

The experiment uses eight piezoelectric sensors (Vallen Systeme VS900) attached with magnetic holders on a metal plate (90 cm × 86.6 cm × 0.3 cm). The VS900 devices can operate either in active or passive mode. Six VS900 are used in passive mode as acoustic sensors and two in active mode to produce acoustic emissions. These acoustic emissions simulate anomalies on x- y- coordinates as well as the noise disturbance on the system. See **Fig.** 4.10(a). To create data sets, the samples of acoustic emissions are labeled with their coordinates.

Data Sets

The data sets are recorded applying pulses on the metal plate, the x- y- coordinates of these pulses are used as labels. The pulses for training and validation data sets are shown in **Fig.** 4.10(b) and **Fig.** 4.10(c), respectively. The pulses for training and validation data sets are mutually exclusive, this exclusion is represented by the cross symbols in **Fig.** 4.10(c). This creates a grid layout used to collect samples for the data sets. This grid is 10 × 10 divisions, these are on the metal plate area (90 cm × 86.6 cm). This grid does not consider the four corners as they are used for magnetic holders.

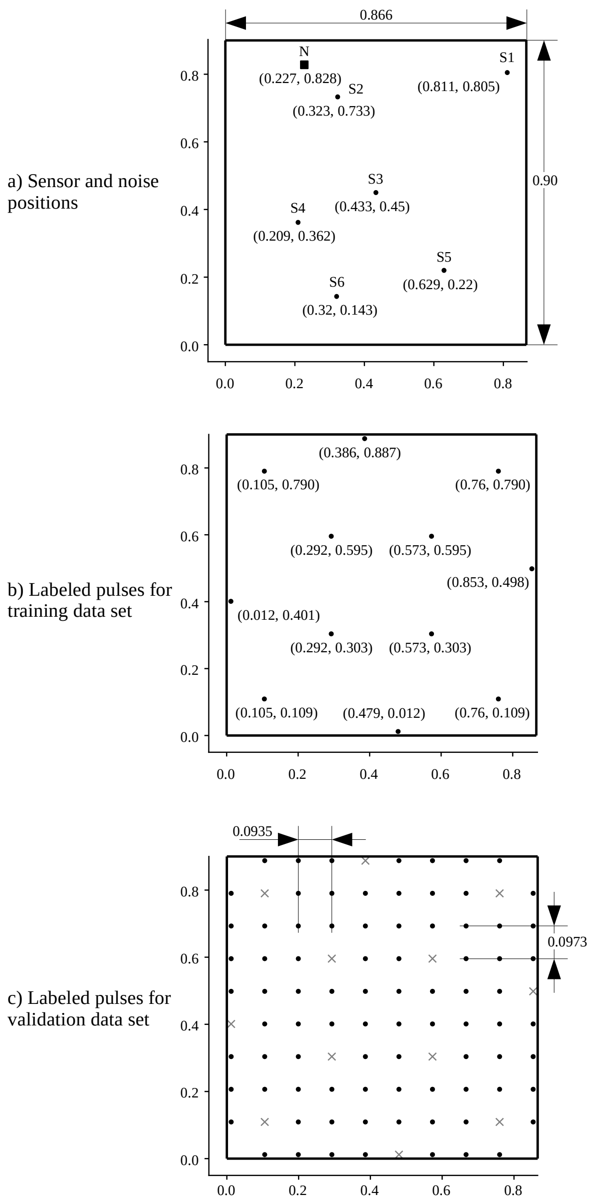

Figure 4.10.: Experimental setup for sensor analytics on structural health monitoring, all lengths are in meters (m).

In order to create reproducible acoustic emissions, this demonstration uses 9-cycle sine pulse in a Hanning window with central frequency f_c (narrow-banded in the frequency domain). This experiment assumes guided Lamb waves based on the plate structure. The narrow-band behavior also reduces the dispersion of the acoustic emission waves [125]. The waveform can be expressed as a function of time t as follows:

$$x_{\text{pulse}}(t) = \frac{1}{2}\left(1 - \cos \frac{f_c t}{5}\right) A_0 \sin f_c t. \tag{4.8}$$

To generate the data sets, slightly different pulse amplitudes and frequencies for excitation are used. The pulse frequency f_c is varied in 1 kHz steps between 300 kHz and 349 kHz and the amplitude A_0 is varied in 0.1 V steps between 2.6 V and 3.5 V. This produces 500 different pulses for each of the excitation points.

The signals for labeled pulses and noise disturbance are generated by Arbitrary Waveform Generators (AWGs). The sensor signals are recorded via a Vallen AMSY-6 measurement system with a resolution of 18 bits and a sampling rate of $f_S = 10\,\text{MHz}$. The disturbance signal is gaussian noise with amplitudes between 0-3 V. This noise is applied via the piezoelectric device N at $x = 0.227$ m and $y = 0.828$ m, see **Fig.** 4.10(a).

To obtain frequency components, the sampled pulses are converted into the frequency-time domain using the Short-Time Fourier Transform (STFT). This is calculated as follows [126]:

$$\mathcal{F}^{\gamma}_{m,k} = \sum_{n=0}^{N-1} x[n] \cdot \gamma^*[n - m\Delta M] \cdot \mathrm{e}^{\frac{-j2\pi kn}{N}} \tag{4.9}$$

Here $x[n]$ describes a discrete-time signal and $\gamma^*[n - m\Delta M] \cdot \mathrm{e}^{\frac{-j2\pi kn}{N}}$ the time- and frequency-shifted window function inside the considered interval $[0, N-1]$. ΔM describes the time shift and N the transformation window. Since only discrete frequencies and time points are considered, $m = 0, 1, ..., M - 1$ is valid. For pictorial representation, the magnitude of the complex-valued STFT is employed in a spectrogram $\mathcal{S}_{m,k}$:

$$\mathcal{S}_{m,k} = \left|\mathcal{F}^{\gamma}_{m,k}\right|^2 = \left|\sum_{n=0}^{N-1} x[n] \cdot \gamma^*[n - m\Delta M] \cdot \mathrm{e}^{\frac{-j2\pi kn}{N}}\right|^2 \tag{4.10}$$

In addition, these spectrograms are scaled in decibels. The spectrogram in decibels $\mathcal{S}_{m,k,\text{dB}}$ produces $\mathcal{S}_{m,k,\text{dB}} = 20 \cdot \log_{10}(\mathcal{S}_{m,k})$. For conversion of data, the experiment uses a signal length of 400 µs (75 µs pretrigger and 325 µs post trigger). Thus, the arrival times of the pulses are included in the spectrogram for all channels and labeled positions. This uses Blackman window function [127], Fast Fourier Transform (FFT) length of 32 samples, and overlap of 8 samples. The spectrograms are calculated for frequencies in the range of 100 kHz to 500 kHz. This produces a spectrogram size of 8x16 (8 frequency bins, 16 time values).

In order to generate larger data sets, four further variants are created with time shifts of 15 µs/ 30 µs/ 45 µs/ 60 µs. Subsequently, all spectrograms are converted to grayscale with scaling between -100 dB and -40 dB, see **Fig.** 4.11.

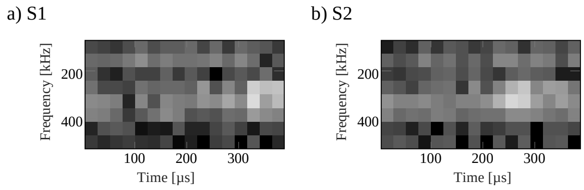

Figure 4.11.: Spectrograms of sensors S_1, S_2 converted to grayscale for pulses at $x = 0.105$ m, $y = 0.109$ m with noise disturbance.

In overall, the data set has a size of 1,440,000 images. This is the result of 500 (pulses) · 5 (spectrograms) · 6 (listening sensors) · 96 (excitation points).

CNN-Regression Model

The data analytics is implemented with a CNN-regression model, see **Fig.** 4.12. The structure of the model is described below:

a) Input tensor. This is composed of spectrograms from the sensor signals. The tensor shape is defined by $S \times T \times F$, where S is the number of sensors, and $T \times F$ is the time-frequency resolution of the spectrograms, see **Fig.** 4.12(a).

b) Feature extraction. This is composed of three blocks of convolution, batch normalization, and max-pooling layers, see **Fig.** 4.12(b). The number of channels in the convolution layers are defined by the hyper-parameters A, B, and C.

c) Regression function. This is an arbitrary function implemented with two fully connected layers and an output layer with linear activation, see **Fig.** 4.12(c).

