Doctoral Thesis

Machine Learning Acceleration for Tightly Energy-Constrained Devices

Author(s):
Andri, Renzo

Publication Date:
2020

Permanent Link:
https://doi.org/10.3929/ethz-b-000430821

Rights / License:
In Copyright - Non-Commercial Use Permitted
Machine Learning Acceleration for Tightly Energy-Constrained Devices

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
RENZO ANDRI
MSc ETH EEIT
born on July 17th, 1990
citizen of Müstair GR, Switzerland

accepted on the recommendation of
Prof. Dr. Luca Benini, examiner
Prof. Dr. Andrea Calimera, co-examiner

2020
Acknowledgements

This thesis is the result of almost four and a half years of doctoral studies. First, I would like to express my sincere appreciation to my supervisor, Professor Luca Benini, who has given me the possibility to work in this exciting research field while being part of his inspiring and excellent team. His premium guidance and his own keen interest in the topic, paired with his love for details and constructive discussions have helped me to develop the needed research skills. Second, I want to thank the co-referee Professor Andrea Calimera, for his insightful and concise review. I also want to thank the main funding partners who enabled these Ph.D. studies, namely the Swiss National Science Foundation SNF and Huawei Technologies Sweden AB. A constant companion during the Ph.D. was Lukas Cavigelli, with whom I have had so many interesting technical discussions, joint projects, and supervised a series of student projects together. Davide Rossi has been an additional advisor during the entire time; your support in technical questions and discussion has been an asset. Thanks also to Tomas Henriksson from Huawei Research, who has closely followed the RNN ASIP project, and contributed with his knowledge on the RRM field and the processor extensions. Special thanks also go to Gianna Paulin, who has taken over the RNN ASIP project and is continuing my work. During this thesis, I have supervised over 20 student projects; among them, I would like to thank especially the following students: Geethan Karunaratne and Andrawes Al Bahou have implemented the BNN accelerator presented in chapter 6. The following students have contributed to the embedded Sound Event Detection (SED) topic presented in chapter 3: Andrei Cramariuc (training of
the BNN for Sound Event Detection (SED) and implementation on the STM32F469I discovery board), Li-Yung Chen and Tim Fischer (MFCC and BNN implementation on GAP8 and wake-up circuit). Furthermore, Gianmarco Cerutti has improved the BNN training and implementation from the microphone data acquisition to the final classification, during his Ph.D. exchange period at IIS. I also want to thank my close companions Francesco Conti, Matteo Spallanzani, Gianmarco Cerutti, Alfio Di Mauro, Daniele Palossi, Björn Forsberg, Giovanni Rovere, and Antonio Libri, with whom I not just found interesting conversational partners, but also shared plenty of lunches and more coffees and some rounds of billiardino matches. Thanks, Antonio Pullini, Davide Rossi, Beat Muheim and Frank K. Gürkaynak for the support on the Poseidon/Hyperdrive back-end design, the first globalfoundries 22 nm FDX tape-out at IIS. Big thanks also goes to the entire PULP team: Michael Gautschi, Andreas Traber, Sven Stucki, Florian Zaruba, Davide Schiavone, Florian Glaser, Manuel Eggimann, Stefan Mach, Robert Balas, Matheus Cavalcante, Andreas Kurth, and many more. I had the honor to be part of the team at the very start where Matthias Baer and I designed the first OpenRISC core for the PULP project during my Master studies. Since then, the project has evolved to mature level from software toolflow over optimized RTL implementation to actual silicon-proven chips and boards, serving as an optimal platform for processor architecture research. IIS provided me a nice working environment, which is provided by following people who I want to thank particularly: Frank K. Gürkaynak providing support from scientific to administrative points; Christine Haller organizing HR and administratives. Hans-Jörg Gisler providing soldering equipment and tools, and very important the coffee machine. Christoph Wicki, and Adam Feigin providing IT support, and Beat Muheim support for EDA tools. Special thanks go to my project advisors during my master’s projects: Michael Gautschi, Niko Münzenrieder, Giuseppe Cantarella, Michele Magn, and Andres Gomez. All of them contributed to improving my skills in methodology and piqued my interest in research and motivated me to start a Ph.D. research project. Finally, I want to thank all people who have supported me outside of ETH. Primarily, my family who has always supported me in all respects. Ultimately, I thank Yu for her unconditional love and joy she has shared with me in the last intense time of my Ph.D.
Abstract

Neural Networks have revolutionized the artificial intelligence and machine learning field in recent years, enabling human and even superhuman performance on several challenging tasks in a plethora of different applications. Unfortunately, these networks have dozens of millions of parameters and need billions of complex floating-point operations, which does not fit the requirements of rising Internet-of-Things (IoT) end nodes. IoT nodes are connected sensor nodes integrated ubiquitously in our daily life as wearables, smartphones, smart homes, and many more. A common approach to supporting artificial intelligence in these devices is running neural networks in the cloud, but this is often not reasonable due to privacy concerns, latency, reliability, scalability, and high energy costs for data transmission. Therefore, it is needed to run the networks directly on the node enabling artificial intelligence on IoT end nodes.

In this thesis, we tackle this challenge on three levels: The embedded domain, Application-specific processor design, and custom hardware accelerator design:

In the embedded domain, we have developed an energy-efficient smartwatch system based on low-power sensors and components, using a light-weight decision tree achieving 84% accuracy within 2.2 mJ energy costs. Furthermore, we have trained and implemented a binary neural network, to fit on the low-power microcontroller GAP8 (i.e., 28 times smaller memory footprint). Furthermore, we present the full system including feature extraction and a bit-level parallelized implementation of the BNN. An accuracy of 77.9% has been achieved,
which is a drop of 7.2% in accuracy compared to the full-precision baseline at an energy cost of 25.5 mJ.

In the application-specific processor design domain, we have implemented a benchmark suite of typical neural networks from the Radio Resource Management field, on a RISC-V processor (i.e., RI5CY core of the PULP project). As neural network topologies and algorithms change very frequently and FPGA solutions are too costly for large-scale distribution, we have extended the RISC-V core with new instructions. Combined with optimized software implementation, the energy efficiency has been improved 10 times to 436 GOp/s/W with 15 times higher throughput.

In the last part of the thesis, we present convolution neural network accelerators for highly-quantized neural networks. Binary-Weight Neural Networks and Binary Neural Networks show high performance compared to their full-precision baselines and have therefore been evaluated for hardware acceleration. Fully binary neural networks still have a rather high gap in performance (e.g., 12 points in classification accuracy on ImageNet) to their full-precision equivalent networks. BWN, on the other side, have managed to reduce their gap massively and reach state-of-the-art performance in simple task and good performance on even harder tasks (i.e., 1-2 percentage points on ImageNet). Thanks to the simple arithmetics in binary-weight and binary neural networks, combined with efficient latch-based memories, data re-use, and optimized adder trees, peak energy efficiencies up to 149 TOp/s/W\(^1\) for binary-weights and 205 TOp/s/W for the fully binarized neural networks have been achieved, 26 times better than comparable full-precision accelerators.

\(^1\)61.2 TOp/s/W in umc 65 nm technology. Scaled to 22 nm technology, based on Dreslinksi et al. [1].
Zusammenfassung


Im Entwurf von eingebetteten Systemen haben wir ein energieeffizientes Smartwatch-System entwickelt, das auf stromsparenden Sensoren und Komponenten basiert und das einen Entscheidungsbaum mit tiefer Rechenkomplexität verwendet. Das System erreicht eine Genauigkeit von 84% bei 2,2 mJ Energiekosten pro Klassifikation. Darüber hinaus haben wir ein binäres Neuronales Netz (BNN) trainiert und implementiert, das auf den stromsparenden Mikrocontroller GAP8 passt (bei 28-fach kleineren Speicherplatzbedarf). Darüber hinaus stellen wir das vollständige System vor, einschließlich der Datenverarbeitung und der
optimierten Implementierung des BNNs. Es wurde eine Genauigkeit von 77,9% erreicht, was einem Genauigkeitsverlust von 7,2% im Vergleich zum Referenznetz mit präzisen Fließkomma-Operationen entspricht, was aber dem normalen Genauigkeitsverlust von ähnlichen BNNs entspricht.


\(^2\)In der umc 65 nm-Technologie eine Energieeffizienz von 61,2 TOp/s/W wurden erreicht. 149 TOp/s/W basierend auf der Technologieskalierungsmethode von Dreslinksi et al. [1].
Contents

Acknowledgements iii
Abstract v
Zusammenfassung vii

1 Introduction 1
  1.1 Outline ............................................. 6
  1.2 Contributions and Publications .................... 8

2 Energy-Efficient Design of Embedded Context Recognition 13
  2.1 Introduction ........................................... 14
  2.2 Related work .......................................... 17
  2.3 Smartwatch System Architecture .................... 20
    2.3.1 MSP430 Core ................................. 21
    2.3.2 PULP Accelerator ............................. 22
    2.3.3 Sensors ......................................... 25
  2.4 Context Classification .............................. 26
    2.4.1 Feature Extraction on the MSP430 .......... 27
    2.4.2 Artificial Neural Networks ................... 29
    2.4.3 Convolutional Neural Networks ................. 30
    2.4.4 Visual Feature Extraction on PULP .......... 31
3 Embedded BNN Enabling Sound Event Detection

3.1 Introduction

3.2 Related Works

3.3 Feature Extraction and BNN

3.4 Embedded Implementation

3.5 Experimental Results

3.6 Conclusions

4 Extending the RISC-V ISA for Efficient RNN-based 5G Radio Resource Management

4.1 Introduction

4.2 Related Works

4.2.1 Generic Software-Programmable Platforms

4.2.2 ML Compute Platforms

4.2.3 RISC-V and RI5CY
5.5.1 Real Applications .......................... 129
5.5.2 Comparison with State-of-the-Art ............ 133
5.6 Conclusion .................................. 134

6 XNORBIN: BNN Hardware Acceleration 137
6.1 Introduction ................................. 137
6.2 BNN and related HW optimization ............... 138
6.3 Architecture ................................ 141
  6.3.1 Data Organization and Data Reuse .......... 145
  6.3.2 Scheduling .............................. 146
6.4 Scalability ................................ 149
6.5 Results .................................... 151
  6.5.1 Physical Implementation .................. 151
  6.5.2 Experimental Results .................... 151
6.6 Analysis Summary ............................ 155
6.7 Conclusion ................................ 155

7 Hyperdrive: Solving the I/O Bottleneck in BWN HW Accelerators 159
7.1 Introduction ................................ 160
7.2 Hyperdrive Architecture ....................... 162
7.3 Computational Model .......................... 166
  7.3.1 Binary Weights for Residual Networks .... 166
  7.3.2 Principles of Operation .................. 169
  7.3.3 CNN Mapping ............................ 170
  7.3.4 Supported Neural Network Topologies .... 175
7.4 Scalability to Multiple Chips .................. 177
  7.4.1 Access Pattern and Storing Scheme of the Border Memories ..................... 179
  7.4.2 Border and Corner Exchange ............... 180
  7.4.3 Border and Corner Memory ............... 180
  7.4.4 Interface Implementation ............... 181
7.5 Experimental Results ........................ 182
## CONTENTS

7.5.1 Implementation Results ........................................... 183  
7.5.2 Benchmarking ..................................................... 187  
7.5.3 I/O in Multi-Chip Setup ......................................... 188  
7.5.4 Comparison with State-of-the-Art ......................... 191  
7.6 Conclusion ............................................................ 193  

8 Summary and Conclusion ............................................ 195  
  8.1 Overview of the Main Results .................................. 196  
  8.2 Outlook ............................................................. 200  

A Notations and Acronyms ............................................ 203  
  Operators ............................................................... 203  

Bibliography ............................................................. 209  

Curriculum Vitae ........................................................ 231
Chapter 1

Introduction

The machine learning field has seen a veritable breakthrough avalanche within the last few decades, driven by the extensive newly available compute capability, public accessibility to large and diverse datasets, and easy-to-use deep learning frameworks like Tensorflow, Torch, Caffe. Especially, Convolutional Neural Networks (CNNs) or Deep Neural Networks (DNNs) have revolutionized computer vision and data analytics in a broad spectrum of applications and challenges [2]:

- Image Classification starting from small images (e.g., handwritten writing [3], traffic signs [4]) to high-resolution images [5–8]
- Object segmentation/detection [9–11], and face detection [12]
- Natural language processing [13,14], speech recognition [15,16] and text understanding [17,18], as well as video analysis [19,20]
- Artificial intelligence in games [21–23]
- Self-driving cars [24]
- Automated surveillance, personalized advertising [25], augmented reality applications
- Mobile communication [26,27] and many more.
Important milestones for the new AI era have been set, first in 2012 as AlexNet reduced the Top-5 error from 26% to 15.3% [6] on the prestigious ImageNet Large-Scale Visual Recognition Challenge (ILSVRC), secondly in 2015 when ResNet-101 surpassed human-level performance on this same challenge [8], and thirdly in 2016 when DeepMind’s AlphaGo beat the champion Lee Sedol in the game Go which was believed to be unlearnable for a machine due to the extreme size of state/action space [21]. Following these milestones, huge investments have been made in industry and research, thus in 2018 the AI-derived business value surpassed the threshold of 1 trillion USD (i.e., $10^{12}$) [28], and the number of Artificial Intelligence (AI) and Machine Learning (ML) related research papers increased by 6.6 times from 1998 to 2017 [2, 29]. Whereas Moore’s law introduced a doubling of compute capabilities every two years for the last decades, there has been seen a doubling of required compute resources every 3.4 months for training of large-scale machine learning tasks (e.g., AlphaGoZero with 158 million PFLOP [30]), a 300’000 times since 2012 [31], while the cost per operation just halves every 18 months [32, 33], which obviously led to an exponential increase of costs. Opposite to this
high-performance and allegedly resource-unconstrained ML branch, there have been seen the trend towards Internet of Things (IoT), where connected sensor nodes are becoming ubiquitous in our world in the form of wearables, smart home, smart networks, digital health, and always-on cameras (e.g., face/angle detection on smartphones) and many more. Recently, there has also been formed an entire new community for machine learning on these devices, namely the TinyML movement with its annual TinyML Summit [34]. TinyML devices come with tight restrictions in power, energy, memory, and compute capabilities, which is totally orthogonal to the requirements of the trend of new state-of-the-art ML algorithms. Already now, Neural Network models require hundreds of watts for inference, hundreds of megabytes, and billions of complex floating-point operations [35]. Off-loading computation to the cloud is a common strategy, but actually not a reasonable option due to privacy concerns, latency, reliability (i.e., network connections), scalability, and energy-costly data transmissions. Thus, for enabling the same success of Neural Networks also on IoT devices, there are three directions with a strong interdependency:

1) Development of Neural Network Topologies and Algorithms that fit the requirements of IoT devices and hardware architectures (i.e., memory footprint, computational requirement, and compute unit) which will be suggested in 2) under performance requirements (e.g., acceptable accuracy for the specific use-case or latency constraints).

2) Co-Design of energy-efficient hardware architectures supporting network topologies suggested in 1) while exploiting potential energy optimizations.

3) Define algorithmic requirements and preferable properties for novel network topologies based on the design of AI hardware and based on possible hardware architectures and optimizations.

From the algorithmic side (i.e., point 1)), there have been presented several approaches:

- optimized network topologies while reducing the memory footprint [6,36–39],
CHAPTER 1. INTRODUCTION

• replacing filters with smaller kernels (i.e., 1×1 kernels [37]),
• reducing the (input) channels [37],
• increasing the number of zero weights and sparsity [40],
• learning a subset of weights and storing indices only (i.e., weight sharing [41]),
• exploiting algebraic properties, e.g., using block-circulant matrices [42],
• splitting input channels into groups and calculate convolution layers per group, and shuffle channels after every layer (i.e., ShuffleNet [39]),
• splitting convolution layer into depthwise convolution, where every input channel is convolved with a single filter per output channel, followed with a 1×1 (point-wise) convolution layer to determine how much every input channel contributes to the output channel [38],
• and using light-weight fixed-point operations instead of Full-Precision Floating-Point (FP32) while reducing the arithmetic precision [43].

On the hardware side, there have been plenty of work optimized software for mainstream systems like HPC [44], CPUs [45], and GPUs [46,47], and new algorithms like FFT-based approaches and Winograd convolutions improved the throughput and energy efficiency for the machine learning workload [48,49]. However, these implementations still cannot fulfill the power constraints imposed by mobile/IoT end-node devices. In parallel, General-purpose processors and GPUs have been extended with new matrix and vector Instruction Set Architecture (ISA) extensions to handle the common compute patterns in Neural Networks, and introduced the support of Half-Precision Floating-Point (FP16) and diverse fixed-point formats [50,51]. Nvidia launched the Nvidia Tegra series, a system-on-chip mastered by ARM cores with a tightly attached embedded GPU. Furthermore, new easy-to-use tools for embedded microcontrollers to map from ML frameworks to firmware implementation have been developed like TensorFlowLite [52], Keras-based STM32CUBE.AI, and CMSIS-NN [53].

FPGA implementations still have the flexibility needed for new ML advances and already significantly boost the performance and
energy efficiency [54], but are still too expensive for large-distribution of devices and are also known to be at least one order of magnitude worse than custom ASICs [55]. Therefore, there has been plenty of new AI processors presented in research (e.g., [56–74] and many more) and from industry in recent years; namely from Google (TPU [75]), Alibaba (Hanguang), Cerebras (wafer-scale chip [76]), Graphcore (IPU), Habana (Gaudi), Qualcomm (Snapdragon), Huawei (Kirin), Intel (Nervana) and many more. Still, most of these AI processors rely on high-precision computations and floating-point number format, and therefore lack the energy efficiency needed for smart applications on IoT end-nodes.

Nevertheless, the higher energy efficiencies and throughput come with a price, which is the loss in flexibility to adapt to the very fast-changing AI research field. Furthermore, custom ASIC accelerators have very high non-recurrent costs, which can just be compensated with a high number of sold chips. Thus the efficiency vs. flexibility trade-off (as illustrated in Fig. 1.2) has to be evaluated carefully for every single use case.
CHAPTER 1. INTRODUCTION

1.1 Outline

This thesis tackles the problem of energy-efficient AI for embedded systems and IoT and its efficiency/flexibility trade-off for different uses-cases: from efficient embedded system design and SW-level optimization over Application-Specific Instruction-Set Processor (ASIP) to full-custom ASIC Accelerators. The organization is illustrated in Fig. 1.3 and is described in the following:

Starting from the Embedded Systems Level Design, Chapter 2 introduces context-recognition on a low-power smartwatch\footnote{Based on Andri's master’s thesis [77] and in part published by Magno et al. [78]}. The system is based on low-power sensors, sensor fusion, a computationally light-weight algorithm (i.e., decision-tree algorithm C4.5), and a small neural network running on the low-power multi-core PULP which has been attached as an accelerator to the MSP430 microcontroller. Context classification over five classes with high accuracy is enabled (84%) within 2.2 mJ or 64% within 91 µJ. In a follow-up work, presented in Chapter 3, we train a network with highly-quantized neural networks (i.e., Binary Neural Network (BNN)), while loosing not more than 7 points in accuracy. Furthermore, we show an efficient way to implement these networks on a low-power embedded compute platform for acoustic event detection.

In Chapter 4, we are looking into processor extensions in the field of Radio Resource Management (RRM). As the research field is very active, network types and topologies are frequently changing. As a first step, we define a representative benchmark suite for neural networks applications in RRM. Then we implement them efficiently for the RISC-V instruction set, optimize it for the existing RI5CY ISA extensions, and introduce novel instructions like hyperbolic tangent and sigmoid instructions and a concurrent load and compute instruction to improve throughput further.

Then, the thesis continues on the design of standalone custom accelerators for highly quantized neural networks. Even though BNNs are very promising, they still lack in performance. E.g., the AlexNet-BNN has an 11% worse Top-5 accuracy than the full-precision equivalent [79], therefore in chapter 5, we concentrate on accelerating
the more robust binary-weight neural networks. YodaNN is the first BWN accelerator in the literature, and also the first one running highly-quantized neural networks in general. Binary-Weight multiplication becomes simple sign inversion and addition. By exploiting high data re-use, energy-efficient latch-based memories, multi-path adder trees (for different kernel sizes), a core energy efficiency of up to 61.2 TOPS/W has been achieved which was 32x better than state-of-the-art neural network accelerators. But due to rather small memories and thus high I/O bandwidth requirements, the device efficiency is still rather low with 1 TOPS/W.

In chapter 6, we show the design of a fully-binary neural networks accelerator (i.e., BNN), with the help of very simple logic like XNOR gates and simple adder trees (popcount and accumulate). Thanks to the massive reduction in data size (up to 32x for the intermediate feature maps and parameters), a binary AlexNet can be fit onto a 1 mm\(^2\) chip. At the same time, the weights are streamed once to the chip and a very high device energy efficiency of 27 TOp/s/W and core energy efficiency of 205 TOPS/W, has been achieved.