4.3.2. Training

Base Model

The model in **Fig.** 4.12 is trained using Adam algorithm with iterative search. The Adam optimizer is configured with the default settings presented in [128]: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1\text{e}{-8}$. The training-cycle has a patience of 10 iterations before stop, the optimizer is executed with early stop patience of 10 epochs, and mini-batch size of 512 samples. This is applied using the method described in **Algorithm** 2 with $N_I = 10$, $N_E = 10$, $B_{size} = 512$.

The training results are illustrated in **Fig.** 4.13(a). In this optimization, the initial and the final models achieve $MSE = 0.0135\,\text{m}^2$ and $MSE = 0.0122\,\text{m}^2$, respectively. The MSE is

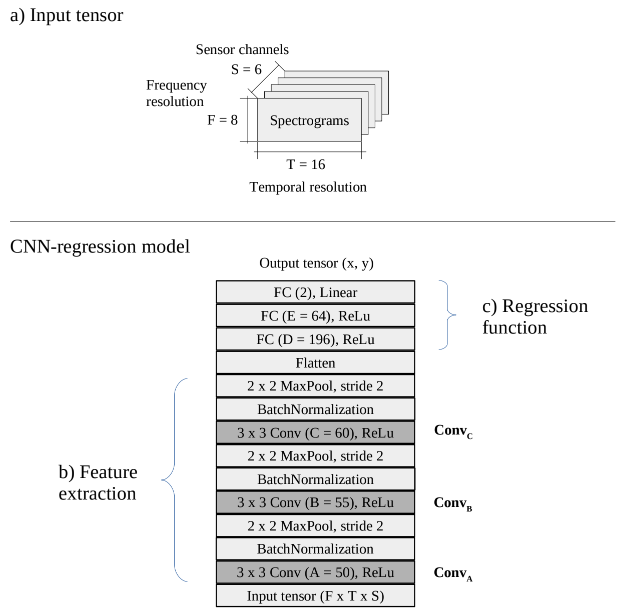

Figure 4.12.: CNN-regression model for sensor analytics.

calculated with the Euclidean distance (loss) between the real/expected and the predicted/inferred coordinates. The initial model is obtained at the first early stop (after 10 epochs). In each stop, the moving averages of the Adam optimizer get re-initialized. This facilitates searching for better local minima. The model gets saved/updated when finding a better minimum.

The final model achieves $MSE = 0.0122\,\mathrm{m}^2$, which corresponds to $MAE = 0.0955\,\mathrm{m}$. See **Fig.** 4.14(a). In total, the training takes 379 epochs in 25 cycle-search iterations. The first search takes 43 epochs for the initial model and subsequent search iterations take an average of 14 epochs. The total time is 53 minutes using a Personal Computer (PC) with AMD Ryzen 5 5600H and NVIDIA GeForce RTX 3050 GPU.

TensorFlow Lite 8-bit Quantization

This optimization method converts filter and bias tensors as well as activation maps to 8-bit integer representation, this allows inference using integer-only arithmetic [125]. In this research, this quantization is applied only to the convolution layers as they are the compute bound operations. Other layers employ 32-bit FP representation.

In the compute graph, the input and output feature maps are glued with linear quantization at the input and output of the *Conv2D* operations.

The base model is quantized using the TensorFlow Lite library with integer-only quantization on the *Conv2D* tensor operations. The filter and bias tensors are represented by 8-bit and 32-bit signed integers, respectively. The input and output activation maps are represented by 8-bit

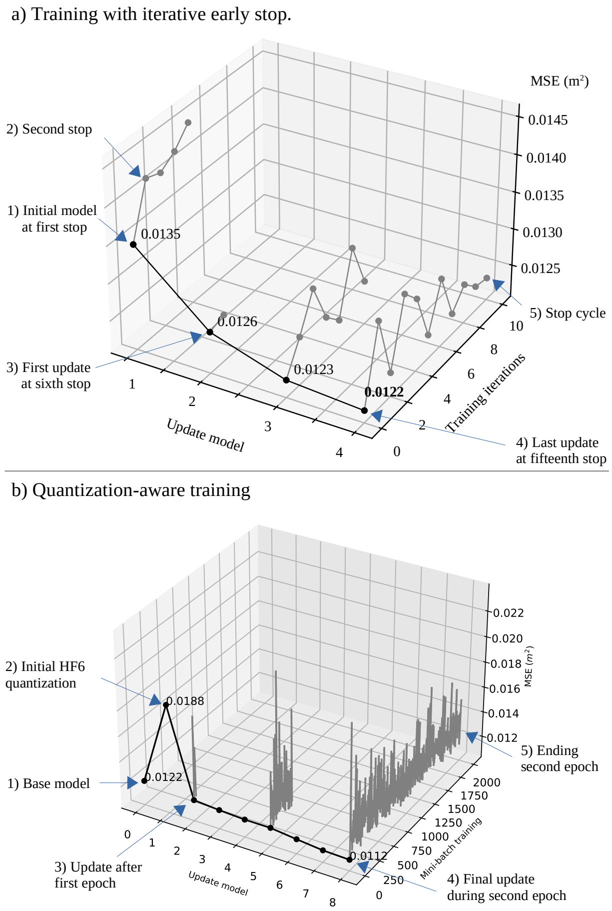

Figure 4.13.: Training results.

signed integer. The TensorFlow quantization includes two additional vectors (output-multiplier and output-shift coefficients), these two vectors are the same shape as the bias vector with 32-bit integer representation.

This model achieves $MSE = 0.0126\,\mathrm{m}^2$ and $MAE = 0.0992\,\mathrm{m}$. See **Fig.** 4.14(b). The MAE increases 5.1% of the base model. We attribute this degradation to the 8-bit quantization on the *Conv2D* layers.

Inference of non-quantized models on HF6 hardware

To demonstrate backward compatibility, the inference quality of the base model is measured without quantization on the HF6 hardware. See **Fig.** 4.14(c). This obtains $MSE = 0.0188\,\mathrm{m}^2$ and $MAE = 0.1232\,\mathrm{m}$. The MAE increases 29.5% of the base model. We attribute this degradation to the rounding errors of non-quantized filters and bias in *Conv2D* layers.

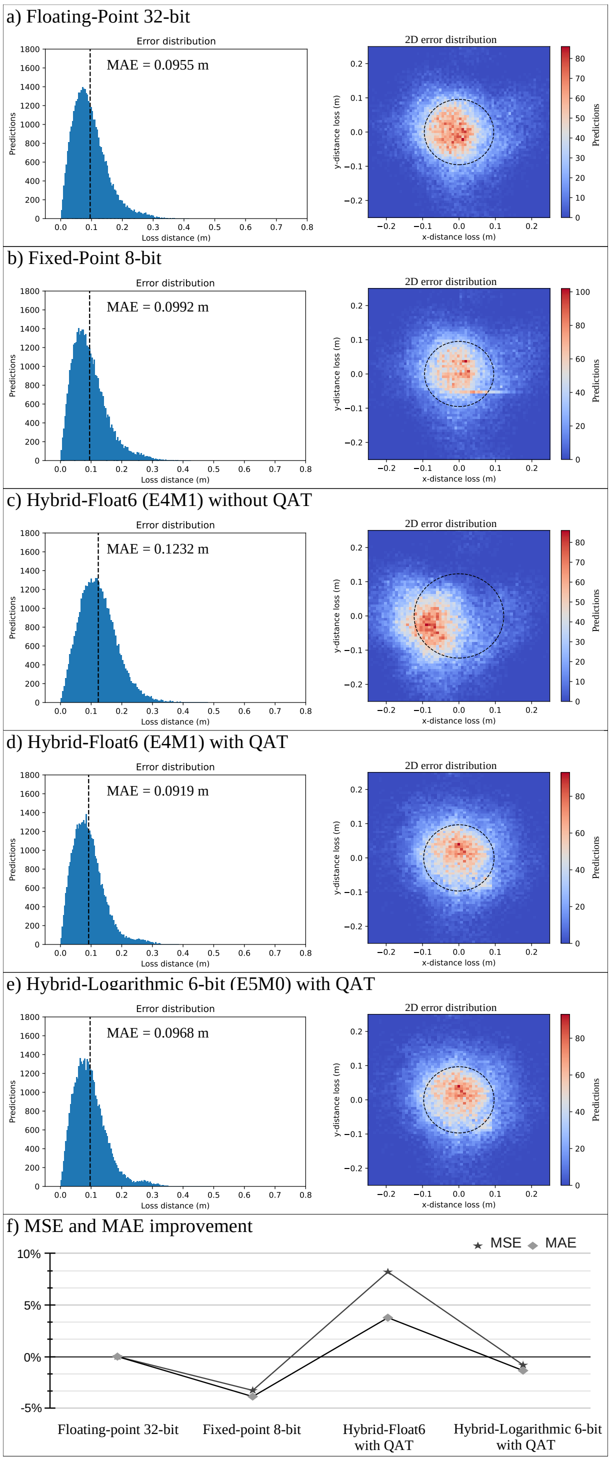

Figure 4.14.: Performance of the model with different data representations.