Chapter 7, presents Hyperdrive, where the I/O problem from YodaNN is tackled. The design is significantly different from previous architectures. It is input and output (feature map) stationary and consists of a systolic 2D mesh of processing units which operate on spatial tiles. The memories are attached to the PEs, and tile borders are exchanged between neighboring processing units. Furthermore, motivated by the generally limited silicon area, the systolic design also allows to extend to a multi-chip systolic system, where neighboring pixels are exchanged among neighboring chips, and stored in border memories (i.e., for single data exchange). A system-level energy efficiency of 4.3 TOPS/W has been shown which is 3.1\(\times\) higher than previous state-of-the-art BWN accelerators.

In the conclusion chapter, the overall results from software to hardware level are discussed and reasoned. Furthermore, this chapter gives an overview of ongoing research and potential future trends.
### 1.2 Contributions and Publications

As described before, the thesis discusses the design of AI systems for constrained devices for the three high-level use-cases Embedded Systems, Application-Specific Instruction-Set Accelerator and Custom ASIC design. In the following, the contributions of this thesis are listed and explained more in detail:

#### On the Embedded System Side (Chapters 2 and 3):

- We enable smart context recognition on a smartwatch, using a very low-light decision tree, combined with a tiny neural network for visual features, while keeping the overall system within a tight 10 mW power consumption, and a 2.2 mJ classification cost,

- Achieving an accuracy of 84% on our own small dataset, enabling ego-vision applications without the need of external communication, and therefore providing huge savings in energy.

- We show a near state-of-the-art acoustic event detection results while training and implementing highly-quantized BNNs on a

---

**Figure 1.3: Thesis Overview**

<table>
<thead>
<tr>
<th>Domain</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded Domain</td>
<td>2: Energy-Efficient Design of Embedded Context Recognition</td>
</tr>
<tr>
<td>Application-Specific ISA</td>
<td>3: Embedded BNN for SED</td>
</tr>
<tr>
<td>Full-Custom HW Accelerator</td>
<td>4: RNN ASIP for RRM</td>
</tr>
<tr>
<td></td>
<td>5: YodaNN: BWN Acceleration</td>
</tr>
<tr>
<td></td>
<td>6: XNORBIN: BWN Acceleration</td>
</tr>
<tr>
<td></td>
<td>7: Hyperdrive: Solving the I/O Bottleneck in BWN HW Accelerators</td>
</tr>
</tbody>
</table>
low-power compute platform (i.e., Gapuino). We show that BNNs render this application possible, due to 28× reduction in memory requirements, and the low-level computation optimizations based on xor and popcount operations, an average board-level energy efficiency of 34.5 GOp/s/W is shown.

- Exploiting the capabilities of the low-power DSP-enhanced PULP platform (i.e., GAPuino), we show a 10× faster and 51× more energy efficient performance on the BNN inference compared to the ARM Cortex-M4F platform, which comes from multi-core capabilities (i.e., 7.2/2.6×), the build-in popcount instruction (i.e., 2.5/2.9×), and other low-power capabilities (e.g., latch-based memories).

**Application-Specific Instruction-Set Processor (Chapter 4):**

- We show how to improve the throughput and energy efficiency of recurrent neural networks on a RISC-V core in the field of Radio Resource Management (RRM). Exploiting the existing RI5CY Extension, including hardware loop, post-increment loads, and SIMD instructions, a 4.4× higher efficiency is shown.

- The ISA is extended by a custom hyperbolic tangent and sigmoid instruction for the compute-intensive activation functions in Recurrent Neural Networks (RNNs) and Long Short-Term Memorys (LSTMs). Based on the piece-wise linear approximation, an in-depth evaluation is evaluated for good accuracy and energy efficiency trade-off. A 1.13 times increase in efficiency/throughput with a little increase of 3% in core area is achieved.

- We show how we efficiently re-use data by tiling feature maps an other 1.9× improvement in throughput/energy efficiency is achieved.

- Finally, we introduce a new instruction combining load and compute in a single-instruction, which improved the overall performance by 1.8× or 1.2× in throughput or energy efficiency.

- We show how to combine software optimizations and ISA-level HW extensions in real RRM benchmark networks to reach
efficient performance while keeping the flexibility to adapt to new algorithmic development, whereas the hardware infrastructure cannot be updated so frequently.

**Custom ASIC accelerators (Chapters 5, 6 and 7):**

- We present the first CNN accelerator exploiting binary weights, achieving an energy efficiency of 61.2 TOp/s/W.

- Binary weights are exploited by replacing floating-point multiply-accumulate with sign-inversion and optimized adder trees.

- We present an efficient Latch-Based Memory architecture reducing the energy cost for data access by $3.5 \times$ and enable voltage scaling down to the logic limit.

- We present an efficient way of how data can be efficiently re-used while using a sliding window approach, and by exploiting efficient adder trees while keeping flexibility for a broad variation of CNN kernel sizes.

- With large-scale image problems, YodaNN’s energy consumption is dominated by I/O bandwidth. Therefore, we develop a second binary-weight neural network accelerator. Hyperdrive has a novel approach to efficiently distribute the computation of binary-weight CNNs on a two-dimensional array of chips. While reducing the I/O bandwidth by up to $58 \times$, the system-level energy efficiency is improved by $3.1 \times$ to 4.3 TOp/s/W.

- We further propose a fully binary neural networks accelerator reaching 205 TOp/s/w, while exploiting simple XNOR operation and accumulation, latch-based memories.

Most of the content of the thesis has been published in the following journals and conferences:


Original conference papers with journal extensions included in this thesis:


Other contributions not included in this thesis:


Chapter 2

Energy-Efficient Design of Embedded Context Recognition

This and the next chapter investigate how to enable machine learning applications in a combination of embedded system design and software development for two typical applications for wearables: context recognition and sound event detection. Unfortunately, wearable devices currently on the market are either incapable of complex functionality or severely impaired by short battery lifetime.

In this chapter, we present a smartwatch platform based on off-the-shelf components based on an ultra-low-power (ULP) heterogeneous system composed by a Texas Instruments MSP430 microcontroller, the PULP programmable parallel accelerator and a set of ULP sensors, including a camera. By using an optimized classification algorithm based on the C4.5 decision tree, the smartwatch is able to transform the collected sensor data into context-aware higher-level information. The embedded PULP accelerator enables state-of-the-art context classification based on Convolutional Neural Networks (CNNs) to be applied within a sub-10mW system power envelope. Our methodology enables to reach high accuracy in context classification over 5 classes
CHAPTER 2. EMBEDDED CONTEXT RECOGNITION

(up to 84%, with 3 classes over 5 reaching more than 90% accuracy), while consuming 2.2mJ per classification, or an ultra-low energy consumption of less than 91µJ per classification with an accuracy of 64% - 3.2× better than chance. Our results suggest that the proposed heterogeneous platform can provide up to 500× speedup with respect to the MSP430 within a similar power envelope, which would enable complex computer vision algorithms to be executed in highly power-constrained scenarios.

2.1 Introduction

A fast-growing class of highly power-constrained devices which can profit from machine learning is smart wearables, where electronics and sensors are tightly coupled with the human body [91]. This paradigm proposes to transform everyday life objects such as wristwatches, necklaces, and glasses in “smart” objects that look promising for a plethora of applications: such as sports and fitness, augmented reality, and personalized health care. Moreover, top-tier hi-tech companies such as Google, Samsung, and Apple look at wearable devices as a new high growth segment in the consumer market. Smart wearable devices open up new possibilities in terms of context awareness [92], making all devices more conscious of their environment and therefore more “intelligent”. Continued miniaturization and power improvements have eased the construction of a wide variety of wearable multi-sensor systems [93]. In fact, some forecasts preview up to a trillion connected devices, which are going to produce a massive amount of data [92]. Even with so many sensor-rich wearables, however, the sheer amount of data alone will not provide any value, unless it is possible to turn it into actionable, contextualized information. Machine learning technologies are used with great success in many application areas, solving real-world problems in entertainment systems, robotics, health care, and surveillance [94]; they are incredibly flexible and can be applied to heterogeneous data. However, due to their massive requirements in terms of memory and computational throughput, these high accuracy techniques are currently considered to be too computationally expensive for the limited capabilities of wearable
2.1. INTRODUCTION

devices. Instead, sensory data is transmitted to servers “in the cloud” [95] at a high cost in terms of latency and transmission energy.

At the same time, one of the main limitations of the current generation of wearable devices is autonomy, due to the limited amount of energy that can be stored in the batteries. Continuous transmission of data is expensive in terms of energy and severely hinders the autonomy of these devices, posing a practical limit to the amount of useful information that a wearable device can send to the cloud for processing. An alternative approach is that of partially performing the processing locally to the wearable node so that what is sent out via wireless communication is data in a high-level format (such as visual features) and of reduced dimensionality. This is a major challenge for a typical low-power wearable device driven by a low-power microcontroller unit (MCU). Off-the-shelf MCUs are orders of magnitude less powerful than it would be necessary to sustain data classification using state-of-the-art machine learning techniques [94,96]. As a possible solution to this challenge, parallel programmable accelerators have been proposed [97,98] as a means to obtain the necessary level of performance while keeping the power envelope controllable. Accelerators for wearable computers need to perform a variety of tasks and algorithms to fuse data coming from several sensor sources. To provide the necessary level of performance and energy efficiency for this class of algorithms, it is necessary to use deeply integrated technologies that come with high engineering and manufacturing costs. As a consequence, accelerators need to be flexible,

1) to be coupled to many different host devices (e.g. MCUs) and

2) to be applied to a very wide range of scenarios, enabling cost-efficient economy of scale.

One of the target applications for wearable devices is that of ego-vision, i.e., vision using a first-person video stream as the primary source of information. Ego-vision enables use cases such as gesture recognition for augmented reality with off-the-shelf smartphones [99] or a Google Glass device [100], sign recognition to assist people with visual impairments [101], on top of applicative scenarios such as assisted living (fitness, entertainment, etc.) [102], health-care assistance [103], adaptive environments [104], and Internet of Things ecosystems [105].
For example, an ego-vision system can be used to achieve a multi-node assisted environment (e.g., house, car, gym, office, etc.) where complex multi-device behavior is triggered by an “intelligent device” always aware of the user’s activity [104]. In activity monitoring [103], the final effect is either to affect the real world (e.g., turn on lights) or to inform someone that a monitored incident has happened (e.g., fall detection). As all of the mentioned scenarios are time-critical applications, fast computation plays an essential role in fast “detect and act” capability [102] — on-board computation can provide a definite advantage by minimizing latency.

In this chapter, we propose a low-power platform for wearable computing and ego-vision based on a heterogeneous system composed by a Texas Instruments MSP430 microcontroller and an ultra-low power parallel accelerator, the PULP3 chip. The system is equipped with ultra-low power sensors: an analog camera, a microphone, accelerometer, and temperature sensors. We deploy this platform on a wearable smartwatch device. The proposed approach enhances the application scenarios where on-board processing (i.e., without streaming out the sensor data) enables intensive computation to extract complex features. The smartwatch platform forms a challenging environment for vision due to lighting, obstruction, and continuous motion. We show that by using a light-weight decision tree algorithm combined with a CNN, enabled by the low-power multi-core PULP platform, it is possible to extract meaningful information even in this case. Our claims are:

1) that the availability of more computing power enables the extraction of more complex features out of the same simple ultra-low-power sensors; and

2) that our platform can support workload orders of magnitude more complex than what can be supported by current off-the-shelf wearables, within a similar power envelope.

3) that a high energy efficiency can be reached for always-on context recognition while using light-weight decision tree algorithm enhanced with a CNN for visual context.

The remainder of this chapter is organized as follows: Section 2.2 describes recent related work in the area. Section 2.3 details the
2.2. RELATED WORK

proposed system architecture. Section 2.4 describes the context classification approach. Section 2.5 described experimental results with measurements, simulations, and validation. Section 2.6 concludes the chapter.

2.2 Related work

Due to the need for performance that is typical of many approaches based on machine learning, most research on wearable sensor systems has focused on smartphones, that provides an ideal platform from this point of view as they provide a personal portable, sensor-rich and powerful computing platform [106–108]; they can also be used as a hub for a network of smaller sensors. Using the MEMS sensors embedded in most modern smartphones, it is possible to perform tasks such as activity recognition, crowdsensing and fall detection with great effectiveness [96,109], using classification techniques such as decision trees, k-nearest neighbors, support vector machines (SVMs), naïve Bayes and neural networks [110]. For example, Porzi et al. [101] built a wearable system for gesture recognition to help visually impaired using a Sony Xperia Z smartphone and a Sony Smartwatch. They make use of an optimized kernel method (global alignment kernel) for discrete-time warping in SVMs, allowing to map similar gestures when moving at different speeds.

However, a smartphone-based wearable may not be the best choice, due to its limited battery duration and the requirement of wireless connection with the body sensors, non real-time operation (as it depends on the complex operating system running on the phone) and loose coupling with the body (e.g., it is easy to forget the phone anywhere). The main alternative for body sensing is based on low-power microcontrollers [91] that usually run either bare-metal code or a very small real-time operating system such as FreeRTOS. Examples of ultra-low power microcontrollers that are able to work in a power budget of less than 50 mW include the SiliconLabs EFM32 [111], the Texas Instruments MSP430 [112] series of MCUs, the Ambiq Apollo [113], and the STMicroelectronics STM32-L476 [114]. A typical approach is to employ a heterogeneous set of sensors such as accelerometers, acoustic sensors, gyroscopes and thermometers on the
human body to capture characteristic repetitive motions, postures, and sounds of activities [115] that can then be used for context classification.

Many wearable systems do not include cameras because it is difficult to extract meaningful data out of them while keeping a very tight power and energy budget. On the other hand, it is well known that cameras are a very effective source of information regarding one’s own body [99, 116], especially taking advantage of the preferential ego-vision point of view. To exploit this richness under the tight energy constraints, it is necessary to couple a very efficient imaging sensor with a computing platform that can provide enough throughput to extract significant information out of the frames. Research on ultra-low power cameras focuses on relatively small gray-scale imagers [117–119]. These cameras often output analog pixels, needing an external ADC to convert the frames to the digital domain, and complicating the classification task due to the amount of noise. This further strengthens the need for a relatively high-performance computing platform to be embedded in the sensor node.

To try and overcome the energy efficiency limitations of current commercial ultra-low-power platforms, researchers have to extract as much energy efficiency as possible out of silicon. A well-known approach is near-threshold computing, which exploits the fact that CMOS technology is most efficient when operated near the voltage threshold, where delay and dynamic power are simultaneously small, and therefore total energy per operation is minimal [1]. For example Ickes et al. [120], SleepWalker [121] and Bellevue [122] show examples of near-threshold ultra-low power microcontrollers, with the latter also exploiting SIMD parallelism to improve performance.

Microcontrollers can also exploit accelerators as specialized DSPs [123] and ASICs [91,124] to achieve a higher level of performance; however, such approaches are very limited in flexibility, which negatively impacts economy of scale and cost. Instead, a key enabler to achieve high performance with little or no sacrifice to flexibility is parallel computing, which is an attractive option for highly parallel workloads such as those of computer vision. Operating multiple cores in parallel allows for the inherent data- and task-parallelism of the algorithm at hand to be exploited, while the energy costs of the platform are partially shared between the cores improving overall efficiency. Traditionally,
in the embedded world, parallelism has been exploited by means of special-purpose DSPs relying on SIMD or VLIW. Two examples are the Qualcomm Hexagon DSP [125] that accelerates a Snapdragon 800 with VLIW DSPs and is effective for vision and context inference tasks [126], as well as the Neon SIMD extensions that are integrated into many ARM cores [127]. All these platforms, however, are not meant to couple with a low power microcontroller, as they are designed for high end embedded architectures with DRAM, memory management and complex operating systems with power budget in the hundreds of milliwatts at chip-level, up to a few watts at system level.

Table 2.1 shows an overview of some state-of-the-art activity recognition works. The proposed algorithms target fall detection using the camera sensor as the main device, coupled with low power computational resources. In contrast with our work, neither of the two architectures is based on a low-power microcontroller. CITRIC [128] is based on the Intel XScale microarchitecture (with ARMv5 ISA) running at about 600 MHz. It was initially developed as a standalone video processing node. Exynos 5410 Octa [129] is a commercial system-on-chip by Samsung that can be found in several smartphones such as the Samsung Galaxy S4. It is based on an ARM big.LITTLE architecture and contains 4 Cortex-A7 and 4 Cortex-A15 cores (with SIMD extensions) plus a PowerVR SGX544 GPU.

Compared to our work, the considered platforms require an order of magnitude more power, while targeting a similar class of algorithms in terms of computational requirements.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Architecture</th>
<th>Accuracy</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOG [103]</td>
<td>CITRIC [128]</td>
<td>87%</td>
<td>∼ 1 W [128]</td>
</tr>
<tr>
<td>Optical Flow [103]</td>
<td>CITRIC [128]</td>
<td>85%</td>
<td>∼ 1 W [128]</td>
</tr>
<tr>
<td>Erden et al. [102]</td>
<td>Exynos 5410 [129]</td>
<td>74%</td>
<td>∼ 3 W [129]</td>
</tr>
</tbody>
</table>

Table 2.1. Order of magnitude of power consumption and average accuracy in fall detection and activity classification for several related works.
More recently, research has been very active on the exploitation of intrinsic data and task parallelism with sub-100 mW multi-core platforms; by coupling parallel computing with low power techniques such as near-threshold computing, it is possible to maximize the overall energy efficiency of a platform. Fick et al. [97] propose Centip3de, a large-scale fabric of clusters of 64 Cortex M3 cores, integrated in a 3D matrix and clocked at a very low frequency of 10 MHz; it can reach a peak performance of 0.64 GOp/s. Another similar platform is DietSODA [130] that features 128 SIMD lanes working at relatively low frequency (50 MHz), reaching up to 6.4 GOp/s. On the commercial side, NXP has recently proposed an asymmetric dual-core microcontroller, the NXP LPC54100 [131], that couples a low-power Cortex-M0 for sensor control with a more powerful Cortex-M4 that can be seen as an accelerator.

Our work focuses on enabling high-level visual feature extraction in a low power wearable device. To this end, we augment a low power smartwatch platform with a parallel ULP programmable accelerator that was designed according to the two guidelines that were described with regard to the related work: near-threshold and parallel computing. Our first objective is to provide a platform that allows for efficient context classification using visual features at a low power and energy budget; moreover, we want to demonstrate how such a platform can enable many future developments in the fields of vision and ego-vision embodied in low power wearable devices.

### 2.3 Smartwatch System Architecture

This section describes the system architecture of the proposed smartwatch, whose high-level diagram is shown in Fig. 2.1. The smartwatch is composed of a low power micro-controller coupled with an ultra-low-power accelerator and a set of four different sensors: camera, microphone, accelerometer, and thermistor. The proposed architecture extends the smartwatch, which has been partly designed during Renzo Andri’s master’s theses [77], and has been published at the International Internet of Things Summit 2015 [89]. In this work, we extend this platform with a PULP processor, an ultra-low-power accelerator.
2.3. SMARTWATCH SYSTEM ARCHITECTURE

The main system runs on a 2 V power supply, powered by a power harvester BQ25570 from Texas Instruments. The power harvester is connected to a lithium-ion polymer rechargeable battery and can harvest from solar cells and thermal electric generators (TEGs). For the camera and for the microphone additional supply voltages are needed; the microphone is supplied at 1.2 V by a Linear Technologies LTC3406ES5-1.2 buck converter featuring only 1 µA leakage in active mode and the camera with a buck converter TPS62740 (with quiescent current of 460 nA) from Texas Instruments. In idle mode, all sensors can be switched off: camera and microphone are power-gated and controlled by the microcontroller. The accelerometer features a very low-power idle mode that can be set by the microcontroller and has wake-up by interrupt capability. During idle mode, the microcontroller can be put in ultra-low power mode or deep sleep, waiting respectively on SPI communication or alternatively on a pin interrupt.

2.3.1 MSP430 Core

The central core of the smartwatch is the 16-bit MSP430FR5969 microcontroller from Texas Instruments [112]. This microcontroller incorporates 2kB of SRAM and 64kB of non-volatile Ferroelectric RAM (FRAM), a novel memory technology which enables non-volatile memories with similar performance in terms of power and speed than
SRAM memories. The MSP430 is well known for its ultra-low power consumption as it supports several power modes (one active mode and seven low-power modes), enabling fine-grain control of which components of the MCU are active. Current consumption in active mode is 800 µA at a clock frequency of 8 MHz; this drops to 20 nA in low power mode LPM4.5, making this micro-controller one of the lowest power on the market, though its performance capabilities are severely limited at an ultra-low power operating point.

2.3.2 PULP Accelerator

To provide a boost to the classification capabilities of the smartwatch, in this work, we augment the smartwatch platform with an accelerator based on PULP for its scalability and its many-core compute capabilities. At the same time, it is designed to operate on a broad range of operating voltages, achieving in this way a high level of energy efficiency over a wide range of application workloads [98, 132].

In particular, we focus on PULPv3, the third embodiment of the PULP architecture; we emulated this version of PULP with a RTL-equivalent FPGA emulator based on a Xilinx Zynq Z-7045 device. The third PULP chip features a single quad-core cluster integrated...
with 128 kB of L2 SRAM memory and several IO peripherals accessible through a system bus such as two QSPI interfaces (one master and one slave), GPIOs, a bootup ROM and a JTAG interface suitable for testing. In our smartwatch platform, the MSP430 acts as an SPI master with respect to PULP, allowing to offload code and data and to control the accelerator. Additionally, two interrupt lines (one per direction) can be used to notify the accelerator or the host (respectively) of a notable event, e.g., to wake up the accelerator or to notify the host of the completion of an accelerated task. The architecture of the PULPv3 SoC is shown in Figure 2.2.

Internally, the PULP cluster is based on 4 OpenRISC ISA [133] cores with a power-optimized microarchitecture called Or10n [134,135] and a shared instruction cache ($I$). Or10n is the predecessor of the RISC-V ISA based RI5CY core, which will be used in Chapter 4. The Or10n core is enhanced with respect to the original OpenRISC reference implementation by adding a register-register multiply-accumulate instruction, vectorial instructions for arithmetic on short and char vectors, two hardware loops and support for unaligned memory access. To avoid the energy overhead of memory coherency, the cores have no data cache and no private L1 memory: they all share a multi-banked Tightly-Coupled Data Memory (TCDM) that acts as a shared scratchpad at L1. Communication with this memory is based on a low-latency interconnect that implements word-level interleaving, with the objective of reducing access contention to the TCDM [136]. The TCDM is further divided in SRAM and standard-cell memory (SCM) banks to allow the cluster to work at very low voltage [137]. A lightweight multi-channel DMA can be used for fast communication with the L2 memory and external peripherals [138]. The DMA features a direct connection to the TCDM to reduce power consumption by eliminating the need for an internal buffer. The PULP platform is fully programmable using the standard OpenMP programming model [98], which enables relatively easy implementation of parallel algorithms leveraging a low-overhead runtime.

To enable fine-grained frequency tuning: a Frequency-Locked Loop [139] and two clock dividers (one for the cluster and one for peripherals) are included in the SoC. All cores work at the same speed, but each can separately be clock-gated to reduce dynamic power or “boosted”
by means of a body bias multiplexer. This feature is integrated directly in the thread creation/destruction routine in the runtime to be fully transparent to the user. The cluster also contains a hardware synchronizer used to accelerate synchronization between the cores, making sure that they can be put to sleep and woken up in just a few cycles. Cores and peripherals in the PULP cluster are clock-gated when not in use to save dynamic power, and the cluster can be reversely body-biased also to reduce leakage when not in active usage.

![Figure 2.3: Power consumption and performance of MSP430, PULP and several commercial MCUs.](image)

Figure 2.3 clarifies in a quantitative way why PULP is a highly effective accelerator for highly power constrained microcontroller level systems. The plot shows the power consumption of several low-power MCUs (including the MSP430) and of PULP against their peak throughput in terms of operations per second. The operating points taken into account include all supply voltages from $V_{DD} = 0.5\, \text{V}$ to $V_{DD} = 1.0\, \text{V}$ in 100 mV steps. In the case of the MCUs, the operating points are chosen from those reported in their datasheets, while for PULP, they are those considered during power analysis (see Section 2.5).

Figure 2.3 takes into account 4 state-of-the-art low-power microcontrollers: Texas Instruments MSP430 [112], SiliconLabs EFM32 [111],
Ambiq Apollo [113] and STMicroelectronics STM32-L476 [114]; the latter two feature a relatively powerful ARM Cortex-M4 core.

### 2.3.3 Sensors

The smartwatch hosts four different sensors. The first sensor is an ultra-low-power analog gray-scale $112 \times 112$ *Centeye Stonyman* CMOS camera [119], which has a focal plane size of $2.8 \text{mm} \times 2.8 \text{mm}$ and a pixel pitch of $25 \mu \text{m}$ in an active power envelope of $2 \text{mW} \@ 3.3 \text{V}$ (with quiescent power as low as $30 \text{nW}$). The camera can take a new picture every $\sim 50 \text{ms}$. The brightness values of each pixel are read out row by row while the pixel address is changed by short pulses on the control input pins. As the camera is intended for ultra-low-power application, the camera does not do any on-chip preprocessing (e.g., automated exposure adjustment). The camera comes on a pre-soldered PCB containing the image sensor and a lens and is connected to the smartwatch by a socket connector. The camera is plugged directly to the PULP vision accelerator via an *ADS7042* ADC, as shown in Figure 2.1, while the other sensors are plugged to the MSP430 microcontroller via SPI (accelerometer) and the internal ADC of the MSP430 (microphone, thermometer).

The accelerometer is an ultra-low power *ADXL362* from Analog Devices with high resolution (down to $9.8 \text{mm/s}^2$). While sensing at $100 \text{Hz}$, it needs $1.8 \mu \text{A}$ at a supply voltage of $1.8 \text{V}$, which are reduced to $10 \text{nA}$ in standby mode. The accelerometer features a burst mode, including a FIFO buffer, that allows storing the acquired sensor data inside the sensor while keeping the MCU asleep. To connect the MCU to the accelerometer, the SPI interface is used with the addition of two status signals that can be used to interrupt or wake-up the microcontroller, e.g., when acceleration exceeds a predefined threshold or the FIFO buffer is full. As a microphone, the smartwatch board includes the low power *INMP801* which was mainly designed for hearing aids and consumes $17 \mu \text{A}$ at a supply voltage of $1.2 \text{V}$, with an output voltage in the range of $410 \text{mV}-730 \text{mV}$. The audio signal is amplified by a *TI LMV951*, connected to the internal ADC of the MSP430, which is set to sample the audio signal at $8 \text{kHz}$. Finally, the temperature sensor is a Negative Temperature Coefficient Thermistor
(NTC) from Epcos/TDK used in a voltage divider configuration and is also connected to the ADC of the MSP430. The temperature sensor is directly supplied by an output pin from the microcontroller such that power is only consumed when the temperature is measured, and no additional load-switch is needed.

2.4 Context Classification

In this section, we describe the techniques that were used to extract features out of the various sensory data and to classify it in one of several contexts. As target platforms, we consider both the non-accelerated smartwatch [77, 89] and the accelerated version we described in Section 2.3. As a demonstration of a context classification application, we used the features extracted to infer whether the smartwatch user is in one of five “contexts”: morning preparation, walking outdoors, public transportation, in the car and in the office. The full dataset used for training the classifiers comprised ~35000 data points, each including an image acquired from the Stonyman camera and data from the other sensors. The dataset was collected by a total of 15 people wearing a smartwatch prototype for a combined total of 15 hours in different contexts corresponding to the five classes. All data was fed within the various algorithms we describe in Sections 2.4.1 and 2.4.4 with no preliminary preprocessing. Figure 2.4 shows an example data point for the temperature, accelerometer and camera sensors.
2.4.1 Feature Extraction on the MSP430

The first step of context recognition is extracting features out of raw sensor data. To this end, the data are fed into an algorithm that collapses it into a compact feature space by means of a reduction operation; one of the most straightforward conceivable features is, for example, the average of all inputs. Most algorithms, such as SVMs and CNNs, use a more sophisticated technique to extract features, by first projecting the input data into an intermediate high-dimensional space where the selected features are linearly separable and can be more easily extracted. If the features are selected correctly, the final classifier (e.g., the context classifier in our case) can be simpler and more effective; however, in the case of the proposed smartwatch, it is necessary to trade off the necessity to extract high-level features against the limited available computing capability and stored energy.

**Camera**

Vision sensors in a smartwatch can potentially produce a huge amount of useful data on the person wearing it. However, extraction of high-level features is not possible on low power microcontrollers used in wearable devices, as the MSP430, due to the computational burden of complex feature extractors used in the machine vision field. As a consequence, we consider only very simple features to be computed on the MSP430. In the context of this work, we consider three features: pixel average intensity, intensity variance and max-min difference.

**Accelerometer**

The accelerometer is widely used in many applications, being generally recognized as one of the most important sensors providing contextual information; when mounted on a smartwatch, it can be used to distinguish the type of activity that the user is doing (e.g., drinking a coffee, typing), and hence the most probable context he is in. For each of the acceleration directions, we define two main features: energy, defined as the cumulative square sum of acceleration over a window of samples; and acceleration entropy, defined as

$$H_{\text{accel}} = \sum_{i=0}^{N-1} (|\hat{a}_i| \cdot \log_2 (\hat{a}_i))$$  \hspace{1cm} (2.1)
where \( \hat{a} \) is the normalized acceleration. And the dynamic range defined as:

\[
\text{dyn\_range} = 1 - \frac{\min_{x \in X} x}{\max_{x \in X} x}
\]  

(2.2)

**Microphone**

The microphone is a powerful sensor to distinguish one context from another, because every environment can differ in its audio characteristics. First, we consider *zero-crossing rate* on frames of the duration of 0.5 seconds, as a first-order approximation of the tone pitch. Then we use the average signal energy

\[
\bar{E} = \frac{1}{W} \sum_{i=1}^{W} s_i^2
\]  

(2.3)

and the dynamic Range

\[
\text{dynRange} = 1 - \frac{\min_{x \in X} x}{\max_{x \in X} x}
\]  

(2.4)

To calculate features in the frequency domain, a frame-wise discrete Fourier transform (DFT) is applied. Then different metrics were used: The first is spectral centroid and is calculated using the Equation (2.5).

\[
c_i = \frac{\sum_{u=0}^{M} u \cdot |f_i(u)|^2}{\sum_{u=0}^{M} |f_i(u)|^2}
\]  

(2.5)

The second one is bandwidth (2.6) which is calculated using the spectral centroid [140].

\[
b_i^2 = \frac{\sum_{u=0}^{M} (u - C_i)^2 \cdot |f_i(u)|^2}{\sum_{u=0}^{M} |f_i(u)|^2}
\]  

(2.6)

Furthermore, the spectral roll-off frequency was calculated which differs strongly in segments when people speak from those when nobody speaks. Also music and noise strongly differ in this metric and is shown in Equation 2.7. [141]

\[
\arg \max_h \left( \sum_{u=0}^{h} f_i(u) < TH \cdot \sum_{u=0}^{M} f_i(u) \right)
\]  

(2.7)
2.4. CONTEXT CLASSIFICATION

The other features depend on a frequency domain representation of the audio signal; we used a 1024 point Fast Fourier Transform (FFT) both as a feature itself and to compute a set of higher level features: 16 Mel Frequency Cepstrum (MFC) coefficients [142], which represent the human ear perception of a given physical frequency and will be introduced and described more in detail in Section 3.3.1.

Temperature

Temperature helps to distinguish outdoor from indoor environments in a given season. Moreover, the corresponding sensor has by far the lowest power consumption, which makes it even more attractive. The only feature of interest we considered is the average over a window of samples.

2.4.2 Artificial Neural Networks

Even though Neural Networks have a long history, they have shown their real big break-through in the last few decades, driven by the availability of high compute capabilities and accessibility to large datasets, which made complex models more than ever trainable. Artificial Neural Networks are brain-inspired models, which traditionally consist of a set of input neurons \( x \in \mathbb{R}^k \), hidden units \( h \in \mathbb{R}^l \) and output neurons \( y \in \mathbb{R}^m \) and the neurons are connected by synapses (or vertices), each of which is assigned a weight value \( W_h(k,l) \) and \( W_y(l,m) \) which is the contribution of the input neurons to the output neurons. The activation or transfer function \( \sigma_h \) and \( \sigma_y \) (e.g., Rectified Linear Unit ReLU\(^1\)) is used to determine the state of \( h \) and \( y \) based on its input neurons and introduces non-linearity in the network which enables to learn non-simple tasks. The most basic neural network block is the Multi-Layer Perceptron (MLP) which has at least three layers: an

---

\(^1\)The ReLU activation function can be expressed as follows: \( \sigma_{ReLU}(x) = \max(0,x) \)
input and output neuron layer and a (fully-connected\(^2\)) hidden layer and can then be represented as follows:

\[
\begin{align*}
    h &= \sigma_h(W_h x + b_h) \\
    y &= \sigma_y(W_y h + b_y)
\end{align*}
\]  

(2.8) 

(2.9)

Feedforward Neural Networks have no dependency on previous samples and therefore do not have any cyclic paths in the network, which makes them (comparably) simple to train. Typically several hidden layers are stacked together and complemented with a final loss layer.

### 2.4.3 Convolutional Neural Networks

A special case of Neural Networks are CNNs, where the spatial dependency (translation invariance and correlation of neighboring pixels) of the input and hidden units in 2-dimensional data like images is exploited and a convolution kernel is learned per input and output neural channel instead of a weight for every neuron. A channel represents here a two-dimensional set of neurons with spatially local connections between intermediate channels. CNNs are typically composed of several neural network layers, whereas the main building block are Convolution Layers which can be formulated as a mapping from the 3D input Feature Map space (i.e., \( FM^{in} \)) of \( n_{in} \) channels with \( h_{in} \times w_{in} \) sized spatial dimensions to the 3D output Feature Map space (i.e., \( FM^{out} \)) of \( n_{out} \times h_{out} \times w_{out} \) size and can be described as follows:

\[
\mathbb{R}^{n_{in}\times h_{in}\times w_{in}} \xrightarrow{CNN} \mathbb{R}^{n_{out}\times h_{out}\times w_{out}}
\]  

(2.10)

\[
FM^{out}(c_{out}, \cdot, \cdot) = \beta_{c_{out}} + \alpha_{c_{out}} \sum_{c_{in} \in I_{n_i}} FM^{in}(c_{in}, \cdot, \cdot) \ast k_{c_{out}, c_{in}}(\cdot, \cdot)
\]  

(2.12)

Every single output channel \( c_{out} \) is calculated by convolving all input feature maps \( c_{in} \) with the corresponding filter kernel \( k_{c_{out}, c_{in}} \in \mathbb{R}^{h_k \times w_k} \), scaled by the factor \( \alpha_{c_{out}} \) and accumulated to a bias term \( \beta_{c_{out}} \).

\(^2\)All input neurons contribute to all output neurons
2.4. CONTEXT CLASSIFICATION

![Small CNN Architecture Diagram]

**Figure 2.5:** *small* CNN architecture for feature extraction on PULP.

### 2.4.4 Visual Feature Extraction on PULP

The availability of the PULP accelerator makes it possible to implement much more sophisticated feature extractors. In particular, the information coming from the camera is decidedly under-utilized in the MSP430 due to the sheer amount of computations that would be necessary to extract complex features from an image. Conversely, PULP is well-suited for the acceleration of vision kernels due to the amount of algorithmic parallelism available. In the accelerated smartwatch, we can afford to augment or replace the three features available for the camera (average, variance, max-min difference) with more complex algorithms.

In particular, we focused on a simplified version of a CNN, usually available in higher-level computer vision platforms. CNNs are state-of-the-art in many current visual classification, detection, and scene understanding benchmarks using big networks designed to run on relatively high-performance platforms such as GPUs [6, 143, 144]. However, in this case (as is shown in Figure 2.5), we consider a very small CNN architecture that begins with a substantial reduction in the dimensionality of the input (using a 4:1 max-pooling layer) to reduce the computational complexity of the model. Our CNN implementation is based the *CConvNet* library [129], that takes advantage of the OpenMP programming model for better performance on the parallel PULP platform.

### 2.4.5 Sensor fusion and Classification

The sensor fusion and classification stage is based on a Decision Tree (DT), one of the simplest and most widely applied supervised
classification techniques [145]. We selected this technique in particular because of the need for an algorithm with low computational complexity and high energy efficiency in inference, constraints that made the DT a suitable choice for our specific domain. We use the Decision Tree as the final classification stage, feeding it with all features described in Sections 2.4.1 and 2.4.4. A basic example of a decision tree is illustrated in Fig. 2.6. Inference in decision trees works by exploring the tree, starting from the root node until one of the leaf nodes is reached, which point to the most probable activity class. During the tree traversal for classification each node compares the value of its associated feature to decide on which branch to take next.

The specific algorithm we used to create the tree is based on the continuous C4.5 algorithm [146], resulting in a single tree that takes into account all the features evaluated by the MSP430 and by the PULP SoC. The C4.5 algorithms create a decision tree that is iteratively composed of nodes with four attributes: feature \( f \), threshold \( T \), and two children nodes. The C4.5 algorithm tries to split the dataset into two subsets with as much information content as possible, i.e., with the activity classes as uniform as possible in each subset; the measure of this uniformity is entropy in the sense of information theory.

Building a tree follows a “Divide and Conquer” method: Starting at the root node, the aim is to split the data into smaller subsets such that the classes in these subsets are more homogeneous than in the initial set. This is done recursively as long as there are significant enough data sets available, or no improvement in purity is possible with any feature.

2.4.6 C4.5 Decision Tree Algorithm

First, for the sake of simplicity, the discrete version is explained, and in a second step, the needed changes for continuous values are explained. The algorithm is based on the entropy metric:

\[
H = - \sum_i p_i \cdot \log_2(p_i) \tag{2.13}
\]

Entropy is a measure for the information content of a symbol or random variable in information theory. Equation 2.13 shows the according
2.4. CONTEXT CLASSIFICATION

![Decision Tree Example](image)

Figure 2.6: Decision Tree Example

formula where $p_i$ is the distribution of the $i$-th symbol. In a first step, the feature with the highest information gain (based on the splits for this feature) is calculated. The information gain is the benefit of entropy when splitting on the selected feature $k$. The entropies of all splits are evaluated, basically for each possible value of the feature. They are added up while they are weighted with the probability of the feature taking this value. Equation 2.14 shows the corresponding formula, where $a$ are the possible values of the evaluated feature $k$ and $H_a$ is the entropy in the split where the $k$-th feature takes the value $a$ and finally $p_a$ is the probability that the feature takes the corresponding value.

$$\text{InfGain} = H_{\text{root}} - \sum_a p_a \cdot H_a \quad (2.14)$$

**Extension for Continuous Features**

For continuous features, Ross proposes to find a threshold voltage which separates the data in two splits, where the information gain is optimized [146]. This is done for each feature. In the first step, the data are sorted. This is known to be possible in linearithmic time ($O(n \log n)$). Then the threshold is found with an exhaustive search [146]. This approach showed to be quite inefficient if the values are quite diverse. In this work, the inefficiency issue was solved by a binary search approach. The sorted data are split into two subsets, and the information gain of the median element of both splits are
evaluated. The subset with the higher information gain is chosen and again split into two subsets and so on until $n$ elements are left. Because only one threshold is searched for, a tree with only continuous data becomes a binary tree such that each node has two children/subtrees or is a leaf.

Considerations for overfitting

A general problem in decision trees is overfitting. Especially for continuous data, the algorithm splits the data until there is only one class left. In the worst case, this happens when there is only one sample left. If this happens, it is very likely that this sample is an outlier or a too precise case, which leads to overfitting and also tend to generate huge trees. An often-used method to overcome this problem is pruning. With pruning, the tree is fully developed at first. Then, in a bottom-up approach, leaves which only have few samples are combined to new leaves, which indicate the class, which is most probable in the merged set of samples. In this work, another approach was chosen. If the amount of data is too low based on a threshold, the subtree defines itself as a leaf, even if the tree is not fully extended yet.

Final continuous algorithm

Algorithm 1 shows the pseudocode of the implemented C4.5 algorithm. In line 2-3 the stop criterion is implemented, such that a leaf is generated if all class labels are equal or the sample size is lower than a desired threshold. In line 4 the current entropy of the root node is calculated. Then, in lines 5 and 6 every feature is evaluated and the information gain is calculated. The binary search will happen in line 6. In line 7 the best feature in terms of highest information gain is selected. Finally two new trees are generated in line 8 and processed by the C4.5 algorithm in a recursive way.

We used leave-one-out cross-validation [147] for evaluation. Thus, for each collected sequence of activity, a decision tree was trained based on the full set excluding the test sequence; the results we present in the following Section 2.5 are averaged over all test sequences. We have selected this extreme variant of cross-validation for optimal exploitation
Algorithm 1 C4.5 Algorithm (Continuous)

Require: Training Data $x_i \in X^k$, with $i \in \{1, 2, ..., m\}$
Require: Labels $l_i \in C$, with $i \in \{1, 2, ..., m\}$
Ensure: Decision Tree $T$

1: if $l_i = l_j \forall i, j \in N \leq m$ or $m < M_{\text{min}}$ or $\text{depth} > \text{depth}_{\text{max}}$ then
2: return leaf of class $c$, where $\arg \max_c p_c$
4: end if
5: Calculate Entropy of the root node

$$H_{\text{root}} \leftarrow \sum_{c \in C} (-p_c \cdot \log_2 p_c)$$

6: for all feature $k$ do
7: Find feature $F_k$ and threshold $T_k$ which maximize the information gain:

$$(F_k, T_k) \leftarrow \arg \max_{(F_k, T_k)} \left( H_{\text{root}} - p(x_k \leq T) H(x_k \leq T) - p(x_k > T) H(x_k > T) \right)$$

8: end for
9: Find feature with highest information gain:

$$F_{\text{sel}} \leftarrow \arg \max_{F_k} IG_k$$

10: Create two subtrees $T_1$ and $T_2$:

$$T_1 \leftarrow \text{C45\_MAKE\_TREE}(\{(x_n, l_n) | x_n \in x, x_n^k \leq T\})$$
$$T_2 \leftarrow \text{C45\_MAKE\_TREE}(\{(x_n, l_n) | x_n \in x, x_n^k > T\})$$
of the small training set while avoiding any contamination of the test set by samples from the corresponding training set. It should be mentioned here, that training time grows at least quadratically with the number of samples, but thanks to the fast convergence time, the training time was below 48 hours.