Quantization-Aware Training for HF6 hardware

The QAT is a post-training optimization. This has been run during two epochs with mini-batch size of 10 samples. This quantization is executed targeting the HF6 format: 4-bit exponent and 1-bit mantissa. This is applied to filter and bias tensors of *Conv2D* layers. This method is described in **Algorithm** 3 with $N_{ep} = 2$, $B_{size} = 10$, $E_{size} = 4$, $M_{size} = 1$. The optimization results are illustrated in **Fig.** 4.13(b).

The resulting model achieves $MSE = 0.0112\,\mathrm{m}^2$ and $MAE = 0.0919\,\mathrm{m}$. This corresponds to an error reduction of 8.2% and 3.77%, respectively. We attribute this improvement to the regularization effect. See **Fig.** 4.14(d). The QAT time is 185 minutes.

Quantization-Aware Training for Hybrid-Logarithmic 6-bit

For the sake of quality comparison with logarithmic quantization, the model with 6-bit logarithmic representation is generated. See **Fig.** 2.4(e). This quantization matches the bit size of HF6. The filter and bias tensors of *Conv2D* layers are quantized with the 6-bit logarithmic format: 1-bit sign, 5-bit signed exponent, and 0-bit mantissa. This is applied using the method described in **Algorithm** 3 with $N_{ep} = 2$, $B_{size} = 10$, $E_{size} = 5$, $M_{size} = 0$.

The model achieves $MSE = 0.0123\,\mathrm{m}^2$ and $MAE = 0.0968\,\mathrm{m}$, which correspond to an error increase of 0.82% and 1.36%, respectively. We attribute this degradation to the 6-bit logarithmic quantization lacking fractional bits. See **Fig.** 4.14(e).

A summary of improvement-degradation of MSE and MAE with different data representations is presented in **Fig.** 4.14(f).

4.3.3. Hardware Design Exploration

The proposed hardware/software co-design is demonstrated on the Zynq-7007S SoC on the MiniZed development board. This SoC integrates a single ARM Cortex-A9 PS and a PL equivalent to Xilinx Artix-7 FPGA in a single chip [116]. The Zynq-7007S SoC architecture maps the custom logic and software in the PL and PS, respectively.

In this platform, the proposed hardware/software architecture is implemented to deploy the sensor analytics application. The desired model is converted to TensorFlow Lite (floating-point) and deployed on the embedded software as a hex dump as a C array. The Zynq-7007S SoC executes inference with TensorFlow Lite on the PS. The computational workload of convolution layers is delegated to the dedicated hardware.

Benchmark on Embedded CPU

First, the performance of the embedded CPU is explored for inference without hardware acceleration. In this case, TensorFlow Lite creates the CNN model as a sequential compute graph executing all computation on the CPU (ARM Cortex-A9) at 666 MHz with power dissipation of 1, 187 W.

The compute performance and run-time inference of the CPU are shown in **Tab.** 4.2(a) and **Fig.** 4.16(a), respectively.

Benchmark on Tensor Processor Synthesized with Xilinx LogiCORE IP for Floating-Point Computation

For this design, the TP is implemented with standard Xilinx FP hardware prior synthesis. The design parameters for the maximum required accelerator on-chip size are:

- Max convolution kernel size: $K_W = K_H = 3$.
- Max input tensor width: $W_I = 16$.
- Max input and output channels: $C_I = 55$, $C_O = 60$.
- Filter and bias bit size: $BitSize_F = BitSize_B = 32$.
- Input tensor bit size: $BitSize_I = 32$.

Using equations from Section 4.2.2, the on-chip memory utilization are $Input_M = 84,480$b, $Filter_M = 950,400$b, and $Bias_M = 1,920$b. Hence, the required on-chip memory buffer size is $TP_B = 1,036,800$b.

The post-implementation resource utilization and power dissipation are presented in **Tab.** 4.1(a). The complete hardware platform utilizes 83% of BRAM, this includes the on-chip memory requirements of the TP, DMA, and AXI interconnects. The total available on-chip memory (BRAM) on the Zynq-7007S SoC is 1.8 Mb. After hardware syntheses, the estimated power dissipation of the TP is 85 mW at 200 MHz (this estimation is provided by Xilinx Vivado).

The compute performance and inference schedule of the model on this hardware implementation are shown in **Tab.** 4.2(b) and **Fig.** 4.16(b), respectively. During run-time, the software (TensorFlow Lite) delegates computation to the TP as dedicated hardware for *Conv2D* tensor operations.

The implementation of the dot-product with standard FP engine (IEEE 754 arithmetic) utilizes proprietary multiplier and adder floating-point operator cores. Vivado HLS implements FP

Table 4.1.: Resource utilization and power dissipation on the Zynq-7007S SoC.

TP engine	**Post-implementation resource utilization**				**Power (W)**
	LUT	**FF**	**DSP**	**BRAM 36Kb**	
(a) Floating-Point	5,578 39%	8,942 31%	23 35%	41.5 **83%**	1.429
(b) Hybrid-Float6	7,313 51%	10,330 36%	20 30%	15 **30%**	1.424

Table 4.2.: Compute performance of the CPU and TP on each Conv2D tensor operation. This table presents: tensor operation, computational cost in mega floating-point operations (MFLOP), latency, throughput, power efficiency, and estimated energy consumption as the energy delay product (EDP).

Operation	**MFLOP**	**t (ms)**	**MFLOP/s**	**MFLOP/s/W**	**EDP (mJ)**
		a) CPU (ARM Cortex-A9) @666MHz, 1.187 W			
Conv_A	0.691	112.24	6.16	5.19	133.23
Conv_B	1.584	213.13	7.43	6.26	252.99
Conv_C	0.475	46.59	10.20	8.59	55.31
		b) TP (Floating-Point engine) @200MHz, 85 mW			
Conv_A	0.691	12.49	55.34	651.11	1.06
Conv_B	1.584	16.39	96.66	1,137.20	1.39
Conv_C	0.475	3.59	132.44	1,558.13	0.30
		c) TP (Hybrid-Float6 engine) @200MHz, 84 mW			
Conv_A	0.691	6.92	99.81	1,188.24	0.58
Conv_B	1.584	4.41	358.94	4,273.09	0.37
Conv_C	0.475	0.99	482.44	5,743.29	0.08

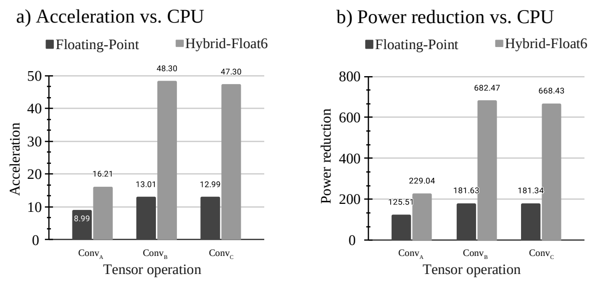

Figure 4.15.: Inference acceleration and power reduction on the TP with floating-point and HF6 vs. CPU on the Zynq-7007S SoC.

arithmetic operations by mapping them onto Xilinx LogiCORE IP cores, these FP operator cores are instantiated in the resultant RTL [117]. In this case, the implementation of the dot-product with the standard FP computation reuses the multiplier and adder cores in different compute sections of the TP. The post-implementation resource utilization and power dissipation of the individual floating-point operator cores are shown in **Tab.** 4.3.

Table 4.3.: Resource utilization and power dissipation of individual multiplier and adder floating-point (IEEE 754) operator cores (Xilinx LogiCORE IP).

Core operation	DSP	FF	LUT	Latency (clk)	Power (mW)
Multiplier	3	151	325	4	7
Adder	2	324	424	8	6

Tensor Processor Synthesized with Hybrid-Float6 Hardware Architecture

To demonstrate the proposed design, the TP with HF6 hardware reuses the standard FP design parameters with the following variation for the 6-bit representation in filter and bias: $BitSize_F = BitSize_B = 6$.

Using equations from Section 4.2.2, the on-chip memory requirements for the hardware accelerator are $Input_M = 84,480$ b, $Filter_M = 178,200$ b, $Bias_M = 360$ b. Hence, the required on-chip memory buffer size is $TP_B = 263,040$ b.

The post-implementation resource utilization and power dissipation are presented in **Tab.** 4.1(b). The complete hardware platform utilizes 30% of BRAM, this includes the on-chip memory requirements of the TP, DMA, and AXI interconnects. The estimated power dissipation of the TP is 84 mW at 200 MHz (this estimation is provided by Xilinx Vivado).

The compute performance and inference schedule of the model on this hardware implementation are shown in **Tab.** 4.2(c) and **Fig.** 4.16(c), respectively. **Fig.** 4.15 presents a comparison of the acceleration and the reduction of power dissipation between standard FP and HF6 hardware implementations.

This deployment does not require model treatment for hardware compatibility. For backward compatibility, the 6-bit FP representation is wrapped into the standard FP. The dedicated hardware design extracts the 6-bit format automatically to perform computation.

4.3.4. Discussion

Training and Quantization

The training with iterative early stop obtains a model with enhanced accuracy than standard early stop. This method iteratively resets the moving averages of Adam's optimizer, which helps to iteratively search for better local minima. This iterative search is suitable for models with low computational cost.

The TensorFlow Lite 8-bit quantization preserves the overall model accuracy. In some cases, the associated regularization effect can improve the accuracy. However, the error distribution in CNN linear regressions gets slightly degraded. In particular, 8-bit quantized output layers incur in

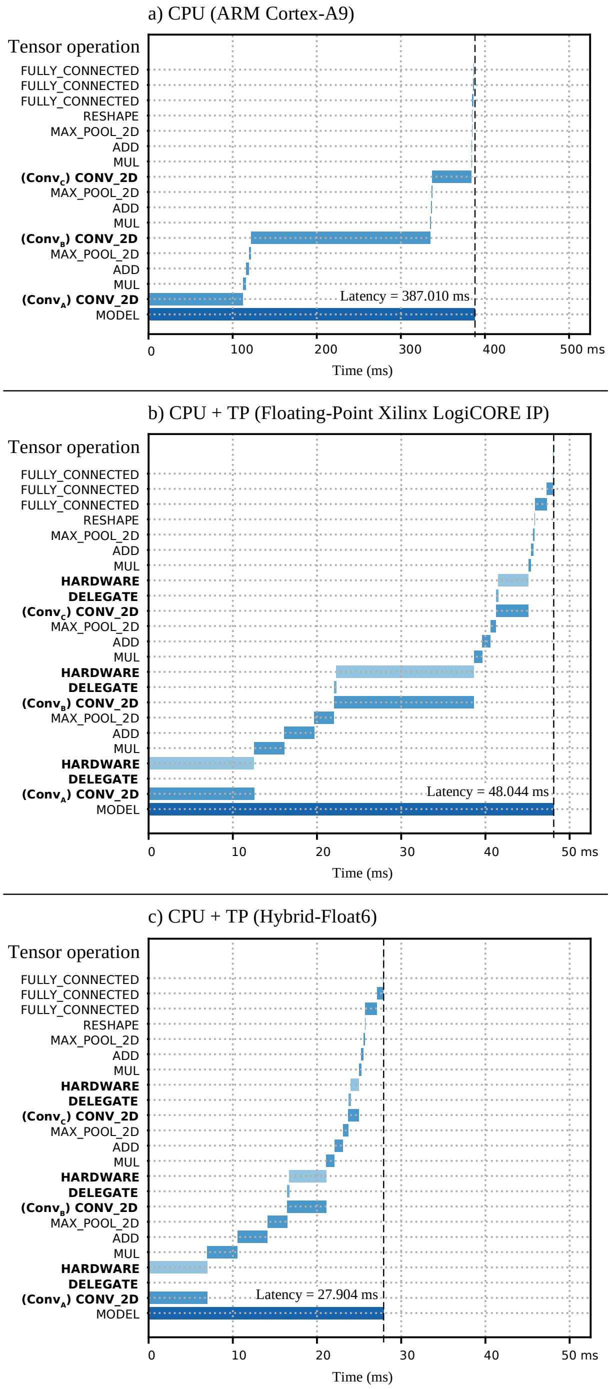

Figure 4.16.: Run-time inference of TensorFlow Lite on the Zynq-7007S SoC. (a) CPU ARM Cortex-A9 at 666 MHz, (b) cooperative CPU + TP with floating-point Xilinx LogiCORE IP at 200 MHz, and (c) cooperative CPU + TP with Hybrid-Float6 at 200 MHz.

discrete-degradation patterns, **Fig.** 4.17(b) shows this effect on three different models. Vertical

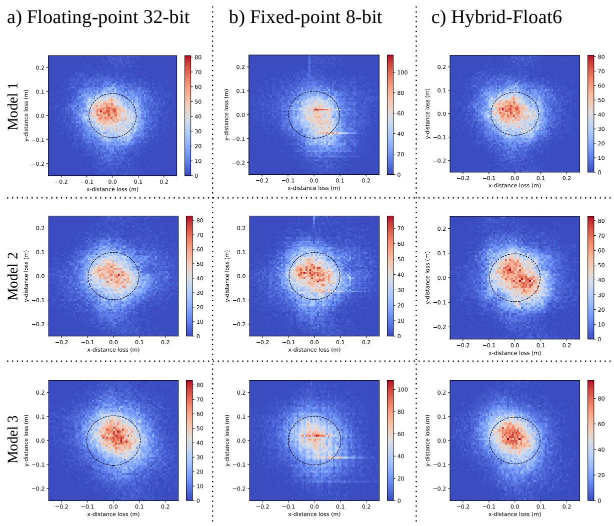

Figure 4.17.: 2D error distribution of three CNN-regression models.

and horizontal patterns appear in the error distribution of 8-bit fixed-point quantization. We attribute this effect to the 8-bit resolution in the activation maps. In the case of HF6 quantization, the activation maps are represented by floating-point preventing this degradation.

The proposed 6-bit FP representation (E4M1) improves latency, hardware area, and power dissipation, while preserving model accuracy. For comparison, in our application, this number format produces better results than the 6-bit logarithmic representation (E5M0). This is demonstrated in **Fig.** 4.14(d) and **Fig.** 4.14(e).

In [32], Lai et al. demonstrated that 4-bit exponent and X-bit mantissa preserves accuracy on SqueezeNet, AlexNet, GoogLeNet, and VGG-16. To contribute on this, I investigated 4-bit exponent and 1-bit mantissa to ALL-CNN-C [129], this produces an accuracy degradation of 1.39% and 0.11% with QAT. While applying 6-bit logarithmic produces a degradation of 11.18% and 7.22% with QAT.

Implementation and Performance

The proposed HF6 implementation reduces on-chip memory and Digital Signal Processing (DSP) utilization while slightly increasing FFs and LUTs compared to the standard FP implementation. See **Tab.** 4.1 and **Fig.** 4.18. This is attributed to the HF6 logic implementation using FF and LUT, while the FP logic implementation uses Xilinx LogiCORE IPs mainly with DSPs.

The compute performance of the CPU and TP on each convolution layer is presented in **Tab.** 4.2 and **Fig.** 4.15. The peak acceleration and power efficiency of the TP with standard FP (Xilinx LogiCORE IP) is 13× and 1,558.13 MFLOPS/s/W, respectively. While the peak acceleration and power efficiency of the TP with HF6 is 48.3× and 5,743.29 MFLOPS/s/W, respectively. The

HF6 hardware demonstrates an improvement of 3.7× in acceleration and power efficiency with respect to the standard FP hardware. See **Fig.** 4.15.

The estimated power dissipation on the SoC is presented in **Fig.** 4.19. This shows a very similar breakdown of power dissipation in both implementations. However, the energy efficiency is increased due to the reduced latency in HF6 hardware. A comparison of related work is presented in **Tab.** 4.4.

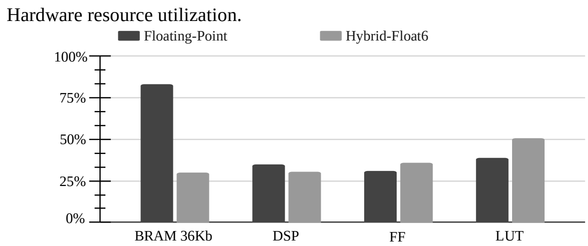

Figure 4.18.: Hardware resource utilization on the Zynq-7007S SoC.