2.5 Results

In this section, we evaluate the accelerated smartwatch platform in terms of power and execution time, as well as in the accuracy of the context classification task. As a term of comparison, we use the non-accelerated smartwatch [89]. MSP430 code was compiled using the \texttt{ti-cgt-msp430 4.4.6} toolchain, while for PULP we used a custom Or10n toolchain, based on GCC 5.2. We estimated power consumption for PULP using backannotated switching activities from three input vectors in power analysis: \textit{idle, matmul} (which simulates a case where the cores are all running, with a low pressure on the shared memory) and \textit{dma} (which simulates a case where the DMA is running, with high pressure on memories). Then, we run our tests on an FPGA-based emulation platform for PULP [148], collecting active and idle cycles for cores, DMAs and interconnects. We model leakage power, dynamic power density, and maximum clock frequency at each operating point after the post-layout backannotated timing and power analysis results for the latest PULP chip. For this purpose, we considered the $V_{DD} = 0.5$ V operating point, which shows the best energy efficiency according to Figure 2.3. In this operating point, $f_{clk}$ is 50 MHz. The power consumption for the MSP430 and the peripherals were measured during idle and active mode where the microcontroller was supplied by 2 V and was operating at 8 MHz.

2.5.1 Context classification

To compare the non-accelerated platform with the proposed PULP-accelerated platform, we considered a set of combinations of several feature extractors, fused inside the decision tree, as explained in Section 2.4.5. In particular, we consider the following set of features: temp, cam, mic(no fft), mic, accel and their combinations indicate
2.5. RESULTS

tests using the features described in Section 2.4.1, which work without using the accelerator in the same way as in the baseline implementation [89]. \textit{mic(no fft)} does not include features based on the frequency domain representation of the audio signal, while \textit{mic} includes all audio features. \textit{all(no fft)} and \textit{all} indicate that all the features described in Section 2.4.1 (\textit{temp+cam+mic+accel}) are used (without or with FFT-based features, respectively); in case of the non-accelerated platform [89], all of them are executed on the MSP430, whereas in the accelerated platform we execute the extraction of features from the camera on PULP and that of the other features on the MSP430. \textit{cnn} is a test running on the accelerated platform where the classifier is the small CNN described in Section 2.4.4; in this case the Decision Tree is not used. \textit{all+cnn}, finally, considers the case in which we use the accelerated smartwatch with all non-visual features of Section 2.4.1 extracted on the MSP430, while we also integrate the output of the small CNN of Section 2.4.4 into the Decision Tree. We directly use the output of the softmax layer, which returns a certainty for all classes (i.e., values between 1 and 0. 1 in case the classifier is confident that it is for sure this class or 0 if it is clearly not this class). This allows the decision tree to learn also inter-class correlation.

Figures 2.7 focus on a preliminary analysis of our baseline, i.e., the non-accelerated platform [89]. We show the timing of using the accelerometer, microphone, and camera sensors. Further, we divide it in \textit{acquisition} time/energy, that is needed for the acquisition of data from the sensors, and \textit{feature extraction} time/energy; the thermistor is
left out of this analysis as it is orders of magnitude less expensive than the other sensors in both energy and time. Moreover, since we want to understand whether the acquisition or feature extraction time is dominant for a given sensor (and hence if it makes sense to accelerate elaboration with PULP), we do not consider using double buffering. I.e., we do not consider an overlapping acquisition of sensor data with computation. The accelerometer and the microphone need a long time to acquire data (on the order of 1s), while in the non-accelerated platform the camera is more than 20× faster, taking only 61 μs to acquire data. Similar time/energy is spent in the non-accelerated platform to extract audio and camera features, but while for the former it is possible to extract relatively complex frequency-domain features, for the latter the same energy is spent to extract very simple average-based features.

The Figures also report energy/time in the proposed accelerated platform when using the simple CNN of Section 2.4.4; the external ADC connected to PULP is also more efficient than the internal MSP430 ADC, providing a significant efficiency improvement to the platform. Overall, feature classification energy is reduced by using the PULP accelerator even if the feature extractor is much more complex, as more thoroughly exposed in the following.

Figure 2.9 plots accuracy versus energy per classification for both the platforms being compared. The blue dots in the plot refer to the non-accelerated case [89], where all computation is performed
2.5. RESULTS

Figure 2.9: Context recognition accuracy vs energy spent per acquisition+classification. Note: all+cnn includes all non-visionary features on the MSP430 and the CNN for the camera image.

Figure 2.10: Context recognition accuracy vs peak system power.
by the MSP430, while the red ones refer to the PULP-accelerated one. Each dot is tagged with the set active sensors and with the total classification accuracy obtained, and the dashed line highlights the Pareto-dominant points for the non-accelerated platform [89] in the accuracy-energy tradeoff. As could be expected, a clear tradeoff between accuracy and energy is shown here; it is necessary to spend more energy to obtain a better result in terms of accuracy. It is interesting to observe that of the four points where the camera is used in the non-accelerated platform, two (mic+cam+temp, all) are Pareto-dominant, clearly indicating that even with the very simple features that can be run on the MSP430 the camera achieves a good level of separation over the five classes considered (morning preparation, walking outdoors, public transportation, in the car, in the office); in particular, the fact that the results exceed those obtained with the accelerometer alone confirms that sensor data from the camera can be significant for the context recognition task. The two PULP-accelerated points are both abundantly Pareto-dominant in terms of accuracy per Joule, yielding up to 84% accuracy when using all features non-vision features on the MSP430 and the CNN on the PULP platform (all+cnn case) while at the same time saving more than 400 µJ per classification with respect to the best non-accelerated point (all).

The pure cnn case achieves a 64% accuracy comparable to that available when using the audio features in the non-accelerated platform, but at an energy budget per classification that is 25× lower (∼91 µJ).

The two all and all+cnn points are relatively close in terms of accuracy; adding the CNN, we are able to get an additional 3% of average accuracy on the five classes. Although the difference in terms of average accuracy is small, a closer look at the confusion matrices shows that the all+cnn case is actually a significant improvement over the all one. Figure 2.11 shows that in the all case there are two sources of inaccuracy: confusion between in the car and public transportation, and confusion between walking outdoors and in the office. As a consequence, only the accuracies of morning preparation and walking outdoors are above 90%. The all+cnn eliminates the second of these two inaccuracies, bringing the precision of in the office above 90%. The confusion between in the car and public transportation also stays in the all+cnn case; however, in our opinion, this can be
2.5. RESULTS

![Confusion Matrix](image)

Figure 2.11: Confusion matrices for all and all+cnn tests.

justified by the objective similarity of the two situations (sitting in a bus versus sitting in a car).

Figure 2.10 expands our analysis with the tradeoff between accuracy and peak power, an important metric for wearable systems as their small batteries are typically limited not only in terms of total energy capacity but also in sustainable power output. Accelerometer and thermistor contribute relatively little to the total system power consumption; the main dominant costs are, therefore, the compute units (MSP430 and PULP), the camera, and the microphone. The first interesting point to raise is that even when all sensors and compute units are kept on, total system power peaks at $\approx 9\text{mW}$, and that the addition of the PULP accelerator increases this peak power by less than 15% with respect to the peak power consumption of the baseline platform [89]. Moreover, by comparing Figures 2.9 and 2.10, it is easy to observe that even if the peak power consumption in the accelerated platform may be slightly higher, the overall energy consumption (and thus average power) is considerably lower, which means that if the platform is able to provide $\sim 10\text{mW}$ of peak power, the accelerated platform is convenient in terms of both energy and average power.
2.5.2 Battery Lifetime Estimation

As mentioned in Section 2.3, the system is supplied with two harvester sources (TEGs and solar cells). On average, these sources are able to provide $\sim 41 \mu W$, while the system power in deep sleep mode (with the MSP430 in LPM4 mode and PULP and peripherals power-gated) is $38 \mu W$. Assuming that the platform mounts a small lithium-ion polymer 4 V 150 mAh battery, in Table 2.2 we estimate the expected lifetime, knowing the energy per acquisition from Section 2.5.1 ($2.6 \text{mJ}$ for all, $2.2 \text{mJ}$ for all+cnn).

<table>
<thead>
<tr>
<th>Harvesting</th>
<th>all</th>
<th>all+cnn</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle (LPM4.5)</td>
<td>No</td>
<td>661d</td>
</tr>
<tr>
<td>always on</td>
<td>No</td>
<td>9d</td>
</tr>
<tr>
<td>every minute</td>
<td>No</td>
<td>307d</td>
</tr>
<tr>
<td>once a day</td>
<td>No</td>
<td>660d</td>
</tr>
<tr>
<td>always on</td>
<td>Yes</td>
<td>9d</td>
</tr>
<tr>
<td>every minute</td>
<td>Yes</td>
<td>617d</td>
</tr>
<tr>
<td>every 14m</td>
<td>Yes</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Table 2.2. Lifetime evaluation

Apart from the benefit in accuracy, the accelerated platform is also beneficial in terms of battery lifetime. This benefit steadily grows as we increase the interval between consecutive acquisitions.

At the limit, if that interval is brought to $\sim 14 \text{min}$ or more, the device is completely autonomous when using harvesting.

2.6 Conclusions

We have proposed an ultra-low-power smartwatch with multiple low-power sensors designed for recognizing context during our normal daily life, based on a very light-weight feature extractor based on decision trees.

Using the PULP programmable accelerator, our results show that we enable the implementation of vision algorithms of a significant level of complexity while keeping the overall system power budget below 10 mW at peak. This allows us to deploy these vision techniques to be
embodied directly in low power wearable devices such as smartwatches, glasses, and necklaces. Our results have shown that leveraging a speedup as high as $500\times$ on the computation of visual features, the heterogeneous platform we propose can achieve the same accuracy as our baseline [149] with a more than $25\times$ reduction in energy cost. A significant accuracy improvement was achieved with 84% average correctness at 2.2 mJ per classification. This opens up new possibilities in terms of implementation of ego-vision functionalities directly on low power wearable devices, allowing for significant savings in energy used for communication to external, higher-performance computing devices.
Chapter 3

Embedded BNN
Enabling Sound Event Detection

Decision trees, as used in the previous chapter, are indeed very lightweight due to their logarithmic time and memory complexity, and their inference has proven to be very energy-efficient. But their classification performance is limited due to the long training time, the need for a reasonable selection of hand-crafted input features, and the tendency for overfitting. In this chapter, we use a different approach and do the entire classification with a neural network for the Sound Event Detection application. Even though existing algorithms are already based on neural networks, they cannot be implemented on current IoT systems due to the high demands in terms of memory, power, and throughput. It turns out that the network indeed cannot be fit into a standard microcontroller, as 6.4 MByte would be required for parameters and a minimal set of intermediate feature maps. To overcome this issue, we evaluate and implement binary neural networks due to its extreme reduction in computational and memory requirements. We retrain an existing network for SED while constraining it to binary weights and activations. We reach a
classification accuracy of 77.9% on 28 classes, 7% point worse than the full-precision base-line. Furthermore, we implement the entire MFCC feature extraction and efficient software implementation on the low-power GAPuino platform. Our implementation reaches an energy efficiency of 34.5 GMAC/s/W. We compare the performance with an ARM Cortex-M based implementation, showing that the model running on an ultra-low-power-platforms requires 14.7x less execution time with 57x more energy efficiency.

3.1 Introduction

Edge computing targets the power issues and privacy concerns in AI on IoT end nodes, moving the information extraction directly on the node, and transmitting only significant information to the cloud. As already discussed in the previous chapter, microcontrollers with consumption in the range of mW are available, enabling AI directly on board, without the need to access the cloud. These concepts fit perfectly in the field of Sound Event Detection (SED): here are much pervasive application related to IoT and Smart City, such as traffic monitoring [150], crowd monitoring [151], measurement of occupancy level for efficiency in buildings [152], detection of emergencies [153]. While analyzing data locally, the system get benefits from a privacy perspective, it reduces the latency to the order of ms, and the energy required for transmission of the relevant information enables hardware solutions based on energy harvester with a lifetime of several years. However, this vision implies data processing on the sensor node. Unfortunately, high accuracy classification algorithm for Sound Event Detection (SED) are also very computationally intense and resource-demanding, both in terms of memory and power consumption. Over the last few years, many researchers put efforts towards specialized hardware and optimized inference algorithms to run such Neural Network (NN) on power-constrained devices. On the software side, network complexity reduction while preserving the quality of predictions is of significant interest in porting deep and complex architectures on a heavily constrained IoT node. There are several approaches to target this goal, e.g., knowledge distillation [154], network pruning [155] or network quantization [156]. In this work, we implement an extreme
3.1. INTRODUCTION

Quantization for neural networks, in which every weight and activation is described by a single bit so that we can assume a value of -1 or 1, and is introduced in Section 3.3. On the hardware side, IoT implementations are often based on Cortex-M cores, thanks to their consumption in the range of mW and their throughput in the order of MOp/s. However, very few implementations of NN on microcontrollers are presented in literature [157,158], because standard microcontrollers do not solve the dual requirement of low power consumption and fast processing. In this chapter, we are using the GAP8 platform, which is a commercial product originating from the PULP project. Differently to the PULPv3 processor used in the previous chapter, GAP8 is based on the RISCY core — a DSP-extended RISC-V ISA processor, which will be introduced more in detail in Section 4.2.3.

The GAP8 platform is used in this work, which is a commercial product based on the RISCY processor core from the PULP project, and includes all these features. Differently to the OpenRISC based PULPv3 core used in Chapter 2, RISCY is a DSP extended RISC-V processor, and will be introduced more in detail in Section 4.2.3. Besides, it has useful built-in instruction for \texttt{popcount}$^1$, post-increment load and hardware loops, that boost the processing significantly for BNN. The contributions of this work are:

1. We propose, train, and efficiently implement a novel BNN architecture for SED, comparing it with a full-precision baseline network.

2. We present the design of a full system, based on the low-power and ISA optimized for GAP8 microcontroller. The full pipeline is developed from audio acquisition with a low-power microphone, over the Mel bins feature extraction to the on-board classification. We present a detailed analysis of throughput and energy trade-off in a variety of supported configurations as well as on-board measurements.

3. We demonstrate that binarization of weights and activations are the key factor in matching hardware constraints. Experimental

\footnote{The popcount function/instruction returns the number of ‘1’ in an integer value, i.e., \texttt{popcount}(4'b1011) = 3.}
evaluation shows that our implementation on the Parallel Ultra Low Power (PULP) platform is 51x more efficient and 10x faster than the implementation of the same network in the Cortex-M4 based counterpart.

3.2 Related Works

Historically, SED was addressed with Mel Frequency Cepstral Coefficients (MFCC) features and GMM, HMM or SVM classifier [159–161]. Recently, DNN [162], CNN [163] or RNN [164] have been used. However, these high performance models require a lot of memory to perform predictions: embedding extractor for sound event detection such as L3 [165] or VGGish [163] require approximately 4M and 70M parameters, respectively. In literature, there are works, targeting the IoT application scenario, in which the authors reduce the structure size of an existing network for SED. Employing knowledge distillation, the L3 network is compressed to edge-L3 [165], and VGGish is further compressed to baby VGGish [164].

By replacing the fully connected layer of an existing CNN with average max-pooling, Meyer et al. [36] reduced the number of parameters while increasing the accuracy for the targeted dataset. Still, Meyernet is not suitable for our very constrained IoT use-case. Therefore further model compression is required to match these constraints.

In addition to model structure modification, recent works on CNN have investigated quantization to reduce the storage and computational costs of the inference task [37,53,156]. As an extreme case of quantization, BNNs reduce the precision of both weights and neuron activations to a single-bit [79,166]. BNNs work on simple tasks like MNIST, CIFAR-10, and SVHN without drop in accuracy [167]. On the challenging ImageNet dataset, BNNs/TNNs have a drop of 12%/6.5% [168,169]. Recent approaches use multiple binary weight bases, or part of the convolutions are done in full-precision. An accuracy drop down to 3.2% has been achieved [170]; unfortunately, these approaches increase the weight memory footprint and computational complexity.
3.3. FEATURE EXTRACTION AND BNN

BNNs are suitable to be implemented on resource-constrained platforms, thanks to their reduced memory requirements and their potential to convert multiplications in hardware-friendly XNOR operations.

Several works have implemented CNNs with fixed-point format and operations, in video domain [171] and in audio domain, where keyword spotting in Cortex-M4 based microcontroller [158], Cortex-M0+, and Raspberry Pi based platforms [157].

One of the challenges in this field is the development of energy-efficient NN firmware implementation for embedded systems. Wang et al. [172] developed a library for neural network porting from the FANN framework to ARM MicroController Units (MCUs) and PULP platforms. In this case, the hardware is fully utilized, but there is support only for multilayer perceptrons. Garofalo et al. developed a custom library for quantized convolutional neural networks on PULP [173]. Unfortunately, they do not support efficient BNN mapping. Thus we implemented our custom functions. To the best of our knowledge, this is the first BNN proposed and implemented on a parallel RISC-V based microcontroller.

3.3 Feature Extraction and BNN

State-of-the-art solutions for SED are mostly based on CNNs fed with mel-spectrogram [36, 174–176] of the sequential audio data. First, we are introducing the MFCC feature extraction, followed by a short introduction in binary CNN, its software implementation, and the neural network topology.

3.3.1 Spectrogram-based CNN and MFCC

Sequential data can also be represented in a spectrogram, where as a sliding window of sequential samples are mapped to the frequency domain using FFT. The resulting time-to-frequency ‘image’ is than fed to a conventional convolutional network. This approach has been used for speech recognition based on MFCC spectrogram [177]. MFCCs base on the Mel frequency, which is a logarithmic frequency space
which should be more similar to the human perception of sound. Thus the interval of two musical pitches (e.g., an octave) is perceived linearly, even though it is exponential in the frequency space (e.g., doubling or halving in case of an octave). Typically, the following steps are required to create the MFCC spectrum from a raw audio signal:

1. Framing and Windowing: The signal is split into small overlapping tiles on which the Fourier Transform is applied to. This tiles have to be small enough to catch relevant details in the audio signal. Optionally, the signal is multiplied with a windowing function (e.g., Hamming) to avoid edge effects.

2. Fourier Transform: Typically, the Short Time Fourier Transform (STFT) is used, which can be efficiently be implemented on microcontroller by the Cooley-Turkey algorithm [178].

3. Logarithmic Filter Banks: The mapping from the frequency space and mel space is done through applying triangular-shaped filters, where as the filter sizes increase exponential to fulfill the frequency-to-mel mapping: 

$$f_{[\text{mel}]} = 1127 \cdot \ln\left(\frac{f_{[\text{Hz}]}}{700} + 1\right)$$

4. Logarithm of the resulting coefficients to consider the exponential property of the loudness.

5. Optionally, the Discrete Cosine Transform (DCT) is used for dimension reduction:

$$X_n = \sum_{k=0}^{K-1} x_k \cos\left(\frac{n\pi(2k + 1)}{2K}\right)$$

with $K$ input coefficients, and the output coefficients $X_n$.

The preprocessing part computes the STFT in windows of 32 ms every 8 ms. Then we use the mel-filter and discrete cosine transform to generate 64 Mel-frequency Cepstral Coefficient. 400 features are then tiled together to create the mel-spectrogram for 3.2 s of audio. The matrix, with shape $64 \times 400$, is the input to the neural network.
3.3. **FEATURE EXTRACTION AND BNN**

3.3.2 **First Layer and Binarization**

The input data to the network is non-binary and has, therefore, to be treated separately. A robust approach is to keep the first network layer in full-precision, like in Courbariaux et al. [166]. In this way, the network learns the binarization function from the training set.

After the convolution we apply batch normalization, following this formula

\[
y_c = \frac{x_c - \mu_c}{\sqrt{\sigma^2_c + \epsilon}} \gamma_c + \beta_c
\]  

(3.1)

where \( \mu_c \) and \( \sigma^2_c \) are mean and variance of the input for the specific output channel \( c \). During training the framework compute the running average of these parameters using the training set. \( \gamma_c \) and \( \beta_c \) are learned using back propagation. To reduce the number of operation, we can define two parameters:

\[
\gamma' = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \quad \beta' = \beta - \frac{\gamma \mu}{\sqrt{\sigma^2 + \epsilon}}
\]  

(3.2)

and (3.1) becomes

\[
y = x \gamma' + \beta'.
\]  

(3.3)

Finally, we binarize the equation using the signum function:

\[
\text{sgn}(y) = \begin{cases} 
-1, & \text{if } y < 0 \\
1, & \text{if } y \geq 0
\end{cases}
\]  

(3.4)

To avoid floating-point operations, all the operations described in this subsection are done in fixed-point. It turns out to be more efficient in terms of execution time and energy consumption without significant loss of performance [156]. We confirm this hypothesis in the result session.