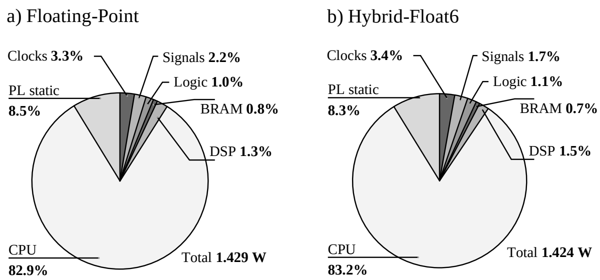

Figure 4.19.: Estimated power dissipation on the Zynq-7007S SoC with PS at 666 MHz and PL at 200 MHz.

The run-time inference of TensorFlow Lite on the SoC is illustrated in **Fig.** 4.16. This shows the convolution layers as the compute-bound operations. The proposed embedded platform is a cooperative system where the convolution operations are delegated to the dedicated hardware accelerator. The ARM CPU obtains a latency of 387 ms (2.58 FPS). The platform with standard FP hardware obtains a latency of 48 ms (20.8 FPS), while the implementation with HF6 obtains a latency of 27.9 ms (35.84 FPS). These represent an overall acceleration of 8× and 13.87× over the CPU, respectively.

This design facilitates ML compatibility/portability as the 6-bit FP is wrapped in the standard FP representation. The dedicated hardware design extracts the 6-bit format automatically and performs computation.

Table 4.4.: Comparison of hardware implementation with related work.

Platform	Chunsheng et al. [103]	Chen et al. [104]	BFP [102]	Paolo et al. [105]	This work
Device	XC7VX690T	XC7K325T	XC7VX690T	XC7Z007S	XC7Z007S
Year	2017	2019	2019	2019	2023
Dev. kit cost	$7,494	$1,299	$7,494	$89	$89
Format (activation/weight)	FP 16-bit	FP 8-bit / 8-bit	FP 16-bit / 8-bit	INT 16-bit	FP 32-bit / 6-bit
Frequency (MHz)	200	200	200	80	200
Peak power efficiency (GFLOP/s/W)	18.72	115.40	82.88	2.98	5.74
Peak throughput (GFLOP/s)	202.42	1086.8	760.83	10.62	0.482
Wall plug power (W)	10.81	9.42	9.18	2.5	2.3
BRAM 36Kb utilization	196.5	234.5	913	44	15
DSP utilization	1728	768	1027	54	20

SoC Design and Compatibility

The proposed design is an alternative for high accuracy and low-power floating-point inference. The system runs as a cooperative hardware/software mechanism. This architecture delegates compute-bound tensor operations to a hardware accelerator.

The hybrid 32-bit FP and 6-bit FP quantization enables high quality of results and backward ML compatibility. Backwards ML compatibility gives portability from training to inference. This enables to run inference of HF6 quantized models on standard FP hardware and vise versa. The proposed HF6 architecture allows to compute inference of non-quantized floating-point ML models for rapid deployment; however, this will incur in accuracy degradation depending on the resilience of the model, see **Fig.** 4.14(c).

4.4. Conclusions

This chapter presents the Hybrid-Float6 quantization and its dedicated hardware accelerator for floating-point CNN computation. Feature maps and weights are represented by 32-bit and 6-bit FP, respectively. The 6-bit FP format is composed of 1-bit sign, 4-bit exponent, and 1-bit mantissa. The 1-bit mantissa enables low-power MAC implementations by reducing the mantissa multiplication to a multiplexer-adder operation. The intrinsic error tolerance of neural networks is exploited to further reduce the hardware design with approximation. This approach improves latency, hardware area, and energy consumption. To preserve accuracy, a QAT training method is presented that, based on regularization effects can improve accuracy. A lightweight TP implementing a pipelined vector dot-product is presented. For ML compatibility/portability, the 6-bit FP is wrapped in the standard floating-point format, which is automatically extracted by the proposed hardware. The hardware/software architecture is compatible with TensorFlow Lite. To evaluate the applicability of this approach, it is presented a CNN-regression model for anomaly localization in a SHM application based on acoustic emissions. The embedded hardware/software framework is demonstrated on XC7Z007S as the smallest Zynq-7000 SoC, suitable for low-power IoT applications. The proposed architecture achieves a peak power efficiency and acceleration on convolution layers of 5.7 GFLOPS/s/W and 48.3×, respectively.

5. Conclusion and Outlook

The use of AI is entering a new era based on the use of ubiquitous connected devices. The sustainability of this transformation requires the adoption of design techniques that reconcile accurate results with cost-effective system architectures. As such, improving the efficiency of AI hardware engines as well as ML portability is considered in this dissertation.

State-of-the-art ML algorithms, specially SNNs and CNNs, represent elevated computational and energy cost. Considering the intrinsic error resilience of ML algorithms, paradigms such as approximate computing came to the rescue by offering efficiency gains to assist the aforementioned concerns. This dissertation contributes with state-of-the-art methods to address low-power neural network accelerator design with custom FP computation.

5.1. Conclusion

In the field of SNN, this dissertation presents a hardware design methodology for low-power inference of SbS neural networks targeting embedded applications. This research exploits the intrinsic error resilience of SbS to improve performance and to reduce hardware complexity. More precisely, it is proposed a vector dot-product module based on approximate computing with configurable quality using hybrid custom FP and logarithmic number representations. This approach reduces computational run-time, memory footprint, and power dissipation while preserving inference accuracy. To demonstrate this approach, a design exploration flow is presented with HLS on a resource-constrained FPGA. The proposed design reduces 20.5× run-time and 8× weight memory footprint, with less than 0.5% of accuracy degradation without retraining on a handwritten digit classification task.

In the field of CNN, this dissertation presents a hardware design methodology for low-power inference targeting sensor analytics applications. In this work, it is proposed the HF6 quantization and its dedicated hardware processor. This quantization allows an optimized FP MAC hardware design by reducing the mantissa multiplication to a multiplexer-adder operation. Additionally, this design exploits the intrinsic error tolerance of neural networks to further reduce the hardware architecture with approximation on the FP subnormal number computation. To preserve model accuracy, a QAT method is presented, which in some cases improves accuracy based on regularization effects. This concept is demonstrated in 2D convolution layers. A low-power TP is proposed implementing a pipelined vector dot-product. For ML portability, the custom FP representation is wrapped in the standard format, which is automatically extracted by the proposed hardware. The hardware/software architecture is integrated with TensorFlow Lite to demonstrate compatibility and portability with industry standard ML frameworks. The applicability of this approach is evaluated with a CNN-regression model for anomaly localization in a SHM application based on AE. The embedded hardware/software framework is demonstrated on XC7Z007S. This is the smallest Zynq-7000 SoC, suitable for ubiquitous IoT devices. The proposed implementation achieves a peak power efficiency and acceleration of 5.7 GFLOPS/s/W and 48.3×, respectively.

5.2. Outlook

In this research, three lines of future work are foreseen:

- **Reducing energy consumption and memory footprint.** The proposed architectures consists of a hybrid floating-point quantization using 32-bit activation maps. These can be represented using lower-bit formats; for example, Bfloat16 and 8-bit or lower custom floating-point. This would reduce hardware resource utilization, memory footprint and data transfer, while preserving backward compatibility and accuracy. (FP formats present better QoR than fixed-pint representations based on their dynamic value range.)

- **Increasing throughput.** The proposed designs require matching higher computational throughput with memory bandwidth. This would replace the low-power pipeline hardware design with a parallelized structure. This can be achieved by using wider memory channels and systolic arrays to increase throughput. This will increment hardware area and energy consumption, suitable for UltraScale SoC FPGA architectures or ASIC implementations.

- **Performing computer vision applications.** The proposed implementations are designed for sensor analytics workloads. For computer vision applications, the hardware design would require increased on-chip memory capacity for larger bias and filter vectors and higher computational throughput in a larger FPGA SoC or ASIC.

5.3. Summary

Based on the relaxed need for fully accurate or deterministic computation of neural networks, approximate computing techniques allow substantial enhancement in processing efficiency with moderated accuracy degradation. This dissertation focuses on the investigation of design methodologies to exploit the intrinsic error resilience of ML algorithms to optimize high-quality FP inference on low-power embedded systems. These design techniques reconcile accurate results with cost-effective system architectures.

The outcome of this dissertation contributes to the rise of a sustainable next generation of energy efficient neural network processors with ML portability and high-accuracy as design philosophy.

A. Appendix

A.1. SbS algorithm

The SbS network inference is described in **Algorithm** 5, while spike production and layer update are described in **Algorithm** 6 and **Algorithm** 7, respectably.

Algorithm 5: SbS network inference.

input: Layers of the network as H^l, where l is the layer index.
input: N_L as the number of layers.
input: N_X^l, N_Y^l as the size of layers.
input: N_{Spk} as the number of spikes for inference (iterations).
output: H^l.

```
for t = 0 to N_{Spk} - 1 do
    Initialization of H^l(i_X, i_Y, :) :
    if t == 0 then
        for l = 0 to N_L - 1 do
            for i_X = 0, i_Y = 0 to N_X^l - 1, N_Y^l - 1 do
                for i_H = 0 to N_H^l - 1 do
                    H^l(i_X, i_Y, i_H) = 1/N_H^l
                end for
            end for
        end for
    end if
    Production of spikes :
    for l = 0 to N_L - 1 do
        if l == 0 then
            Draw spikes from input // (Algorithm 6)
        else
            Draw spikes from H^l // (Algorithm 6)
        end if
    end for
    Update layers :
    for l = 0 to N_L - 1 do
        Update H^l // (Algorithm 7)
    end for
end for
```

Algorithm 6: Spike production.

```
input: Layer as H_t ∈ ℝ^(N_X×N_Y×N_H), where
    N_X is the layer width,
    N_Y is the layer height
    N_H is the length of h⃗ (IP vector).
output: Output spikes as S_t^out ∈ ℕ^(N_X×N_Y)
for i_X = 0, i_Y = 0 to N_X − 1, N_Y − 1 do
    Generate spike :
    th = MT19937PseudoRandom()/(2^32 − 1)
    acu = 0
    for i_H = 0 to N_H − 1 do
        acu = acu + H_t(i_X, i_Y, i_H)
        if th ≤ acu or i_H == N_H − 1 then
            S_t^out(i_X, i_Y) = i_H
        end if
    end for
end for
```

Algorithm 7: SbS layer update.

input: Layer as $H \in \mathbb{R}^{N_X \times N_Y \times N_H}$, where
N_X is the layer width,
N_Y is the layer height
N_H is the length of $\vec{h}$ (IP vector).
input: Synaptic matrix as $W \in \mathbb{R}^{K_X \times K_Y \times M_H \times N_H}$, where
$K_X \times K_Y$ is the size of the convolution/pooling kernel,
M_H is the length of $\vec{h}$ from previous layer,
N_H is the length of $\vec{h}$ from this layer.
input: Input spike matrix from previous layer as $S_t^{in} \in \mathbb{N}^{N_{Xin} \times N_{Yin}}$, where
N_{Xin} is the width of the previous layer,
N_{Yin} is the height of the previous layer.
input: Strides of X and Y as $stride_X$ and $stride_Y$, respectively.
input: Epsilon as $\epsilon \in \mathbb{R}$.
output: Updated layer as $H^{new} \in \mathbb{R}^{N_X \times N_Y \times N_H}$.

Update layer :
$z_X = 0$ `// X and Y index for` S_t^{in}
$z_Y = 0$
for $i_Y = 0$ **to** $N_Y - 1$ **do**
  for $i_X = 0$ **to** $N_X - 1$ **do**
    $\vec{h} = H(i_X, i_Y, :)$
    Update IP :
    for $j_X = 0, j_Y = 0$ **to** $K_X - 1, K_Y - 1$ **do**
      $s_t = S_t^{in}(z_X + j_X, z_Y + j_Y)$
      $\vec{w} = W(j_X, j_Y, s_t, :)$
      $\vec{p} = 0$
      Dot-product :
      $r = 0$
      for $j_H = 0$ **to** $N_H - 1$ **do**
        $\vec{p}(j_H) = \vec{h}_{(j_H)}\vec{w}(j_H)$
        $r = r + \vec{p}(j_H)$
      end for
      if $r \neq 0$ **then**
        Update IP vector :
        for $i_H =$ **to** $N_H - 1$ **do**
          $h^{new}(i_H) = \frac{1}{1+\epsilon}\left(h(i_H) + \epsilon\frac{\vec{p}(i_H)}{r}\right)$
        end for
        Set the new H vector for the layer :
        $H^{new}(i_X, i_Y, :) = \vec{h}^{new}$
      end if
    end for
    $z_X = z_X + stride_X$
  end for
  $z_Y = z_Y + stride_Y$
end for

Acronyms

Abbreviations

AI	Artificial Intelligence. 2, 3, 5, 17, 18, 47, 77
ML	Machine Learning. 2–6, 47–49, 51, 58, 75–79
IoT	Internet-of-Things. 2, 5, 6, 48, 76, 78
SNN	Spiking Neural Network. 5, 12, 18, 19, 21, 22, 51, 77
CNN	Convolutional Neural Network. 3–7, 9, 12, 14, 18, 19, 48–51, 55–57, 60, 64, 69, 73, 75–78, 87
ANN	Artificial Neural Network. 3–5, 7, 18, 21, 22, 48, 49
FP	Floating-Point. 3, 5–7, 9, 15, 16, 19, 20, 44, 45, 48–50, 53–55, 58, 60, 66, 69–71, 73, 75–79
SbS	Spike-by-Spike. 5–7, 9, 11–14, 18–23, 25, 31, 32, 36–39, 41, 43–46, 77, 87, 91
LIF	Leaky Integrate-and-Fire. 5, 12, 18
HF6	Hybrid-Float6. 6, 7, 49, 50, 54, 68, 70, 71, 73–75, 78
MAC	Multiply-Accumulate. 6, 7, 12, 21, 49, 53–55, 76, 78
CPS	Cyber-Physical Systems. 2
ASIC	Application-Specific Integrated Circuit. 3, 6, 22, 48, 78, 79
NPU	Neural Processing Unit. 3
FPGA	Field-Programmable Gate Array. 3, 6, 7, 22, 23, 31, 48–51, 60, 69, 77–79
HAR	Human Activity Recognition. 6
HPC	High Performance Computing. 6, 48
CPU	Central Processing Unit. 6, 23, 32–35, 38, 41, 43, 51, 52, 69, 73, 75
IP	Inference Population. 11–13, 18, 22, 24–26, 87
NaN	Not a Number. 16, 55
MLP	Multi-Layer Perceptron. 18
DNN	Deep Neural Network. 18, 21
NNMF	Non-Negative Matrix Factorization. 18

NN	Neural Network. 18
QoR	Quality of Result. 19, 20, 26, 27, 38, 46, 78
WQ	Weight Quantization. 21
BNN	Binary Neural Network. 21
XNOR	Logical Exclusive Non-Disjunction. 21
PU	Processing Unit. 23–25, 27, 32–35, 38, 41, 44, 87
API	Application Programming Interface. 23, 32, 58
DMA	Direct Memory Access. 23, 34, 51, 53, 59, 69, 71
HLS	High-Level Synthesis. 24, 36, 52, 53, 70, 77
PS	Processing System. 31, 68, 69
PL	Programmable Logic. 31, 68, 69
RTL	Register-Transfer Level. 36, 70
LUT	Look-up Table. 44, 73
FF	Flip-Flop. 44, 73
GPU	Graphics Processing Unit. 48, 65
QAT	Quantization-Aware Training. 49, 58, 68, 73, 76, 78
SHM	Structural Health Monitoring. 49, 50, 76, 78
SoC	System-on-Chip. 49, 60, 68–70, 74, 76, 78, 79
AE	Acoustic Emission. 50, 78
RNN	Recurrent Neural Network. 51
TP	Tensor Processor. 51–53, 56, 57, 59, 60, 69–71, 73, 76, 78
AWG	Arbitrary Waveform Generator. 62
STFT	Short-Time Fourier Transform. 62, 63
FFT	Fast Fourier Transform. 64
PC	Personal Computer. 65
DSP	Digital Signal Processing. 73

List of Figures

List of Tables

Bibliography

[1] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.

[2] Héctor Espinoza, Gerhard Kling, Frank McGroarty, Mary O'Mahony, and Xenia Ziouvelou. Estimating the impact of the internet of things on productivity in europe. *Heliyon*, 6(5):e03935, 2020.

[3] Vitor Alcácer and Virgilio Cruz-Machado. Scanning the industry 4.0: A literature review on technologies for manufacturing systems. *Engineering science and technology, an international journal*, 22(3):899–919, 2019.

[4] Jing Zhang and Dacheng Tao. Empowering things with intelligence: A survey of the progress, challenges, and opportunities in artificial intelligence of things. *IEEE Internet of Things Journal*, 2020.