On the other hand, fixed-point quantization requires additional effort in finding the correct amount of integer and fractional bits for each parameter representation. For doing this, we check the range of the parameters, and we choose the number of integer decimals that represents without saturation most of the numbers (99.9%)
3.3.3 Binary Neural Networks BNNs

BNNs are a subset of Neural Networks, whereas the intermediate feature maps and the weights are quantized to a single bit, and thus \( \mathbf{I} \in \{-1, 1\}^{n_{in} \times h \times b} \) and \( \mathbf{W} \in \{-1, 1\}^{n_{out} \times n_{in} \times k_y \times k_x} \). While calculating the output feature maps, the full resolution is preserved and is re-binarized after all input channel contributions have been summed together. Typically, the signum function is used as the activation function (i.e., \( \text{sgn}(x) = (-1)^{1_{x>0}} \)) for re-binarization. Training of BNNs is not trivial, as gradients are not smooth anymore due to the high non-linearity of the parameter space. The most common approach bases on shadow weights in high precision (e.g., FP32). These weights are binarized during the forward-propagation. During back-propagation, the gradients are applied to the shadow weights. Even though the binarization itself is not derivable, it can be modeled as the identity function. This can be interpreted as propagating the stochastic expected value of the gradient to the weights (i.e., straight-through estimator) [179]. The \( k \)-th output feature map \( o_k \) is the sum of convolutions of every binarized input feature map \( \hat{i}_n \) with the corresponding binary weights \( \hat{w}_{k,n} \) and the bias \( C_k \): \[
   o_k = \text{sgn} \left( C_k + \alpha \sum_{n \in I} \text{sgn}(i_n) \ast \text{sgn}(w_{k,n}) \right) \quad (3.5)
\]

Tab. 3.1 gives an overview of the performance of state-of-the-art BNNs and TNNs\(^2\) on the challenging ImageNet Large Scale Visual Recognition Challenge\(^3\) compared with the full-precision baseline networks. Recent research has been focusing mainly on minimizing the quantization error, improving the loss function, and reducing the gradient error [184]. XNORnet extends the stochastic gradient descent algorithm (commonly used to train NNs) by quantizing the weights and activations in the forward path and scales the feature maps \( \ell^1 \) matrix norm of the weight kernels [79]. On ImageNet, they

\(^2\)Ternary Neural Networks have ternary (i.e., \{-1,0,1\}) weights and activations, where

\(^3\)ImageNet is composed of more than 1 million images of 1000 different object classes (e.g., dalmatian, border collie, judo, ...)
Table 3.1. Overview of SoA BNN and TNN compared to their full-precision baseline networks.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Network Model</th>
<th>Quantization Baseline Acc.</th>
<th>Baseline Acc.</th>
<th>BNN Accuracy</th>
<th>BNN Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wght. Act.</td>
<td>Top-1</td>
<td>Top-5</td>
<td>Top-1</td>
<td>Top-5</td>
</tr>
<tr>
<td>Spallanzani19 [169]</td>
<td>MobileNet-V2 2</td>
<td>71.3</td>
<td>64.8</td>
<td>-6.5</td>
<td></td>
</tr>
<tr>
<td>Spallanzani19 [169]</td>
<td>AlexNet 2</td>
<td>55.9</td>
<td>45.8</td>
<td>-10.1</td>
<td></td>
</tr>
<tr>
<td>Zhou16 [168]</td>
<td>AlexNet 1</td>
<td>55.9</td>
<td>43.6</td>
<td>-12.3</td>
<td></td>
</tr>
<tr>
<td>Phan20 [180]</td>
<td>MobileNet 1</td>
<td>70.9</td>
<td>54.4</td>
<td>-16.5</td>
<td>-12.4</td>
</tr>
<tr>
<td>Rastegari16 [79]</td>
<td>ResNet-18 1</td>
<td>69.3</td>
<td>51.2</td>
<td>-18.1</td>
<td>-16.0</td>
</tr>
<tr>
<td>Hubara16 [167]</td>
<td>AlexNet 1</td>
<td>55.9</td>
<td>36.1</td>
<td>-19.8</td>
<td></td>
</tr>
<tr>
<td>Lin17 [181]</td>
<td>ResNet-34 1</td>
<td>73.3</td>
<td>52.4</td>
<td>-20.9</td>
<td>-14.8</td>
</tr>
<tr>
<td>Lin17 [181]</td>
<td>ResNet-18 1</td>
<td>69.3</td>
<td>42.7</td>
<td>-26.6</td>
<td>-21.6</td>
</tr>
</tbody>
</table>

**Non-Standard Binary Approaches**

<table>
<thead>
<tr>
<th>Paper</th>
<th>Network Model</th>
<th>Quantization Baseline Acc.</th>
<th>Baseline Acc.</th>
<th>BNN Accuracy</th>
<th>BNN Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wght. Act.</td>
<td>Top-1</td>
<td>Top-5</td>
<td>Top-1</td>
<td>Top-5</td>
</tr>
<tr>
<td>Zhuang19 [170]</td>
<td>ResNet-50 8x1(^a)</td>
<td>76.0</td>
<td>92.9</td>
<td>72.8</td>
<td>90.5</td>
</tr>
<tr>
<td>Berthge19 [182]</td>
<td>ResNetE18(^b) 1(^c)</td>
<td>58.1</td>
<td>80.6</td>
<td>54.4</td>
<td>77.8</td>
</tr>
<tr>
<td>Lin17 [181]</td>
<td>ResNet-34 5x1(^a)</td>
<td>73.3</td>
<td>91.3</td>
<td>65.0</td>
<td>85.9</td>
</tr>
<tr>
<td>Mishra17 [183]</td>
<td>ResNet-34 2x2x1(^d) 2x1</td>
<td>73.6</td>
<td>69.9</td>
<td>-3.7</td>
<td></td>
</tr>
<tr>
<td>Mishra17 [183]</td>
<td>ResNet-34 3x3x1(^d) 3x1</td>
<td>73.6</td>
<td>72.4</td>
<td>-1.2</td>
<td></td>
</tr>
</tbody>
</table>

\(^a\)BNN with multiple of binary weight bases. E.g., One binary neural network is duplicated in 8 parallel layers.
\(^b\)ResNet-18 with bypasses from every layer to all subsequent layers
\(^c\)1x1 convolutions on bypasses in full-precision
\(^d\)Number of channels are scaled up by 2 or 3×
achieved 51.2% using a binarized ResNet-18, which was a significant drop of -18.1%. Courbariaux et al. achieved then state-of-the-art results with 99.04 on MNIST (+0.34%), 97.47 on SVHN (-0.09%), and 89.85% Cifar-10 (-0.46%), but these tasks are much simpler than ImageNet [167]. Recently, the accuracy gap between BNN and their full-precision equivalent have been brought down to 12% (DoReFaNet on Alexnet [168]) and MoBiNet reached a Top-1 Accuracy of 54.4 (-16.5%) and Top-5 of 77.5 (-12.4%) [180]. To further close the gap, the most promising approaches are increasing the number of feature maps or use several binary layers (i.e., weight bases) in parallel to replace the full-precision layers. While the accuracy gap has been reduced significantly, this approach also leads to a linear or quadratic increase of computational complexity and memory footprint. Lin et al. achieved 69.3%/89.2% (Top-1/Top-5, -8.3%/-6.0% vs. ResNet-18) using 3 bases [181], and Zhuang et al. 72.8%/90.5 (Top-1/Top-5, -3.2%/-2.4% vs. ResNet-50) using 8 bases [170].

3.3.4 BNN Implementation

To avoid using two bits, we represent $-1$ with 0, whereas the actual binary numbers are indicated with a hat (i.e., $\hat{i} = (i + 1)/2$). It turns out that multiplications become xnor operations $\bar{\oplus}$ [79]. Formally the output $o_k$ of an output channel $k \in \{0, ..., n_{out} - 1\}$ can be described as$^4$:

\[
\begin{align*}
o_k &= sgn \left( \sum_{n=0}^{n_{in} - 1} i_n \ast w_{k,n} \right) = sgn \left( \sum_{n=0}^{n_{in} - 1} 2 \left( \hat{i}_n \ast \hat{w}_{k,n} \right) - k_y k_x \right) \\
&= sgn \left( \sum_{n=0}^{n_{in} - 1} \sum_{(\Delta x, \Delta y)} 2 \left( \hat{i}_n^{\Delta y, \Delta x} \oplus \hat{w}_{k,n}^{\Delta y, \Delta x} \right) - 1 \right)
\end{align*}
\]

Whereas $\Delta y$ and $\Delta x$ are the relative filter tap positions (e.g., $(\Delta y, \Delta x) \in \{-1, 0, 1\}^2$ for $3 \times 3$ filters). As calculating single-bit operations on

\footnote{For simplicity reasons we omit bias and scaling factor in the formula.}
microcontroller is not efficient, we pack several input channels into a 32-bit integer (e.g., the feature map pixels at \((y + \Delta y, x + \Delta x)\) in spatial dimension and input channels 32\(n\) to \((32(n + 1) − 1)\) packed in \(\hat{i}_{32n+32}^{y+\Delta y,x+\Delta x}\)), while the Multiply Accumulates (MACs) can be implemented with \textit{popcount} and \textit{xnor} operations. Furthermore, as common embedded platforms like GAP8 do not have a \textit{xnor} instruction, the result is inverted and \textit{xor} is used:

\[
= \text{sgn} \left( \sum_{n=0}^{\frac{n+1}{32} - 1} \sum_{(\Delta x, \Delta y)} 2\text{popcnt} \left( \hat{i}_{32n+32}^{y+\Delta y,x+\Delta x} \oplus \hat{w}_{k,32n+32}^{\Delta y,\Delta x} \right) - 32 \right) \tag{3.8}
\]

Furthermore, as common embedded platforms like GAP8 do not have a built-in \textit{xnor} operator, the \textit{xor} operator \(\oplus\) is used and the result is inverted. Therefore the final equation for the output channel \(o_k\) is as follows.

\[
o_k = \text{sgn} \left( \sum_{n=0}^{\frac{n+1}{32} - 1} \sum_{(\Delta x, \Delta y)} 32 - 2\text{popcnt} \left( \hat{i}_{32n+32}^{y+\Delta y,x+\Delta x} \oplus \hat{w}_{32n+32}^{\Delta y,\Delta x} \right) \right) \tag{3.9}
\]

### 3.3.5 Batch Normalization and Binarization

A batch normalization layer follows each binary convolutional layer. As the output of binary layers are integer values, and the signum function can be written as a comparison function, the activation function is simplified to:

\[
\text{binAct}(x) = \begin{cases} 
0, & \text{if } x \cdot \text{sgn}(\gamma') \geq \left\lfloor \frac{\beta'}{\gamma'} \right\rfloor \\
1, & \text{if } x \cdot \text{sgn}(\gamma') < \left\lfloor \frac{\beta'}{\gamma'} \right\rfloor 
\end{cases} \tag{3.10}
\]

whereas \(\gamma'\) is the scaling factor and \(\beta'\) is the bias based on the batch normalization parameters. While exporting the model, we compute the integer threshold value \(\left\lfloor \frac{\beta'}{\gamma'} \right\rfloor\) in advance. In inference, one sign
Table 3.2. Kernel size, channel, and computational effort for each layer

<table>
<thead>
<tr>
<th>Layer</th>
<th>Kernel Size</th>
<th>Channel</th>
<th>Stride</th>
<th>MACs</th>
</tr>
</thead>
<tbody>
<tr>
<td>First (real-valued)</td>
<td>3 × 3</td>
<td>32</td>
<td>1</td>
<td>7M</td>
</tr>
<tr>
<td>1. Binary Layer</td>
<td>3 × 3</td>
<td>64</td>
<td>2</td>
<td>109M</td>
</tr>
<tr>
<td>2. Binary Layer</td>
<td>3 × 3</td>
<td>128</td>
<td>1</td>
<td>405M</td>
</tr>
<tr>
<td>3. Binary Layer</td>
<td>3 × 3</td>
<td>128</td>
<td>2</td>
<td>186M</td>
</tr>
<tr>
<td>4. Binary Layer</td>
<td>3 × 3</td>
<td>128</td>
<td>1</td>
<td>154M</td>
</tr>
<tr>
<td>5. Binary Layer</td>
<td>1 × 1</td>
<td>128</td>
<td>1</td>
<td>17M</td>
</tr>
<tr>
<td>Last (real-valued)</td>
<td>1 × 1</td>
<td>28</td>
<td>1</td>
<td>6M</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td></td>
<td></td>
<td></td>
<td>884M</td>
</tr>
</tbody>
</table>

comparison and one threshold comparison have to be calculated for each activation value.

### 3.3.6 Last Layer and Prediction

In the last layer, the fixed-point values from the last binary layer are convolved with the fixed-point weights, and N output channels are calculated, where N is the number of classes. Finally, the network performs an average pooling over the whole image giving N predictions for each class.

### 3.3.7 Neural Network Architecture

Tbl. 3.2 summarizes the architecture of the NN. The neural network consists of 7 hidden layers, 5 of which are binary. The first and last layers are real-valued. Their required computations are significantly smaller than in the binary layers (e.g., 7MMAC in the first layer compared to 109MMAC in the second layer), and therefore they minimally contribute to the overall computational effort. The reason for having real-valued layers is the high loss of accuracy with entirely binarized neural networks [79].
3.4 Embedded Implementation

The Mel bins extraction and BNN are implemented on GAP8. The application scenario for this device is low-latency low-power signal processing. The device has a tunable frequency and voltage supply. Fig. 3.4 shows the main block of the chip: GAP8 has two main programmable components, the fabric control, and the cluster. The fabric control is the central microcontroller unit, and it is meant to manage peripherals and offload workloads to the cluster. The cluster is composed of eight parallel RISC-V cores, a convolution accelerator, and shared memory banks. The two domains share the same voltage source but keep two different frequencies: On-chip DC-DC converters translate the voltage, and two independent frequency-locked loops (FLLs) generate the two different clock domains. The fabric controller is a single-core in-order microcontroller implementing the RISC-V instruction set. To customize the core for signal processing application, GAP8 extends the RISCV-IMC instruction set for signal processing.
CHAPTER 3. EMBEDDED BNN

application. In addition to integer, multiplication, and compressed instruction (IMC), GAP8 ISA supports Multiply and Accumulate, Single Instruction Multiple Data (SIMD), Bit manipulation, post-increment load/store, and Hardware Loops.

The fabric controller is directly interconnected to an L2 memory of 512 kB SRAM.

The cluster has eight cores identical to the fabric controller. The cores share the 64 kB L1 SRAM scratchpad memory, equipped with a logarithmic interconnect that supports single-cycle concurrent access from different cores requesting memory locations on separate banks.

The cores fetch instructions from a multi-ported instruction cache to maximize the energy efficiency on the data-parallel code. Moreover, an efficient DMA (called µDMA) enables multiple direct transfers from peripherals and L1 to the L2 memory. The cluster has a hardware synchronizer for event management and efficient parallel threads dispatching. The fabric controller and cluster communicate with each other by an AXI-64 bidirectional bus. The software running on the fabric controller overviews all tasks offloaded to the cluster and the µDMA. At the same time, a low-overhead runtime on the cluster cores exploits the hardware synchronizer to implement shared-memory parallelism in the fashion of OpenMP [148].

The overall prediction cannot run directly for the whole image because of L1 memory constraints, so we split the image into 4 tiles. The tiles have an overlap of 20 pixels to take into account the receptive field of convolutional kernels at the border of the tile. The firmware implements a double buffering for the weight loading: before the program processes the input of a specific layer, the cores configure the DMA to load the weights of the next layer, from the L2 memory to the single-cycle accessible L1 memory. An interesting feature of GAP8 is the built-in popcount instruction, which takes just one cycle and decreases the execution time significantly in binary layers. The single 3×3×C kernel application gains speed thanks to loop unrolling. Finally, the code parallelization over the 8 cores is implemented using the OpenMP API.
3.5 Experimental Results

To accurately evaluate the BNN approach, we designed a full system. Thus, the power and energy-efficient measurements are performed on the hardware platform.

3.5.1 Dataset

In this work, we use the dataset of Takahashi et al. [186], which is based on the Freesound database, an online collaborative sound database [187]. It consists of 28 different event types, e.g., instruments, animals, mechanical sounds. Each clip has a variable length, and the total length of all 5223 audio files is 768 minutes. All audio samples have a sampling rate of 16 kHz, a bit depth of 16, and are single-channel. The dataset is split into training (75%) and test set (25%). We compute the STFT in windows of 512 samples every 128 samples, respectively 32 ms and 8 ms. Then we apply 64 Mel-filters to generate 64 Mel bins. 400 features are then tiled together to create the Mel-spectrogram for 3.2 s of audio. For the training set, we split each audio clip in consecutive chunks of 3.2 s.

In the test set, we extract one single patch of 3.2 s, starting from half of the clip.

3.5.2 Accuracy

We start from MeyerNet [36] and use the Additive Noise Annealing (ANA) algorithm [169] to train the network with binary weights and activations. Tbl. 3.3 provides an overview of the original MeyerNet, the BNN and some different quantization schemes. Q8NN is the network quantized to 8-bits, BNN&FP with the first and last layer in FP32 and BNN&FXP with the non-binary layers in 16-bit fixed-point. For the accuracy in Q8NN, we consider the energy efficiency results from PULP-NN [173] and the accuracy is expected to be the same the FP32 baseline. The BNN has a accuracy gap of 7.4% point, which is in-line with the literature about BNNs [79].

5Neural Networks are robust to quantization down to 8 bits [53,171,188]
Table 3.3. Accuracy, Memory Footprint and Energy Cost for the baseline CNN (full-precision), BNN with first/last layer full-precision, BNN with first/last layer in 16-bit Fixed-Point on the GAPuino

<table>
<thead>
<tr>
<th></th>
<th>FP32-NN</th>
<th>Q8NN [36]</th>
<th>BNN&amp;FP</th>
<th>BNN&amp;FXP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>85.1%</td>
<td>85.1%</td>
<td>77.9%</td>
<td>77.9%</td>
</tr>
<tr>
<td>Energy [mJ]</td>
<td>-</td>
<td>36.8$^a$</td>
<td>1877</td>
<td>25.6</td>
</tr>
<tr>
<td>Memory [kB]</td>
<td>6380$^b$</td>
<td>1200$^b$</td>
<td>230</td>
<td>230</td>
</tr>
</tbody>
</table>

$^a$Based on the energy efficiency of PULP-NN on GAPuino, presented in Garofalo et al. [173] based on 8-bit quantized neural network, while exploiting the 8-bit SIMD instructions where as the convolution kernels are written as matrix-matrix multiplication while the preprocessing is not included in the number (i.e., im2col algorithm).

$^b$It does not fit into the 512 kB SRAM of the GAP8 microcontroller.

Tab. 3.3 presents the accuracy values for three different models. The CNN refers the full precision model described in [36]. Then we have the BNN with the first and last layer in full precision. The gain in energy efficiency and computational time of BNN costs 7.3 points in accuracy, but it is in line with literature about BNN (i.e., 12% binary and 6.5% ternary neural networks for ImageNet [168,169]).

Finally, we computed the prediction and feature extraction directly on the GAPuino board. The floating-point operations are here converted into fixed-point operation with a bit-width of 16. We observes that there is no relevant difference between the two models, respectively, the second and third column of Tab. 3.3.

### 3.5.3 Energy Efficiency

Tab. 3.3 gives also an overview of the energy consumption for a single classification sample. The BNN is obviously more efficient than the 8-bit quantized version because the \texttt{xnor} and \texttt{popcount} operations merge 32 MAC into two instructions. However, the memory requirement is the key difference between the two networks: all the weights should fit inside the L2 memory for energy efficiency and high throughput. The number of parameters is 401k, so the 8-bit quantized version requires
3.5. **EXPERIMENTAL RESULTS**

8 bit per parameter, while the network presented here requires 1 bit for most of the parameter and 16 bit for the first and last layer. Also, the audio input data has to saved inside the L2 memory, and for 3.2 seconds at 16-bit resolution and sampling rate of 16 kHz, it makes 102 kB of additional memory requirement, and the largest subsequent feature map volumes has a size of 1.2 M. Tab. 3.3 shows that only BNN matches with the memory constraints of 512 kB of L2 memory in GAP8 chip.

In the following, we present energy efficiency changing the frequency and voltage of cluster and fabric controller. Once we find the most efficient solution, we analyze the performance layer by layer using the best combination of frequency and voltage.

We tried many different combination of frequency: cluster frequency and fabric control frequency range in these two set of frequency respectively \{30,50,85,100,150\} and \{10,30,50,100,150\} for 1 V \{50,100,150,200,250\} and \{10,30,50,100,150\} for 1.2 V. Different frequency combinations have different throughput, here measured as frame per second. Each frame of audio lasts 3.2 s, so the real time constraints is 0.3125 frame per second. Fig. 3.2 shows clearly, that the 1.0 V corners pareto-dominate the faster 1.2 V corners. It can be seen that the most energy-efficient corner is at 100 MHz for the fabric controller, and 150 MHz for the cluster, where the system achieves an energy efficiency of 31.3 GMAC/s/W, and a throughput of 1.5 GMAC/s.

### 3.5.4 Execution Time and Power Consumption

We profile time and throughput as well as the energy-efficiency of each layer of the NN. The network architecture is shown in Tbl. 3.2 together with the amount of Multiply-ACcumulate (MAC) required for each layer at the most energy-efficient corner according to the analysis in the previous section (i.e., $V_{dd} = 1.0$ V, $(f_{cl}, f_{fc}) = (150$ MHz, 100 MHz)).