[5] Kou-Hung Lawrence Loh. 1.2 fertilizing aiot from roots to leaves. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 15–21. IEEE, 2020.

[6] Jie Han and Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*, pages 1–6. IEEE, 2013.

[7] Maxence Bouvier, Alexandre Valentian, Thomas Mesquida, François Rummens, Marina Reyboz, Elisa Vianello, and Edith Beigne. Spiking neural networks hardware implementations and challenges: A survey. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(2):1–35, 2019.

[8] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[9] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[10] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[11] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[12] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2:598–605, 1989.

[13] Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, 5:164–171, 1992.

[14] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[15] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[16] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.

[17] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–9, 2013.

[18] Syed Ghayoor Abbas Gillani. Exploiting error resilience for hardware efficiency: targeting iterative and accumulation based algorithms. 2020.

[19] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. Approxann: An approximate computing framework for artificial neural network. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 701–706. IEEE, 2015.

[20] Nicholas P Carter, Helia Naeimi, and Donald S Gardner. Design techniques for cross-layer resilience. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 1023–1028. IEEE, 2010.

[21] Uroš Lotrič and Patricio Bulić. Applicability of approximate multipliers in hardware neural networks. *Neurocomputing*, 96:57–65, 2012.

[22] Zidong Du, Krishna Palem, Avinash Lingamneni, Olivier Temam, Yunji Chen, and Chengyong Wu. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *2014 19th Asia and South Pacific design automation conference (ASP-DAC)*, pages 201–206. IEEE, 2014.

[23] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *Proceedings of the 35th International Conference on Computer-Aided Design*, pages 1–7, 2016.

[24] Syed Shakib Sarwar, Swagath Venkataramani, Anand Raghunathan, and Kaushik Roy. Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 145–150. IEEE, 2016.

[25] Georgios Zervakis, Hassaan Saadat, Hussam Amrouch, Andreas Gerstlauer, Sri Parameswaran, and Jörg Henkel. Approximate computing for ml: State-of-the-art, challenges and visions. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 189–196, 2021.

[26] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. Efficient embedded learning for iot devices. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 308–311. IEEE, 2016.

[27] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[28] Hussam Amrouch, Georgios Zervakis, Sami Salamin, Hammam Kattan, Iraklis Anagnostopoulos, and Jörg Henkel. Npu thermal management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3842–3855, 2020.

[29] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*, 2015.

[30] Philip Colangelo, Nasibeh Nasiri, Eriko Nurvitadhi, Asit Mishra, Martin Margala, and Kevin Nealis. Exploration of low numeric precision deep learning inference using intel® fpgas. In *2018 IEEE 26th annual international symposium on field-programmable custom computing machines (FCCM)*, pages 73–80. IEEE, 2018.

[31] Julian Faraone, Martin Kumm, Martin Hardieck, Peter Zipf, Xueyuan Liu, David Boland, and Philip HW Leong. Addnet: Deep neural networks using fpga-optimized multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(1):115–128, 2019.

[32] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Deep convolutional neural network inference with floating-point weights and fixed-point activations. *arXiv preprint arXiv:1703.03073*, 2017.

[33] Mark D McDonnell and Lawrence M Ward. The benefits of noise in neural systems: bridging theory and experiment. *Nature Reviews Neuroscience*, 12(7):415–425, 2011.

[34] Udo Ernst, David Rotermund, and Klaus Pawelzik. Efficient computation based on stochastic spikes. *Neural computation*, 19(5):1313–1343, 2007.

[35] Joel Dapello, Tiago Marques, Martin Schrimpf, Franziska Geiger, David D. Cox, and James J. DiCarlo. Simulating a primary visual cortex at the front of cnns improves robustness to image perturbations. *bioRxiv*, 2020.

[36] David Rotermund and Klaus R. Pawelzik. Back-propagation learning in deep spike-by-spike networks. *Frontiers in Computational Neuroscience*, 13:55, 2019.

[37] David Rotermund and Klaus R. Pawelzik. Massively parallel FPGA hardware for spike-by-spike networks. *bioRxiv*, 2019.

[38] Kaushik Roy, Akhilesh Jaiswal, and Priyadarshini Panda. Towards spike-based machine intelligence with neuromorphic computing. *Nature*, 575(7784):607–617, 2019.

[39] Aaron R Young, Mark E Dean, James S Plank, and Garrett S Rose. A review of spiking neuromorphic hardware communication systems. *IEEE Access*, 7:135606–135620, 2019.

[40] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar,

W. P. Risk, B. Jackson, and D. S. Modha. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, Oct 2015.

[41] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber. Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits*, 48(8):1943–1953, Aug 2013.

[42] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[43] David Rotermund and Klaus R. Pawelzik. Biologically plausible learning in a deep recurrent spiking network. *bioRxiv*, 2019.

[44] Peter Dayan and Laurence F Abbott. *Theoretical neuroscience: computational and mathematical modeling of neural systems*. Computational Neuroscience Series, 2001.

[45] Yarib Nevarez, Alberto Garcia-Ortiz, David Rotermund, and Klaus R Pawelzik. Accelerator framework of spike-by-spike neural networks for inference and incremental learning in embedded systems. In *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–5. IEEE, 2020.

[46] Guoqiang Li, Chao Deng, Jun Wu, Xuebing Xu, Xinyu Shao, and Yuanhang Wang. Sensor data-driven bearing fault diagnosis based on deep convolutional neural networks and s-transform. *Sensors*, 19(12):2750, 2019.

[47] Fei Dong, Xiao Yu, Enjie Ding, Shoupeng Wu, Chunyang Fan, and Yanqiu Huang. Rolling bearing fault diagnosis using modified neighborhood preserving embedding and maximal overlap discrete wavelet packet transform with sensitive features selection. *Shock and Vibration*, 2018, 2018.

[48] Tomonori Nagayama and Billie F Spencer Jr. Structural health monitoring using smart sensors. Technical report, Newmark Structural Engineering Laboratory. University of Illinois at Urban, 2007.

[49] Jindong Wang, Yiqiang Chen, Shuji Hao, Xiaohui Peng, and Lisha Hu. Deep learning for sensor-based activity recognition: A survey. *Pattern Recognition Letters*, 119:3–11, 2019.

[50] Yong Chan Kim, Hyeong-Geun Yu, Jae-Hoon Lee, Dong-Jo Park, and Hyun-Woo Nam. Hazardous gas detection for ftir-based hyperspectral imaging system using dnn and cnn. In *Electro-Optical and Infrared Systems: Technology and Applications XIV*, volume 10433, page 1043317. International Society for Optics and Photonics, 2017.

[51] Yarib Nevarez, David Rotermund, Klaus R Pawelzik, and Alberto Garcia-Ortiz. Accelerating spike-by-spike neural networks on fpga with hybrid custom floating-point and logarithmic dot-product approximation. *IEEE Access*, 2021.

[52] Yarib Nevarez, Andreas Beering, Amir Najafi, Ardalan Najafi, Wanli Yu, Yizhi Chen, Karl-Ludwig Krieger, and Alberto Garcia-Ortiz. Cnn sensor analytics with hybrid-float6 quantization on low-power embedded fpgas. *IEEE Access*, 11:4852–4868, 2023.

[53] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[54] Varun Gulshan, Lily Peng, Marc Coram, Martin C Stumpe, Derek Wu, Arunachalam Narayanaswamy, Subhashini Venugopalan, Kasumi Widner, Tom Madams, Jorge Cuadros, et al. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *jama*, 316(22):2402–2410, 2016.

[55] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[56] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Achieving human parity in conversational speech recognition. *arXiv preprint arXiv:1610.05256*, 2016.

[57] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[59] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[60] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[61] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, GA, 2013.

[62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[63] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, pages 177–186. Springer, 2010.

[64] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM review*, 60(2):223–311, 2018.

[65] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[66] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[67] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[68] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[69] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–377, 2018.

[70] Wulfram Gerstner, Werner M Kistler, Richard Naud, and Liam Paninski. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[71] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[72] Emre O Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine*, 36(6):51–63, 2019.

[73] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in neuroscience*, 10:508, 2016.

[74] Timothée Masquelier and Simon J Thorpe. Unsupervised learning of visual features through spike timing dependent plasticity. *PLoS computational biology*, 3(2):e31, 2007.

[75] Eugene M Izhikevich. Which model to use for cortical spiking neurons? *IEEE transactions on neural networks*, 15(5):1063–1070, 2004.

[76] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[77] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[78] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[79] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[80] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[81] Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. Ieee standard for floating-point arithmetic. *IEEE Std*, 754(2008):1–70, 2008.