The measurements are performed with the Rocketlogger [189]. Voltage and current of the System-on-a-chip (SoC) are logged. We evaluate the power and duration of measurements and calculate the energy consumption. The results for each layer are listed in Tbl. 3.4.
Figure 3.2: Throughput and energy efficiency at different supply voltages and operating frequencies. All of the measured settings fulfill the requirement of one classification every 3.2s (see the grey dashed line).

Binary layers are the most efficient ones; this is because of the combination of xor and popcount instructions processing 32 pixels in just 2 instructions. The efficiency peak is at 67.1 GMAC/s/W in the fourth binary layer, and the average efficiency is 34.5 GMAC/s/W. The most efficient configuration meets the real-time constraint, and the entire network runs within 0.511 s.

For a further investigation of the improvement in throughput and energy efficiency thanks to the capabilities of the GAP8 SoC, we have implemented the BNN on the STM32F469I Discovery board. Fig. 3.3 gives an overview of the improvements of the GAP8 implementations compared to the single-core ARM Cortex-M4F implementation, which has popcount implemented in software. We port the SW-popcount (i.e., 12 cycles) to GAP8 and run the code on a single core, and all 8 cores. The GAP8 compared to the STM32F469I, running both the BNN on a single-core and without HW-popcount, shows a 7.9× better
3.5. EXPERIMENTAL RESULTS

Table 3.4. Duration and energy consumption for each layer as well as throughput and energy efficiency compared to MACs

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MFCC</td>
<td>-</td>
<td>77.0</td>
<td>2.64</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>First Layer</td>
<td>7M</td>
<td>130.8</td>
<td>5.94</td>
<td>54M</td>
<td>1.2G</td>
</tr>
<tr>
<td>1. Bin. Layer</td>
<td>109M</td>
<td>73.3</td>
<td>3.57</td>
<td>1494M</td>
<td>30.6G</td>
</tr>
<tr>
<td>2. Bin. Layer</td>
<td>404M</td>
<td>168.0</td>
<td>8.86</td>
<td>2404M</td>
<td>45.6G</td>
</tr>
<tr>
<td>3. Bin. Layer</td>
<td>185M</td>
<td>51.2</td>
<td>2.94</td>
<td>3628M</td>
<td>63.2G</td>
</tr>
<tr>
<td>4. Bin. Layer</td>
<td>154M</td>
<td>40.3</td>
<td>2.29</td>
<td>79M</td>
<td>67.1G</td>
</tr>
<tr>
<td>5./6. Layer</td>
<td>21M</td>
<td>47.4</td>
<td>1.93</td>
<td>1724M</td>
<td>1.9G</td>
</tr>
<tr>
<td>Σ</td>
<td>882M</td>
<td>588.0</td>
<td>28.18</td>
<td>1503M</td>
<td>31.3G</td>
</tr>
</tbody>
</table>

Figure 3.3: Improvement in throughput and energy efficiency compared to the ARM Cortex-M4 implementation. With the following use-cases: single-core (SC) implementation on GAP8 with software popcount, GAP8/SC with (HW) popcount instruction, GAP8/8-cores with SW or HW popcount.
energy efficiency, but with a 1.6× lower throughput due to the higher operating frequency of the ARM core. Enabling the HW-popcount gives a significant improvement in energy efficiency (2.8×) and speed in computation (4.3×). Running the BNN on all 8 cores gives an improvement of 6.9/2.4× in throughput and energy efficiency. Finally, the popcount ISA extension gives another boost of 2.4× and 2.6×, respectively.

Overall the GAP8 implementation that uses all the functionality of the core (i.e., popcount instruction and multi-core) is 10× faster and 51× more efficient than running the same network on the Cortex-M4F.

Fig. 3.4 shows the power trace of the layers in the same setup in Tbl. 3.4. As described in Sec. 3.4, we split the input data into tiles to match the memory constraints. The traces refer to one tile out of four. Thus the execution time is approximately one-fourth of the one presented in Tbl. 3.4. Between layers, the fabric controller offloads the cluster for configuring the next layer: it switches the input and output buffer, allocates memory for the next weights, configures the DMA, and so on. This behavior is visible in the drop of power traces because the cluster is in sleep, and the activity of the fabric controller consumes less. Similar behavior can be observed inside binary layers, where the processing is split in chunks of 32 channels.

3.6 Conclusions

Starting from the best-performing DNN for Sound event detection on our target dataset, we have proposed and trained a DNN with the same topology but binary weights and activations. The proposed BNN matches the memory and resource-constraints of milliwatt range of the target embedded platforms. The resulting BNN has an accuracy of 77.9%, a drop of 7.2 percent point from the full-precision baseline which is in line of similar state-of-the-art BNNs/TNNs (i.e., 6.5-19%). The resulting BNN requires 230 kB of RAM, 3.9× less than the system using 16-bit quantized baseline CNN. Due to this compression, the network fits in the GAP8 PULP Platform. We evaluated energy efficiency with experimental measurement of the power consumption of the full system. We evaluated energy efficiency with experimental measurement of the
power consumption of the full system. The classification of 3.2 s of audio requires 511 ms and 25.54 mJ, with a peak energy efficiency of 67.1 GMAC/s/W and average 34.5 GMAC/s/W. The performance on the GAP8 board has been shown to be 10× faster and 51× more energy-efficient than on an ARM Cortex-M4F platform, which comes from multi-core capabilities (i.e., 4.3/19.3×), the build-in `popcount` instruction (i.e., 2.4/2.6×). Nevertheless, using the BNN approach shows just a 1.45×\(^7\) better energy efficiency than the 8-bit quantized CNN on the same GAPuino microcontroller, presented in PULP-NN [173]. It has therefore to be considered that the gain in energy efficiency compared with the quantized neural network does not compensate for the high loss in accuracy.

\(^7\)Will be slightly higher, as the energy- and time-costly `im2col` algorithm is not included in the numbers of PULP-NN.
Chapter 4

Extending the RISC-V ISA for Efficient RNN-based 5G Radio Resource Management

Radio Resource Management (RRM) in 5G mobile communication is a challenging problem for which Recurrent Neural Networks (RNN) have shown promising results. Accelerating the compute-intensive RNN inference is therefore of utmost importance. Programmable solutions are desirable for effective 5G-RRM top cope with the rapidly evolving landscape of RNN variations. In this chapter, we investigate RNN inference acceleration by tuning both the instruction set and micro-architecture of a micro-controller-class open-source RISC-V core. We couple HW extensions with software optimizations to achieve an overall improvement in throughput and energy efficiency of $15 \times$ and $10 \times$ with respect to the baseline core on a wide range of RNNs used in various RRM tasks.\(^1\)

\(^1\)Hardware, software and benchmarks have been open sourced on GitHub https://github.com/andrire/RNNASIP
4.1 Introduction

RRM is challenging as it aims at achieving maximum utilization of the limited publicly available frequency bands [200], under highly heterogeneous traffic (e.g., tiny sensor-nodes vs. mobile routers), and rapidly varying radio signal propagation conditions. Notably, RRM tasks have to be executed in the frame of milliseconds, which exclude compute-intensive algorithms [196]. Presently, 5G applications impose strict new intensive requirements on radio communication systems:

1. Very high reliability and low-latency for autonomous vehicles.
2. Very high bandwidth requirements for video telephony and virtual reality.
3. Massive machine-to-machine communication for the Internet of (Every)-things.

These challenging requirements ask for extending the existing cellular network with more antennas, improving antenna efficiency, and more effective RRM. Therefore, more advanced allocation algorithms are required to distribute limited resources (e.g., frequency bands, transmit power, data rates) to mobile clients efficiently.

Typically, RRM problems have been modeled with full observability and solving convex problems with traditional optimization approaches. Exhaustive search methods led to very high computation costs [192] and sub-optimal solutions based on Lagrangian relaxation, iterative distribution optimization, and other heuristic approaches had convergence issues and lacked guarantees [192]. Traditional methods like the weighted sum-rate Mean Squared Error (MSE) algorithm [201] and fractional programming [202] are iterative, and most of them need to perform complex operations (e.g., matrix inversion or Singular Value Decomposition (SVD)) in every single iteration. It is, therefore, extremely challenging to push these methods to the throughput and scale required for 5G-RRM. Recently, neural networks have also gained increasing attention for 5G RRM. At the physical layer, RNNs have been used to compensate for imperfections and nonlinearities and collision detection in the RF domain [26, 27]. This is getting even more important for high-frequency communication, where absorption
4.1. INTRODUCTION

starts to strongly depend on the environment, and for ultra-dense cell networks where cross-tier interference has to be compensated [203]. At the data-link layer, which is responsible for resource allocation including dynamic resource scheduling of frequency bands, dynamic range, and handover control, classic multi-layer perceptron [195,197,198,204], (recurrent) Long Short-Term Memories LSTMs [190,191], and Convolution Neural Networks [194] have been used. Reinforcement learning-based deep Q-Learning networks [21] have been used for several typical RRM problems like dynamic spectrum access utilization [191,197,199], power level selection [195,197,204], rate control [204] and time-slotted optimization [198].

These networks are less computationally demanding than classical RRM algorithms, but they are far from trivial. Specialized and efficient stand-alone Neural Networks accelerators have been presented recently [205]. Nevertheless, hardwired RNN accelerators cannot cope with the flexibility requirements found in a typical RRM setting, as base stations typically stay in the field for a very long time, while RRM algorithms are rapidly evolving. To retain flexibility, FPGA-based acceleration has been explored for RNN inference. For instance LSTM acceleration on FPGA achieving up to 13 GMAC/s/W have been presented in Cao et al. [42] and Gao et al. [206]. To further increase efficiency, compression techniques (e.g., block-circulant weight matrices, pruning with zero-skipping [42, 206]) have been applied, and a top (effective) energy efficiency of 82 GMAC/s/W on a Xilinx Zynq-7100 FPGA has been presented in Gao et al. [206]. Nevertheless, these compression schemes have not yet been proven to work for the networks used in the RRM field, and FPGAs have a cost envelope that is not compatible with massive and dense deployment, as required in 5G networks. To address these intertwined flexibility, efficiency, and cost challenges, we propose to enhance the open and royalty-free RISC-V ISA and leverage the availability of high-quality open-source cores based on this widely supported ISA. We demonstrate a micro-controller class RISC-V core with RNN-enhancements for RRM acceleration, and we couple hardware extensions with software optimization. We achieve an energy efficiency of 218 GMAC/s/W, and a throughput of 566 MMAC/s, which is an improvement of 10× and 15×, respectively over the baseline open-source core. Such an order-of-magnitude boost
is obtained thanks to data reuse with output feature map tiling (1.9×), adding custom activation instructions (13% within LSTMs), merging load and compute (1.13×/1.7×), and input FM tiling (5%). The proposed extensions maintain backward compatibility with the baseline RISC-V ISA, and have a very small overhead (3.4%) in area and no increase in the longest path. Improvements are consistently achieved over a quite diverse set of RNNs used for various RRM tasks, thereby confirming the flexibility of our approach.

4.2 Related Works

4.2.1 Generic Software-Programmable Platforms

Also, the GPU’s architecture has been optimized for DNN workload, introducing tensor cores and fast half-precision floating-point (FP16) support. The latest device, Nvidia’s V100, achieves 112 TFLOPS at 250 W [207]—an energy efficiency of 448 GOp/s/W. Its best known competitor, the first version of Google’s TPU [75], works with 8-bit arithmetic and achieves 92 TOp/s at 384 W (240 GOp/s/W)

Since then we have seen optimized implementations [46,208] and algorithmic advances such as FFT-based and Winograd convolutions further raising the throughput [48,49]. The availability of easy-to-use deep learning frameworks (TensorFlow, Torch, Caffe, . . .) exploiting the power of GPUs transparently to the user has resulted in wide-spread use of GPU computing.

4.2.2 ML Compute Platforms

With the machine learning revolution, a variety of different ML compute platforms have been presented in industry and academia, spanning from high-performance server accelerators (e.g., Google’s TPU cores) to embedded platforms (e.g., Nvidia Jetson Xavier) to stand-alone application-specific accelerators [205]. Also the GPU’s architecture has been optimized for DNN workload, introducing tensor cores and fast half-precision floating-point (FP16) support. The latest device, Nvidia’s V100, achieves 112 TFLOPS at 250 W [207]—an energy efficiency of 448 GOp/s/W. Its best known competitor, the first
version of Google’s TPU [75], works with 8-bit arithmetic and achieves 92 TOp/s at 384 W (240 GOp/s/W).

General-purpose processors have been extended with new matrix and vector extensions to handle the common compute patterns in Neural Networks. In the Advanced Vector Extensions AVC-512 of the x86 ISA, Intel added the VNNIW instruction extension, which include 16×32-bit Single Instruction Multiple Data (SIMD) vector operation for efficient convolution kernels in half-precision float FP16 and accumulations in single-precision float FP32 and since Cascade Lake (2019) the fixed-point version (VNNI) with 8-bit (e.g., VPDBUSD) and 16-bit (e.g., VPDBUSSD) vector product with 32-bit accumulation [50]. The AARCH64 Neon extensions in the ARMv8-A processor series, provides special SIMD instructions for sum-dot-products (e.g., BFDOT) and 2×2 matrix-matrix multiplications (e.g., BFMMMA) with 2-way SIMD in brain floating-point format bfloat16. Recently, ARM presented the M-profile Vector Extensions MVE (Helium) for their embedded processor family Cortex-M. Helium instructions feature computations in various SIMD-formats (INT8/16/32, FP16/32), hardware loops, interleaved post-increment load/stores [51]. However Intel typically focuses on the high-performance high-cost processor market and the Helium extensions are not yet available in HW implementations.

Besides ISA extensions, also highly-optimized SW kernels have been developed exploiting these instructions. These includes utilizing parallel SIMD computations (e.g. 16-bit [53], 8-bit [173]) and data reuse with appropriate tiling. Tiling helps to reduce data loads from memory and reuse data with the local registerfile. Output Feature Map (OFM) tiling, where several outputs are calculated in parallel and input FM loads can be shared, has been commonly used (e.g., CMSIS [53], PULP-NN [173]). Furthermore, convolutional layers can be reformulated as matrix-matrix multiplications with the im2col technique [209]. This allows to tile both the input and output FM spatially in $m \times n$-sized tiles and thus reduces the number of loads from $O(mn)$ to $O(m + n)$, as both weights and input FM pixels can be reused. Previous work has mainly focused on and reported results on CNNs [53, 173]. Still, this two-dimensional tiling cannot be applied to (non-convolutional) LSTMs and Linear Layers, which are the main network kernels used in RRM applications.
Neural Networks are commonly trained in floating-point format, but recently, it has been shown that integer-aware training allows us to use more energy and area efficient fixed-point without any significant accuracy drop, especially 16-bit quantization [156], but even eight and fewer bits [188]. Finally, RNNs use transcendental activation functions, which are computationally complex. Previously, there have been 4 approaches to accelerate computation of these functions: Piecewise Linear Approximation (PLA) [53], low-order Taylor series expansion (e.g., 2nd order [210]), Look-Up Table (LUT) with adaptive value granularity [211], or a small neural network [212]. We use a PLA approach, but differently from previous work, we exploit the symmetry property of \( \text{tanh} \) and \( \text{sig} \), we take into account fixed-point quantization and evaluate in detail the error introduced by different numbers of interpolation intervals, rather then selecting a high number of intervals (i.e., 128 in [53]).

### 4.2.3 RISC-V and RI5CY

The PULP project has moved from the OpenRISC (i.e., PULPv3 used in Chapter 2) to the RISC-V ISA. The RISC-V ISA [213] has recently become the de facto standard in open-source and free instruction set architecture. RISC-V provides plenty of encoding space for extensions and is therefore suitable for application-driven processor customization while maintaining compatibility with the baseline ISA. In this work, we rely on the RI5CY [214], a high-quality, silicon-proven and open-source core supporting the standard RISC-V RV32IMFC ISA (including integer, integer multiplications, single-precision floating-point, and compressed instructions). Additionally, RI5CY supports the Xpulp ISA extensions featuring extended fixed-point support (e.g., on-the-fly re-quantization and saturation), SIMD instructions, post-increment store and loads, and hardware loops. Tab. 4.1 gives an overview of these commonly used instruction extension with a basic example of pointwise vector-addition:

```c
for (i = 0; i < 100; i++)
    d[i] = a[i] + b[i];
```
4.2. RELATED WORKS

Table 4.1. Assembly Code Example of RISC-V RV32IMFC ISA with relevant RI5CY extensions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mv x5, 0</td>
<td>2</td>
<td>mv x4, 100</td>
<td>3</td>
<td>lb x2,0(x10)</td>
<td>4</td>
<td>lb x3,0(x11)</td>
<td>5</td>
<td>addi x10,x10,1</td>
</tr>
<tr>
<td>6</td>
<td>addi x4,x4,-4</td>
<td>7</td>
<td>addi x11,x11,1</td>
<td>8</td>
<td>add x2,x3,x2</td>
<td>9</td>
<td>sb x2,0(x12)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>addi x12,x12,1</td>
<td>11</td>
<td>bne x4,x5, Lstart</td>
<td>12</td>
<td>bne x4,x5, Lstart</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1'002 cycles | 702 cycles | 501 cycles | 126 cycles
RV32IMFC | +post-incr. lw/sw | +HW Loops | +SIMD Support

Post-increment load and store merges the load (e.g., lb in line 3) and the incrementing of the addressing (i.e., addi in line 5) and improves the cycle count by 30% in the inner-loop. Hardware Loops (also Zero-Overead Loop) uses a hardware counter to decide on the branching and needs a single instruction to set up the counter, and therefore removes the decrementation of the counter register in line 6 and the branch instruction in line 11. This gives another 28% improvement. Finally, packed SIMD instruction replace the byte wise instruction with a vector-like instruction applying the operation on every item of the vectors, which can improve the performance up to 4× when using 8-bit words (i.e., 1 pv.add.b instead of 4 add instructions).

4.2.4 Benchmark Suite and Neural Networks

We have selected an application benchmark consisting of 10 neural networks which have been presented recently in the RRM domain. These networks differ in network types (Fully-Connected Neural Layers [192,193,195–198], Long-short Term Memories [190,191], Convolutional Neural Network [194]), learning methods (Supervised [190,193,194,196], reinforcement-based [191,195,197–199], unsupervised [192]), application (cellular networks [190,192], peer-to-peer communication [191],...
wireless communication systems [193–195, 198, 199], wired communication [196]) and optimization metric (throughput [190–196, 198, 199], fairness [190, 191], latency [197], energy efficiency [194]).

Table 4.2 gives an overview of the presented benchmark networks comparing the different optimization objectives, network structure, and network types. Not all of the papers have published the exact setup (e.g., number of access nodes or frequency bands) and, therefore, also not all relevant neural network parameters (e.g., input and output neurons). In the following, if the numbers have not been indicated in the paper, we have set the number of antennas to $K = 4$, and the number of frequency bands to $N = 3$. The following sections give a short summary on the networks and the corresponding work:

**Proactive Resource Management in LTE-U Systems: A Deep Learning Perspective [190]**

Challita et al. have presented a learning framework in the field of Long Term Evolution in unlicensed spectrum (LTE-U) (4G), whereas Small (cell) Base Stations (SBS)\(^2\) are sharing unlicensed bands. The number of bands is rather small, therefore fairness and an altruistic policy is needed and should be learned. The SBS’ are not collaborating directly but try to achieve long-term fairness measured in average airtime per radio while optimizing proactively dynamic channel selection, carrier aggregation, and fractional spectrum access. Challita et al. show that they reach a mixed-strategy Nash equilibrium and can gain up to 28% over a conventional reactive approach. They use a combination of LSTMs and MLPs: The state encoder tries to reconstruct the predicted action sequence and is modeled with a one-layer LSTM with 70 cells, followed by a summarizing fully-connected layer with unknown size and has been set to the same size (i.e., 70) and is followed by the action decoder modeled by a one-layer 70 cell LSTM with $K = 4$ output neurons (i.e., one for every antenna).

Furthermore, the authors model the throughput maximization problem of a small cell network with unlicensed spectra as a non-cooperative game and propose a Deep Learning (DL)-based solution.

\(^2\)low-powered cellular radio access nodes with a range of 10 m to 1 km, few concurrent connections/sessions
Table 4.2. Benchmark list for different network models used in the RRM setup.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Paper</th>
<th>Optimization Objective</th>
<th>Network structure</th>
<th>Network Typ</th>
</tr>
</thead>
<tbody>
<tr>
<td>[191]</td>
<td>Deep Multi-User Reinforcement Learning for Distributed Dynamic Spectrum Access</td>
<td>Throughput (Decentralized multi-agents)</td>
<td>(2N+2)-(2N+2)-(2N+2)</td>
<td>LSTM-FC, DQN</td>
</tr>
<tr>
<td>[197]</td>
<td>Deep reinforcement learning for resource allocation in V2V communications</td>
<td>Interference under latency constraints</td>
<td>6-500-250-120-3N</td>
<td>FC-MLP, DQN</td>
</tr>
<tr>
<td>[204]</td>
<td>A reinforcement learning approach to power control and rate adaptation in cellular networks</td>
<td>Throughput and Power</td>
<td>15-?-10</td>
<td>FC-MLP, DQN</td>
</tr>
<tr>
<td>[198]</td>
<td>Deep Reinforcement Learning Multiple Access for Heterogeneous Wireless Networks</td>
<td>(Sum-)Throughput and Fairness</td>
<td>20-64-64-64-64-64</td>
<td>FC-MLP, DQN</td>
</tr>
<tr>
<td>[195]</td>
<td>Deep Reinforcement Learning for Distributed Dynamic Power Allocation in Wireless Networks</td>
<td>Throughput</td>
<td>(10K+7)-200-100-40-1</td>
<td>FC-MLP, DQN</td>
</tr>
<tr>
<td>[194]</td>
<td>Deep Power Control: Transmit Power Control Scheme b. on CNN</td>
<td>Throughput or Energy Efficiency</td>
<td>$10^2\cdot(71\times8\cdot10^2)-10$</td>
<td>CNN (3x3)</td>
</tr>
<tr>
<td>[199]</td>
<td>Deep reinforcement learning for dynamic multichannel access in wireless networks</td>
<td>#Successfull transmission (Throughput)</td>
<td>512-200-200-16</td>
<td>FC-MLP, DQN</td>
</tr>
</tbody>
</table>
A Deep Reinforcement Learning (DRL) algorithm is developed based on LSTM, for channel selection, carrier aggregation, and fractional spectrum access. The primary objective is to maximize throughput in each small cell while maintaining fairness with co-existing networks.

**Deep Multi-User Reinforcement Learning for Distributed Dynamic Spectrum Access [191]**

Naparstek et al. apply Deep Q-Learning to solve the Dynamic Spectrum access (DSA) utilization problem. A short introduction into Deep Q-Learning is presented in Section 4.2.8. The time is slotted into fixed-size times slots, and each user selects a channel and transmits a packet with a certain attempt probability. After each slot, the user gets an acknowledgment of whether the transmission was successful or not. The problem is learned with a Deep Q-Network (DQN) approach, whereas the network consists of $(2N + 2)$ input neuron where the first $(N + 1)$ neurons are encoding the last channel selected, and the other $(N)$ encodes the capacity of the $N$ sub-bands and 1 for the acknowledge signal. These neurons connect to a single-layer LSTM with unknown size (i.e., set to $2N + 2$), and finally fed through two independent linear layers (the value layer and the advantage layer) based on the dueling DQN principle; both layers have $N + 1$ output neurons each.

As the exact network topology has not been published, the number of layers is set to 1. Based on double Q learning, the network is trained separately to choose the action and to estimate the Q-value associated with the corresponding action.

**Learning to optimize: Training deep neural networks for wireless resource management [196]**

H. Sun et al. are solving the general problem of interference channel power control, with $K$ single-antenna transceivers pairs sending data as a Gaussian random variable and independently of each other. Differently, the previous state of the art is learning a MLP to perform the Weighted Minimum Mean Square Error (WMMSE) algorithm. The input to the network is the magnitude of the channel coefficients, and the output is the power allocations. Two models are evaluated: Model 1 is considering all channel coefficient to be Rayleigh fading
distributed with zero mean and unit variance, which has been used in various resource allocation algorithm. Model 2 considers a multi-cell interfering Media Access Control (MAC) setting with $N$ regularly placed cells and $K$ randomly distributed users. The proposed network consists of $K^2$ (Model 1) or $N \times K$ (Model 2) input neurons for the channel coefficients, and the output is the set of power allocations $K$ and 3 hidden layer with 200 neurons each. The presented results have worse accuracy (2-16%) than the baseline algorithm (i.e., WMMSE), but are up to $33 \times$ faster.

Deep reinforcement learning for resource allocation in V2V communications [197]

Ye et al. elaborate on the resource allocation problem in vehicle-to-vehicle and base station-to-vehicle communication setup (e.g., information on traffic safety). Every vehicle is an agent deciding independently to reach an optimal band and power level selection. The system is modeled in a Q-Learning setup, whereas the reward is determined by latency and reliability metrics. The state is based on the channel information of the corresponding link, previous interference on the link, channel information to the base station, selected sub-channel of neighbors in the previous time slot, the remaining load to transmit, and the remaining time to meet latency constraint. The actions are the sub-band to select and transmission power. The Q function is then learned and modeled by a 5 layer fully-connected neural network with 500,250,120 hidden neurons. There are 6 input neurons and $2N$ (#frequency bands) output neurons.

A reinforcement learning approach to power control and rate adaptation in cellular networks [204]

Ghadimi et al. propose a DQN learning approach combined with ensemble learning to optimize downlink power control and rate adaption in cellular networks and to overcome the limitations of missing system observability in previous approaches. Agents are not collaborating and not controlled at a centralized unit. The cell power represents
the state, average Reference Signal Received Power (RSRP)\(^3\), Average interference and cell reward. The action is the increase or decrease of the transmission power by \(\{0, \pm 1, \pm 3\}\) DB, and the reward is based on the \(\alpha\)-fair resource allocation utility function \([215]\). An ensemble of several (fully-connected forward) DQN with 3 hidden layers has been trained, but topologies are unknown.

**Deep-Reinforcement Learning Multiple Access for Heterogeneous Wireless Networks \([198]\)**

The work of Yu et al. focuses on the problem of sharing time slots among multiple time-slotted heterogeneous network nodes adopting different MAC protocols (CDMA, TDMA, and ALOHA) objective to sum throughput or \(\alpha\)-fairness. All nodes connect to the same base station. The network nodes work independently in a re-inforced way. Possible actions \(a_t\) are transmit and wait and reward or channel observation \(r_t\) are 'success', 'collision', or 'idleness'. The state is defined as a history of state/action pairs (of \(M = 20\) length). The Q function is implemented by a multi-layer fully-connected network with \(5M = 100\) input neurons and 2 output neurons, and 6 hidden layers, whereas all of them have 64 neurons and two residual paths are added between layer 3 and 5 and between 5 and 7. The network output 2 Q values for 'transmit' and 'wait' action.

**Learning Optimal Resource Allocations in Wireless Systems \([193]\)**

Eisen et al. are formulating the general resource allocation problem as a Lagrangian Dual Problem and show that the duality gap is zero when learned by a neural network (converging to a local minimum). Two problems are looked at:

1. A simple capacity maximization over a set of simple Additive White Gaussian Noise (AWGN) wireless fading channel, whereas \(K\) users are given a dedicated channel to communicate under the constraint of a total expected power budget.

\(^3\)A metric for average user distance.
2. Capacity maximization in an interference channel with $K$ transmitter sending to a single access point.

As the 2nd example has a non-convex capacity, it is not solvable with the classic dual approach, but with a neural network. The neural network is built with fully-connected layers and $K$ input neurons, 32 and 16 hidden neurons, and $K$ output neurons. The number of users has been set to $K = 4$.

**Deep Reinforcement Learning for Distributed Dynamic Power Allocation in Wireless Networks [195]**

Nasir et al. are presenting another model-free but distributed approach for the power allocation scheme on single frequency bands based on deep reinforcement learning. All transmitters collect Channel State Information (CSI) and Quality of Service (QoS) information from several neighbors. The network is learned in a centralized server, observations and training data is collected at the nodes and transmitted to the central unit, and the weights are updated simultaneously on the base stations. The state of the Deep Q network is based on local information (transmit power in the last time slot, its contribution ratio, downlink channel measurement, total interference-plus-noise power at its own receiver), and the interference from its neighbors or to the neighbors. The system is time-slotted, and actions are taking instantaneously. The actions are the discrete power levels selected, and the reward function is defined as the contribution to the collective spectral efficiency minus the penalty of caused interference. The network consists of one input layer with 7 internal states, 6c interferer neighbor state, and 4c interfered neighbor states, which are 57 input states for the use case of 5 agents. The hidden layers consist of 200, 100, and 40 neurons and 10 neurons or 10 discrete power levels have been chosen.

**Deep Learning for Radio Resource Allocation in Multi-Cell Networks [192]**

Ahmed et al. are looking at the sub-band and power allocation problem in a multi-cell network (with $K$ cells, $U$ users, and $N$ sub-bands).
Differently to previous approaches, the base stations are exchanging channel quality indicators to their neighbors. Stacked-autoencoder is used and pre-trained with a genetic algorithm, before their encoder parts are stacked to an MLP with $K \cdot K \cdot (N + 1) = 100$ input neurons and 4 hidden layers with 1080, 720, 360 and 180 neurons, followed by a softmax layer with 180 output neurons.

**Deep Power Control: Transmit Power Control Scheme based on CNN [194]**

Lee et al. are optimizing spectral efficiency (throughput) and energy efficiency in an environment with $N$ single-antenna transceiver pairs. The state is determined by $h_{i,j} = |g_{i,j}|G_{i,j}$ which is composed of the distance related channel gain $G_{i,j}$ and the multipath fading $g_{i,j}$ between the transmitter $i$ and receiver $j$. After normalization, the $N^2$ state features are fed to a neural network with 7 convolutional layers with $3 \times 3$ kernels and 8 intermediate channels, followed by a single fully-connected layer with $N$ output neurons which are fed to sigmoid activation layer to determine the transmit power. While having full channel information, this approach is slightly better than WMMSE and one order of magnitude faster. In the distributed case where just a few information is transmitted, and just a part of the channel information is available, the performance is just slightly worse than WMMSE algorithm.

**Deep reinforcement learning for dynamic multichannel access in wireless networks [199]**

Wang et al. consider a multichannel access problem, whereas there are $N = 16$ correlated channels, each of which has two possible states (good or bad), and their joint distribution is modeled as a (partially observable) Markovian model. They learn a Deep Q network, which is learned centralized. A single user at a time can select a channel to transmit a packet, and either it is successfully sent (reward = 1) or failed due to a bad state (reward = -1). The state of the agent is defined as a set of $M = N = 16$ previous actions and observed channel conditions, and the action is the selected channel. The DQN
consists of the $M \cdot N \cdot 2 = 512$ input neurons, two hidden layers with 200 neurons each, and $N$ output neurons.

### 4.2.5 Neural Networks in RRM

Three main ML kernels are used within these networks: Fully-connected layers (or Multi-Layer Perceptron MLP), Long-short Term Memories LSTM, and Convolutional Neural Network CNN Layer. A fully-connected layer connects all input (neurons) $\mathbf{x} \in \mathbb{R}^m$ to all outputs (neurons) $\mathbf{o} \in \mathbb{R}^n$ and is described with the following matrix-vector multiplication and the corresponding weight matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$:

$$\mathbf{o} = \mathbf{b} + \mathbf{Wx} \quad (4.1)$$

CNN layers exploit the translation invariance in the data (e.g., in images) and map $n$ $h_{im, in} \times w_{im, in}$-sized input channels $i_n \in \mathbb{R}^{h_{im, in} \times w_{im, in}}$ to $k$ $h_{im, out} \times w_{im, out}$-sized output channel maps by applying $h_k \times b_k$-sized convolution filters $w_{k,n} \in \mathbb{R}^{h_k \times b_k}$ to every input channel for every output channel.

More details on Neural Networks and CNNs have been introduced in Section 2.4.2 and 2.4.3. Recurrent Neural Networks and LSTMs are introduced in the following two sections. Furthermore, Section 4.2.8 introduces briefly reinforcement learning and Q-learning.

### 4.2.6 Recurrent Neural Networks RNN

The sequential property of time series data (e.g., audio samples) in RNNs is represented with recurrent vertices in the network model, as illustrated in Fig. 4.1 whereas $U_h \in \mathbb{R}^{m \times m}$ are the recurrent weights of a single-layer RNN. The network can then be written as:

$$\mathbf{h}_t = \sigma_h(W_h \mathbf{x}_t + U_h \mathbf{h}_{t-1} + \mathbf{b}_h)$$
$$\mathbf{y}_t = \sigma_y(W_y \mathbf{h}_t + \mathbf{b}_y)$$

RNN can support variable length in sequential models, but suffer from vanishing gradient problem during training making training slow.
and long-term dependencies hard to train. RNNs have been trained on a large set of applications, e.g., to create automatic image caption [217], text generation like poems, wikipedia article, linux kernels and language translation [218].

4.2.7 Long Short-Term Memory

Hochreiter and Schmidhuber introduced LSTMs, an extension to the vanilla RNNs, where an additional internal memory cell \( c_t \) and a forget gate \( f_t \) is added. LSTMs have been shown to be much less prone to the vanishing gradient problem and can therefore learn much longer time-dependencies [219]. The following formula and Fig. 4.2 show the structure of a typical LSTM cell unfolded in time:

\[
\begin{align*}
\mathbf{o}_t(x_t, h_{t-1}) &= \operatorname{sig}(W_o x_t + U_o h_{t-1} + b_o) \tag{4.2} \\
\mathbf{f}_t(x_t, h_{t-1}) &= \operatorname{sig}(W_f x_t + U_f h_{t-1} + b_f) \tag{4.3} \\
\mathbf{i}_t(x_t, h_{t-1}) &= \operatorname{sig}(W_i x_t + U_i h_{t-1} + b_i) \tag{4.4} \\
\mathbf{g}_t(x_t, h_{t-1}) &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{4.5} \\
\mathbf{c}_t(x_t, c_{t-1}) &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t \tag{4.6} \\
\mathbf{h}_t(o_t, c_t) &= \mathbf{o}_t \circ \tanh(c_t) \tag{4.7}
\end{align*}
\]
4.2. RELATED WORKS

Whereas the weight matrices $W_o, W_f, W_i, W_g \in \mathbb{R}^{n \times m}$ and $U_o, U_f, U_i, U_c \in \mathbb{R}^{m \times m}$ and bias vectors $b_o, b_f, b_i, b_c \in \mathbb{R}^{m}$, and $\circ$ indicates the (point-wise) vector multiplication. The following activation functions are used in LSTMs:

- $\sigma_g$: sigmoid function $\sigma_g(x) = \sigma(x) = \frac{1}{1+e^{-x}}$
- $\sigma_c$: hyperbolic tangent $\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- $\sigma_h$: hyperbolic tangent or identity.

Even though LSTMs had already been presented in 1997, the big breakthrough came in 2015. LSTMs were introduced in sequence learning, sequence translation, and machine translation (e.g., text-to-image, image-to-text, or automatic video captioning), forming the base for speech assistant systems like Apple’s Siri, Microsoft’s Cortana, Google Assistant, and Amazon’s Alexa [221, 222].

4.2.8 Reinforcement Learning and Q-Learning

Q-Learning is a reinforcement learning technique, whereas reinforcement learning is beside (semi-)supervised and unsupervised learning the third category in machine learning. Instead of learning of labeled
data (i.e., supervised learning), or clustering of static unlabeled data
(i.e., unsupervised learning), it is more a dynamic learning system.
Typically, the model setup includes an agent in an environment, which
can have a set of states \( s_i \in S \) and can apply an action of a set \( a \in A \).
After applying action \( a \), the environment returns a reward \( r \in R \) and
the next state \( s_{i+1} \in S \) to the agent.

The action policy learned in Q-Learning is represented by the
Q-function \( Q : S \times A \rightarrow \mathbb{R} \) mapping state/action pairs to the expected
reward. The learning procedure happens iteratively by updating \( Q \) in
the following way:

\[
Q_{k+1}(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q_k(s_t, a_t) + \alpha \cdot \left( r_t + \gamma \cdot \max_{a \in A} Q(s_{t+1}, a) \right)
\]

where as \( \alpha \) is the learning rate and \( \gamma \) accounts for future rewards.
Typically the agent applies the action which has the highest Q values for
the current state (exploitation) or selects a random action (exploration)
[21].

In traditional Q-Learning, a large Q table is learned, including all
\( |S| \cdot |A| \) possible entries; obviously, this limits the possible number
of state-action pairs. Recent works instead suggest to learn a (deep)
neural network to represent the Q function, also known as DQN [21],
which also works for a continuous state-action space.

In dueling DQN Q function is split into the state value \( V(s) \) and
the advantage value \( A(s, a) \), whereas \( Q(s, a) = A(s, a) + V(s) \). Both
values are being learned by two separate neural networks [223]. DDQN
can better account for cases, where the state is either good or bad
independently from the taken action.

4.3 HW/SW Extension and Optimizations

4.3.1 Baseline Implementation (SW)

We have developed a straight-forward implementation (e.g., organizing
matrix-vector multiplication as a double nested loop over all inputs
and outputs) of all required network kernels in C where weights and
data values are encoded in 16-bit fixed-point format (i.e., $Q_{3.12}$). This format offers a good compromise between accuracy/robustness and energy-efficiency/throughput, and most importantly does not require fixed-point aware retraining that would be necessary for smaller bit-widths. The C implementation is compiled with standard GCC 7.1.1 for RISC-V RV32IMFC ISA and was run on the RI5CY core. The instruction count for the entire benchmark suite is shown in Tab. 4.3a and is used as the baseline for further comparisons.

### 4.3.2 SIMD, HWL and post-increment load (HW)

As a first optimization step, we re-wrote each code to exploit Xpulp extensions as much as possible. The 16-bit data (weights and inputs) are packed into the packed SIMD vector format (i.e., $\mathbf{v2s}$), allowing the compiler to map every two subsequent input FM $p(2c_i)$ and $p(2c_i + 1)$ and the corresponding weights ($c_o, 2c_i$) and $w(c_o, 2c_i + 1)$ (two macs) to a single $\mathbf{pv.sdotsp.h}$ instruction without the need of custom intrinsics.

\[
o(c_o) = b(c_o) + \sum_{c_i=1}^{cin} p(c_i) \cdot w(c_o, c_i) \quad (4.8)
\]

\[
= b(c_o) + \sum_{c_i=1}^{cin/2} \left( \frac{p(2c_i)}{p(2c_i + 1)} \right) \left( \frac{w(c_o, 2c_i)}{w(c_o, 2c_i + 1)} \right) \quad (4.9)
\]

The next optimization is to reduce the overhead of loop control instructions in small loop bodies that are seen in such operations by using hardware loops that are part of the Xpulp extensions. The hardware loop does not use any additional instructions during the loop execution, but requires loop index manipulation instructions (i.e., $\mathbf{pl.setup}$) to set three registers: a loop counter ($rB$), the loop start PC+4 and the loop end (PC+rA). When the PC reaches the loop end, the controller decrements the loop counter and jumps back to the loop start until the loop counter reaches zero.

The final optimization is to take advantage of post-increment load-word instruction (i.e., $\mathbf{lw!}$) to increment the data pointer for weights and input feature maps at the same time as executing the load word instruction, saving a separate $\mathbf{addi}$ instruction in the process.
Figure 4.3: Speedup with respect to the RISC-V IMC baseline implementation for a typical Neural Networks workload in RRM.

<table>
<thead>
<tr>
<th>Legend:</th>
<th>Xpulp ext. (HW)</th>
<th>+Output FM Tiling (SW)</th>
<th>+tanh/sig ext. (HW)</th>
<th>+VLIW ext. (HW)</th>
<th>+Input FM Tiling (SW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>8.4 8.4</td>
<td>14.3 15.0</td>
<td>1.9 9.1</td>
<td>15.7 16.9</td>
<td>15.4 16.2</td>
</tr>
<tr>
<td>LSTM/FC/CNN</td>
<td>1.7 8.6</td>
<td>14.5 16.3</td>
<td>1.9 9.1</td>
<td>15.7 16.9</td>
<td>15.4 16.2</td>
</tr>
<tr>
<td>LSTM/FC</td>
<td>1.7 8.6</td>
<td>14.5 16.3</td>
<td>1.9 9.1</td>
<td>15.7 16.9</td>
<td>15.4 16.2</td>
</tr>
<tr>
<td>Fully-Connected NN</td>
<td>1.7 8.6</td>
<td>14.5 16.3</td>
<td>1.9 9.1</td>
<td>15.7 16.9</td>
<td>15.4 16.2</td>
</tr>
<tr>
<td>CNN</td>
<td>1.7 8.6</td>
<td>14.5 16.3</td>
<td>1.9 9.1</td>
<td>15.7 16.9</td>
<td>15.4 16.2</td>
</tr>
</tbody>
</table>

Table 4.3. Instruction Histogram for different SW/HW optimization levels
Combining these three techniques results in $4.4 \times$ reduction w.r.t. to the unmodified RISC-V IMC baseline in the number of instructions executed as can be seen in Tab. 4.3b.