[82] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[83] Bert Moons and Marian Verhelst. A 0.3–2.6 tops/w precision-scalable processor for real-time large-scale convnets. In *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages 1–2. IEEE, 2016.

[84] Paul N Whatmough, Sae Kyu Lee, Hyunkwang Lee, Saketh Rama, David Brooks, and Gu-Yeon Wei. 14.3 a 28nm soc with a 1.2 ghz 568nj/prediction sparse deep-neural-network engine with> 0.1 timing error rate tolerance for iot applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 242–243. IEEE, 2017.

[85] Xiaoyu Sun, Shihui Yin, Xiaochen Peng, Rui Liu, Jae-sun Seo, and Shimeng Yu. Xnor-rram: A scalable and parallel resistive synaptic architecture for binary neural networks. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1423–1428. IEEE, 2018.

[86] Nitin Rathi, Priyadarshini Panda, and Kaushik Roy. Stdp-based pruning of connections and weight quantization in spiking neural networks for energy-efficient recognition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):668–677, 2018.

[87] Sanchari Sen, Swagath Venkataramani, and Anand Raghunathan. Approximate computing for spiking neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 193–198. IEEE, 2017.

[88] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[89] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International conference on machine learning*, pages 1058–1066, 2013.

[90] Emre O Neftci, Bruno U Pedroni, Siddharth Joshi, Maruan Al-Shedivat, and Gert Cauwenberghs. Stochastic synapses enable efficient brain-inspired learning machines. *Frontiers in neuroscience*, 10:241, 2016.

[91] Gopalakrishnan Srinivasan, Abhronil Sengupta, and Kaushik Roy. Magnetic tunnel junction based long-term short-term stochastic synapse for a spiking neural network with on-chip stdp learning. *Scientific reports*, 6:29545, 2016.

[92] Lars Buesing, Johannes Bill, Bernhard Nessler, and Wolfgang Maass. Neural dynamics as sampling: a model for stochastic computation in recurrent networks of spiking neurons. *PLoS Comput Biol*, 7(11):e1002211, 2011.

[93] Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert Legenstein. Deep rewiring: Training very sparse deep networks. *arXiv preprint arXiv:1711.05136*, 2017.

[94] Gregory K Chen, Raghavan Kumar, H Ekin Sumbul, Phil C Knag, and Ram K Krishnamurthy. A 4096-neuron 1m-synapse 3.8-pj/sop spiking neural network with on-chip stdp learning and sparse weights in 10-nm finfet cmos. *IEEE Journal of Solid-State Circuits*, 54(4):992–1002, 2018.

[95] Sadique Sheik, Somnath Paul, Charles Augustine, Chinnikrishna Kothapalli, Muhammad M Khellah, Gert Cauwenberghs, and Emre Neftci. Synaptic sampling in hardware spiking neural networks. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2090–2093. IEEE, 2016.

[96] M Jerry, A Parihar, B Grisafe, A Raychowdhury, and S Datta. Ultra-low power probabilistic imt neurons for stochastic sampling machines. In *2017 Symposium on VLSI Circuits*, pages T186–T187. IEEE, 2017.

[97] Ming ZHANG, GU Zonghua, and PAN Gang. A survey of neuromorphic computing based on spiking neural networks. *Chinese Journal of Electronics*, 27(4):667–674, 2018.

[98] Yongtae Kim, Yong Zhang, and Peng Li. An energy efficient approximate adder with carry skip for error resilient neuromorphic vlsi systems. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 130–137. IEEE, 2013.

[99] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[100] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[101] Sean O Settle, Manasa Bollavaram, Paolo D'Alberto, Elliott Delaye, Oscar Fernandez, Nicholas Fraser, Aaron Ng, Ashish Sirasao, and Michael Wu. Quantizing convolutional neural networks for low-power high-throughput inference engines. *arXiv preprint arXiv:1805.07941*, 2018.

[102] Xiaocong Lian, Zhenyu Liu, Zhourui Song, Jiwu Dai, Wei Zhou, and Xiangyang Ji. High-performance fpga-based cnn accelerator with block-floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(8):1874–1885, 2019.

[103] Chunsheng Mei, Zhenyu Liu, Yue Niu, Xiangyang Ji, Wei Zhou, and Dongsheng Wang. A 200mhz 202.4 gflops@ 10.8 w vgg16 accelerator in xilinx vx690t. In *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 784–788. IEEE, 2017.

[104] Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. Low-precision floating-point arithmetic for high-performance fpga-based cnn acceleration. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(1):1–21, 2021.

[105] Paolo Meloni, Antonio Garufi, Gianfranco Deriu, Marco Carreras, and Daniela Loi. Cnn hardware acceleration on a low-power and low-cost apsoc. In *2019 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 7–12. IEEE, 2019.

[106] Chang Gao, Antonio Rios-Navarro, Xi Chen, Shih-Chii Liu, and Tobi Delbruck. Edgedrnn: Recurrent neural network accelerator for edge inference. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(4):419–432, 2020.

[107] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[108] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[109] Nassim Abderrahmane, Edgar Lemaire, and BenoÃ®t Miramond. Design space exploration of hardware spiking neurons for embedded artificial intelligence. *Neural Networks*, 121:366 – 386, 2020.

[110] Maxence Bouvier, Alexandre Valentian, Thomas Mesquida, Francois Rummens, Marina Reyboz, Elisa Vianello, and Edith Beigne. Spiking neural networks hardware implementations and challenges: A survey. *J. Emerg. Technol. Comput. Syst.*, 15(2), April 2019.

[111] Katrin Amunts, Alois C Knoll, Thomas Lippert, Cyriel MA Pennartz, Philippe Ryvlin, Alain Destexhe, Viktor K Jirsa, Egidio D?Angelo, and Jan G Bjaalie. The human brain project – synergy between neuroscience, computing, informatics, and brain-inspired technologies. *PLoS biology*, 17(7):e3000344, 2019.

[112] Jongsun Park, Jung Hwan Choi, and Kaushik Roy. Dynamic bit-width adaptation in dct: An approach to trade off image quality and computation energy. *IEEE transactions on very large scale integration (VLSI) systems*, 18(5):787–793, 2009.

[113] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. Impact: imprecise adders for low-power approximate computing. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 409–414. IEEE, 2011.

[114] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):1–33, 2016.

[115] Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[116] UG585 Xilinx. Zynq-7000 all programmable soc: Technical reference manual, 2015.

[117] James Hrica. Floating-point design with vivado hls. *Xilinx Application Note*, 2012.

[118] Michal Lom, Ondrej Pribyl, and Miroslav Svitek. Industry 4.0 as a part of smart cities. In *2016 Smart Cities Symposium Prague (SCSP)*, pages 1–6. IEEE, 2016.

[119] Turker Ince, Serkan Kiranyaz, Levent Eren, Murat Askar, and Moncef Gabbouj. Real-time motor fault detection by 1-d convolutional neural networks. *IEEE Transactions on Industrial Electronics*, 63(11):7067–7075, 2016.

[120] Olivier Janssens, Viktor Slavkovikj, Bram Vervisch, Kurt Stockman, Mia Loccufier, Steven Verstockt, Rik Van de Walle, and Sofie Van Hoecke. Convolutional neural network based fault detection for rotating machinery. *Journal of Sound and Vibration*, 377:331–345, 2016.

[121] Osama Abdeljaber, Onur Avci, Serkan Kiranyaz, Moncef Gabbouj, and Daniel J Inman. Real-time vibration-based structural damage detection using one-dimensional convolutional neural networks. *Journal of Sound and Vibration*, 388:154–170, 2017.

[122] Xiaojie Guo, Liang Chen, and Changqing Shen. Hierarchical adaptive deep convolution neural network and its application to bearing fault diagnosis. *Measurement*, 93:490–502, 2016.

[123] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 5–14, 2017.

[124] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[125] Shirsendu Sikdar, Sauvik Banerjee, and G. Ashish. Ultrasonic guided wave propagation and disbond identification in a honeycomb composite sandwich structure using bonded piezoelectric wafer transducers. *Journal of Intelligent Material Systems and Structures*, 27, 10 2015.

[126] U. Kiencke, M. Schwarz, and T. Weickert. *Signalverarbeitung: Zeit-Frequenz-Analysen und SchÃ€tzverfahren*. Oldenbourh, 2008.

[127] R. B. Blackman and J. W. Tukey. The measurement of power spectra from the point of view of communications engineering - part i. *Bell System Technical Journal*, 37(1):185–282, 1958.

[128] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[129] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.