### 4.3.3 Output Feature Map Tiling (SW)

To compute one MAC two loads to the memory are needed: one for the weight and one for the value of the corresponding input neuron. Fortunately, the read for the input value can be reused for several outputs. The output features are therefore organized in tiles of $N$ output channels and the contribution of the input neurons is calculated for all output neurons of the current input neuron. These partial sums can be stored in registers and are not written back to the memory until all input activations have been weighted and accumulated. Algorithm 2 gives an overview of the implementation and scheduling of the output FM tiling. The load of one input FM can thus be

---

**Algorithm 2** Fully-Connected Layer with Output FM Tiling

Require: All weights $w_{mn}$ and input activations $i_{m}$ for all input channels $m \in c_{in}$ and output channels $n \in c_{out}$ in memory

1. for all d-sized output channel tiles $\tilde{o}_k = \{o_{k,d}, ..., o_{(k+1)d}\}$ do
2. for all output channels $o_l$ in $\tilde{o}_k$ do
3. temp_out$[o_l] = 0$
4. end for
5. for all input channels $i_l \in c_{in}$ do
6. temp_in = Mem($i_l$)
7. #unroll following loop
8. for all output channel $o_k$ in tile $\tilde{c}_{out}$ do
9. w = Mem($w_{o_k,i_l}$)
10. temp_out$[o_k] +=$ temp_in $\times$ w
11. end for
12. for all output channels $o_k$ in $\tilde{o}_k$ do
13. temp_out$[o_k] =$ temp_out$[o_k] \gg$ 12 // requantize
14. Mem($o_k$) = temp_out$[o_k]$
15. end for
16. end for
17. end for

---
shared by $N$ \texttt{pl.sdotsp} instructions (executing 2 MAC operations on 16-bit operands), and thus just $O(1 + 1/N)$ loads are needed per compute operation. Until the available registers are exhausted and data has to be pushed onto the stack memory; furthermore, the load latency can be hidden by the compiler by rearranging the instructions. Previous work has shown that the tiling can be extended to the feature-level in case of a convolutional layer if the input feature map is rearranged and replicated (i.e., \texttt{in2col}) such that the convolution becomes a matrix-matrix multiplication [53,173].

In this chapter, we focus mainly on the optimizations for LSTMs and MLPs, as these network kernels are mostly used in the selected RRM benchmark suite and have not been discussed in previous work. As can be seen in Tab. 4.3c, the optimal tiling brings an additional improvement of $1.89 \times$ on the RRM benchmark.

The results are shown in Tab. 4.3c and Fig. 4.3, most of the networks execution cycles can be improved between $1.79 \times$ [198] and $1.87 \times$ [199], but small FMs suffer from high overhead and therefore less speedup ($1.07 \times$ [193] and $1.30 \times$ [191]).

### 4.3.4 Tanh and Sigmoid Extension (HW)

![Figure 4.4: RNN RISC-V Core with extensions to RI5CY core [214] in blue and datapath for \texttt{pl.sdotpsp} instruction marked in bold.](image)

Sigmoid and hyperbolic tangent are common activation functions in neural networks and used in LSTMs. The piece-wise linear approximation technique can be implemented for these functions in SW with
an increasing number of cycles to reach the required precision. This can be a major contribution to the overall calculation in LSTM-based networks. For example, the calculation of \( \tanh/\text{sig} \) requires 10.3\% in [190] and 33.6\% in [191] of the overall computation cycles. We introduce two single-cycle instructions \texttt{pl.tanh rD, rA} and \texttt{pl.sig rD, rA}. Both function have the following properties, which we are exploited for efficient and precise approximation:

1. They are continuous and smooth (i.e., derivatives are continuous, too); thus, the error is bound for a fixed interval in a Taylor series expansion even for degree one (i.e., \( \tanh(x_0 + \epsilon) = \tanh(x_0) + \tanh'(x_0) \cdot \epsilon \)).

2. The functions converge fast to either 0, 1 or \(-1\). Interpolation is needed only on this limited range of numbers.

3. Both functions are symmetric around 0 (i.e., \( \tanh(-x) = -\tanh(x) \) and \( \text{sig}(-x) = 1 - \text{sig}(x) \)), thus just the positive number range needs to be interpolated and the negative range can be derived from the positive values.

Alg. 3 shows the pseudo-code that was used for the hardware implementation of the proposed interpolation. First, we chose the number of intervals of \( M \) and the size of every interval \( 2^N \), whereas the interpolation range is \( \pm M \cdot 2^N \). For both functions \( f \in \{ \tanh(\cdot), \text{sig}(\cdot) \} \) two \( M \)-entry LUTs \( \text{lut}_m[f][] \) and \( \text{lut}_q[f][] \) are defined. Then the absolute value is calculated (line 2), and the index is calculated by a right shift of the absolute value by \( N \) places; if the result is larger than \( M \), it is considered to be in the convergence area and either \( \{-1, 0, 1\} \) is returned. Otherwise, the value is calculated by linear approximation within the selected interval \( id \) (line 8), sign inverted for negative values (line 9), and subtracted from 1 for negative values in the sigmoid case (l. 10).

We evaluate the proposed piece-wise linear approximation with different number of intervals \( 2^N \) and interpolation ranges, taking into account that fixed-point operations using the \( Q_{3.12} \) format are used. The result of this evaluation is illustrated in Fig. 4.5a and Tab. 4.4. For the actual implementation, we have selected an interpolation range of \([-4, 4]\) and \( 2^5 = 32 \) intervals, which produces an Mean Squared
Algorithm 3 Pseudocode of the $\text{sig}$ and $\text{tanh}$ Interpolation

**Require:** value $x$ and function $f \in \text{tanh}(\cdot), \text{sig}(\cdot)$, interval size $2^N$ and $\#$ intervals $M$

1: $|x| = \begin{cases} -x & \text{sgn}(x) = -1 \\ x & \text{sgn}(x) = 1 \end{cases}$
2: id = $|x| \gg N$
3: if id $>$ M then
4: \hspace{1em} return $\begin{cases} 1 & \text{sgn}(x) = 1 \\ 0 & \text{sgn}(x) = -1, f = \text{sig}(\cdot) \\ -1 & \text{sgn}(x) = -1, f = \text{tanh}(\cdot) \end{cases}$
5: else
6: \hspace{1em} (m,q)=(lut_m[f(id), lut_q[f(id)]
7: \hspace{2em} $y = m|x| + q$
8: \hspace{1em} return $\begin{cases} -y & \text{sgn}(x) = -1 \\ y & \text{sgn}(x) = 1 \end{cases}$
9: \hspace{1em} $f = \text{sig}, \text{sgn}(x) = -1$
10: \hspace{1em} $y$ else
11: end if

![Graphs](image)

(a) Mean Square Error                     (b) Max Absolute Error

Figure 4.5: Hyperbolic Tangent interpolation, sweep of interpolation ranges and number of intervals with $Q_{3.12}$ quantization.
### HW/SW EXTENSION AND OPTIMIZATIONS

#### 4.3

Table 4.4. Mean Squared Error and Maximum Absolute Error (in $\log_{10}$) of the sigmoid and tangent hyperbolic function interpolation for various intervals (rows) and number of intervals (columns)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>-0.7078</td>
<td>-0.7078</td>
<td>-0.7078</td>
<td>-0.7078</td>
<td>-0.7078</td>
<td>-0.7078</td>
<td>-0.7078</td>
<td>-0.7078</td>
</tr>
<tr>
<td>1</td>
<td>-1.0867</td>
<td>-1.1052</td>
<td>-1.1052</td>
<td>-1.1052</td>
<td>-1.1052</td>
<td>-1.1052</td>
<td>-1.1052</td>
<td>-1.1052</td>
</tr>
<tr>
<td>2</td>
<td>-0.5824</td>
<td>-0.7017</td>
<td>-1.2367</td>
<td>-1.8050</td>
<td>-2.2260</td>
<td>-2.2260</td>
<td>-2.2260</td>
<td>-2.2260</td>
</tr>
<tr>
<td>4</td>
<td>-0.3880</td>
<td>-0.3366</td>
<td>-0.7017</td>
<td>-1.2367</td>
<td>-1.8050</td>
<td>-2.4101</td>
<td>-2.9912</td>
<td>-3.3980</td>
</tr>
<tr>
<td>8</td>
<td>-0.2023</td>
<td>-0.1600</td>
<td>-0.3366</td>
<td>-0.7017</td>
<td>-1.2367</td>
<td>-1.8050</td>
<td>-2.4101</td>
<td>-2.9912</td>
</tr>
<tr>
<td>16</td>
<td>-0.1086</td>
<td>-0.0776</td>
<td>-0.1600</td>
<td>-0.3366</td>
<td>-0.7017</td>
<td>-1.2367</td>
<td>-1.8050</td>
<td>-2.4101</td>
</tr>
</tbody>
</table>

(a) $(\log_{10})$ Maximum Error of the tangent hyperbolic function.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-1.9391</td>
<td>-2.3077</td>
<td>-3.5876</td>
<td>-4.7820</td>
<td>-5.6056</td>
<td>-5.8004</td>
<td>-5.8144</td>
<td>-5.8162</td>
</tr>
<tr>
<td>8</td>
<td>-0.5759</td>
<td>-1.2621</td>
<td>-1.6062</td>
<td>-2.3071</td>
<td>-3.5876</td>
<td>-4.8201</td>
<td>-6.0081</td>
<td>-7.1030</td>
</tr>
<tr>
<td>16</td>
<td>-0.3574</td>
<td>-1.1106</td>
<td>-1.2621</td>
<td>-1.6062</td>
<td>-2.3071</td>
<td>-3.5876</td>
<td>-4.8201</td>
<td>-6.0081</td>
</tr>
</tbody>
</table>

(b) $(\log_{10})$ Mean Squared Error of the tangent hyperbolic function.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1.5495</td>
<td>-1.5495</td>
<td>-1.5495</td>
<td>-1.5495</td>
<td>-1.5495</td>
<td>-1.5495</td>
<td>-1.5495</td>
<td>-1.5495</td>
</tr>
<tr>
<td>2</td>
<td>-1.3864</td>
<td>-1.5395</td>
<td>-2.1129</td>
<td>-2.2849</td>
<td>-2.2849</td>
<td>-2.2849</td>
<td>-2.2849</td>
<td>-2.2849</td>
</tr>
<tr>
<td>4</td>
<td>-1.0571</td>
<td>-1.0028</td>
<td>-1.5395</td>
<td>-2.1129</td>
<td>-2.6721</td>
<td><strong>-3.1843</strong></td>
<td>-3.4216</td>
<td>-3.4079</td>
</tr>
<tr>
<td>8</td>
<td>-0.6888</td>
<td>-0.6374</td>
<td>-1.0028</td>
<td>-1.5395</td>
<td>-2.1129</td>
<td>-2.6721</td>
<td>-3.1843</td>
<td>-3.4216</td>
</tr>
<tr>
<td>16</td>
<td>-0.5039</td>
<td>-0.4610</td>
<td>-0.6374</td>
<td>-1.0028</td>
<td>-1.5395</td>
<td>-2.1129</td>
<td>-2.6721</td>
<td>-3.1843</td>
</tr>
</tbody>
</table>

(c) $(\log_{10})$ Maximum Error of the sigmoid function

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-3.7011</td>
<td>-3.9856</td>
<td>-5.1437</td>
<td>-5.6937</td>
<td>-5.7730</td>
<td>-5.7780</td>
<td>-5.7773</td>
<td>-5.7780</td>
</tr>
<tr>
<td>16</td>
<td>-1.2563</td>
<td>-1.6675</td>
<td>-2.0334</td>
<td>-2.7384</td>
<td>-3.9287</td>
<td>-5.1437</td>
<td>-6.3132</td>
<td>-7.4063</td>
</tr>
</tbody>
</table>

(d) $(\log_{10})$ Mean Squared Error of the sigmoid function
Error of $9.81 \cdot 10^{-7}$ and maximum error of $\pm 1.5 \cdot 10^{-3}$ when compared to the full-precision hyperbolic tangent function, and $5.13 \cdot 10^{-8}$ and maximum error of $\pm 9.07 \cdot 10^{-4}$ in sigmoid, which shows the best trade-off in precision and compute complexity, considering using a power-of-2 sized interval for simple index calculation (i.e., simple shift instead of division). Evaluation on the quantized RNN benchmarks shows no deterioration of the end-to-end-error when replacing the activation function with our proposed interpolation, which is not surprising as Neural Networks are known to be robust against noise. This extensions reduces the cycle count from 59.8 to 52.9 $k$ cycles within the LSTM networks [190,191], resulting in a $1.1 \times$ improvement. Tab. 4.6 gives a detailed overview of the instruction and cycle count for the two LSTM [190,191] networks in the RRM benchmark suite while exploiting the Xpulp extensions (SIMD, hardware loops, load-word increment), output feature map tiling. On the left without and on the right with the $\text{pl.tanh/pl.sig}$ extensions.

The $\text{pl.tanh}$ and $\text{pl.sig}$ have two arguments, the destination register $rD$ and the source register $rs1$, Tab. 4.5 shows the exact ISA encoding.
4.3. HW/SW EXTENSION AND OPTIMIZATIONS

Table 4.5. ISA declaration of the tangent hyperbolic and sigmoid extension.

<table>
<thead>
<tr>
<th>funct5</th>
<th>F</th>
<th>rs2</th>
<th>rs1</th>
<th>func3</th>
<th>rD</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1111</td>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>src1</td>
<td>000</td>
<td>dest</td>
</tr>
<tr>
<td>1 1111</td>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>src1</td>
<td>001</td>
<td>dest</td>
</tr>
</tbody>
</table>

Table 4.7. Assembly Code comparison with FM tiling only and with the `pl.sdotsp.h` instruction

```
1:       pl.sdotsp.h.0 r0, rA0, r0       pl.sdotp.0 r0, rA0, r0
2:       pl.sdotsp.h.1 r0, rA1, r0       pl.sdotp.1 r0, rA1, r0
3:       lw rB, Imm(rBAddr!)            lw rB, Imm(rBAddr!)
4:       // bubble rB dependency        lw rB0, Imm(rB0Addr!)
5:       lw rA0, Imm(rAAddr0!)          pl.sdotsp.h.0 rD0, rA2, rB
6:       lw rA1, Imm(rAAddr1!)          pl.sdotsp.h.0 rD0, rA2, rB
7:       lw rA2, Imm(rAAddr2!)          pl.sdotsp.h.1 rD1, rA3, rB
8:       lw rA3, Imm(rAAddr3!)          pl.sdotsp.h.0 rD2, rA0, rB
9:       pv.sdotsp.h rD0, rA0, rB        pl.sdotsp.h.1 rD3, rA1, rB
10:      pv.sdotsp.h rD1, rA1, rB        //}
```

Fig. 4.4 shows the RI5CY core with the extended datapath of the `pl.sdotsp.h` instruction with the changes highlighted in colors and active data paths in bold for the `pl.sdotsp.h` instruction. rA contains the memory address, which is loaded from memory by the load/store
<table>
<thead>
<tr>
<th></th>
<th>without tanh/sig ext</th>
<th>with tanh/sig ext</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instr.</strong></td>
<td><strong>cycles</strong></td>
<td><strong>instrs</strong></td>
</tr>
<tr>
<td>p.lw!</td>
<td>16'418</td>
<td>16'166</td>
</tr>
<tr>
<td>pv.sdotp</td>
<td>14'530</td>
<td>14'530</td>
</tr>
<tr>
<td>add</td>
<td>3'234</td>
<td>3'226</td>
</tr>
<tr>
<td>lw</td>
<td>2'854</td>
<td>2'854</td>
</tr>
<tr>
<td>addi</td>
<td>2'451</td>
<td>2'451</td>
</tr>
<tr>
<td>lh</td>
<td>1'887</td>
<td>1'887</td>
</tr>
<tr>
<td>srai</td>
<td>1'524</td>
<td>1'524</td>
</tr>
<tr>
<td>sw</td>
<td>1'488</td>
<td>1'488</td>
</tr>
<tr>
<td>blt</td>
<td>1'349</td>
<td>891</td>
</tr>
<tr>
<td>sh</td>
<td>1'235</td>
<td>1'235</td>
</tr>
<tr>
<td>bltu</td>
<td>1'143</td>
<td>583</td>
</tr>
<tr>
<td>slli</td>
<td>1'139</td>
<td>1'139</td>
</tr>
<tr>
<td>mul</td>
<td>822</td>
<td>822</td>
</tr>
<tr>
<td>jal</td>
<td>292</td>
<td>146</td>
</tr>
<tr>
<td>rem</td>
<td>192</td>
<td>27</td>
</tr>
<tr>
<td>beq</td>
<td>178</td>
<td>136</td>
</tr>
<tr>
<td>lui</td>
<td>140</td>
<td>140</td>
</tr>
<tr>
<td>lp.setup</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>jalr</td>
<td>76</td>
<td>38</td>
</tr>
<tr>
<td>ori</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>bne</td>
<td>31</td>
<td>29</td>
</tr>
<tr>
<td>andi</td>
<td>30</td>
<td>26</td>
</tr>
<tr>
<td>div</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>xori</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**sum**  51’174  49’499  **sum**  44’546  43’219

Table 4.6. Instruction Statistic for LSTM networks with SIMD, HW, lw-incr, FM tiling. On the left without and on the right with the tanh-sigmoid extension.
4.4. CORE IMPLEMENTATION RESULTS

The extended RI5CY core was implemented in Globalfoundries 22 nm FDX technology using an 8-track low-threshold (LVT) standard cell library and has been synthesized with Synopsys Design Compiler 18.06, back-end flow has been done with Cadence Innovus 18.11 and power
estimates are obtained by running gate-level simulations using Modelsim Questa v2019.1 with back-annotated delays from the final layout. Fig. 4.7 shows the area break-down of the RI5CY core without and with the new extensions. When compared to a standard RI5CY core (RV32-IMCXpulp), the new instructions result in very small circuit area overhead of 2.3 kGE (or 3.4% of the core area). Furthermore, the critical path of the core remains unchanged (between load-store unit and memory in the write-back stage) and the core operates at 380 MHz at 0.65 V at typical conditions at room temperature.

Where the enhanced core excels in energy efficiency. When compared in the same core performing the RISC-V standard RV32-IMC instructions, when executing relevant RNN benchmarks, the enhanced core is on average 15× faster. It performs 566 MMAC/s (instead of 21 MMAC/s). The power breakdown of the original RISC-Y core and this work is presented in Tab. 4.8. The extensions have insignificant impact on the power consumption while running the same code (i.e., with Xpulp, SIMD extensions and output feature map tiling), but when the core is using the extensions, the power consumption rises from 1.73 mW to 2.61 mW (51% total increase). While the decoder contributes insignificantly more power (≈5 µW), the higher power consumption is mainly due to the higher utilization of the compute units (ALU and MAC unit, i.e., 0.57 mW/33% of the total power), the increased GPR usage (0.16 mW/9%), and the higher use of the
### 4.4. Core Implementation Results

<table>
<thead>
<tr>
<th>Core</th>
<th>RISC-Y</th>
<th>RISC-RNN</th>
<th>RISC-RNN</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xpulp, SIMD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Output FM Tiling</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>sdotsp &amp; tanh/sig</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage</th>
<th>RISC-Y</th>
<th>RISC-RNN</th>
<th>RISC-RNN</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex-stage_i</td>
<td>0.46</td>
<td>0.47</td>
<td>1.04</td>
<td>0.57</td>
</tr>
<tr>
<td>➔/mult_i</td>
<td>0.34</td>
<td>0.34</td>
<td>0.73</td>
<td>0.39</td>
</tr>
<tr>
<td>➔/alu_i</td>
<td>0.06</td>
<td>0.06</td>
<td>0.14</td>
<td>0.08</td>
</tr>
<tr>
<td>id-stage_i</td>
<td>0.67</td>
<td>0.69</td>
<td>0.85</td>
<td>0.17</td>
</tr>
<tr>
<td>➔/registers_i</td>
<td>0.34</td>
<td>0.35</td>
<td>0.41</td>
<td>0.06</td>
</tr>
<tr>
<td>➔/datapath</td>
<td>0.31</td>
<td>0.31</td>
<td>0.41</td>
<td>0.10</td>
</tr>
<tr>
<td>➔/decoder_i</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>if-stage_i</td>
<td>0.31</td>
<td>0.29</td>
<td>0.35</td>
<td>0.04</td>
</tr>
<tr>
<td>load_store_unit_i</td>
<td>0.07</td>
<td>0.07</td>
<td>0.13</td>
<td>0.05</td>
</tr>
<tr>
<td>other</td>
<td>0.22</td>
<td>0.22</td>
<td>0.24</td>
<td>0.02</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1.73</td>
<td>1.74</td>
<td>2.61</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Table 4.8. Power breakdown @480MHz/0.65V/TT25C of a) original RISC-Y core with Xpulp, OFM, b) enhanced RNN core with Xpulp, OFM, w/o using new HW extensions c) like b), but using sdotsp, tanh, and sig extensions.
load-store unit (0.05 mW/3%). However, the overall energy efficiency at 218 GMAC/s/W shows a 10× improvement.

4.5 Conclusion

We presented the first RISC-V core design optimized for RRM applications using machine learning approaches based on RNNs. The core achieves order-of-magnitude performance (15×) and energy efficiency (10×) improvements over the baseline RISC-V ISA on a wide range of RNN flavors used in RRM. These results are obtained thanks to a synergistic combination of software and hardware optimizations, which only marginally increase area cost and do not affect operating frequency. It is essential to notice that the proposed optimization to not impact numerical precision, hence labor-intensive quantization-aware retraining is not needed. The enhanced RISC-V core achieves 566 MMAC/s and 218 GMAC/s/W (on 16-bit data types) in 22 nm FDX technology at 0.65 V, thereby providing a fully programmable and efficient open-source IP for future systems-on-chip for 5G Radio Resource Management.
Chapter 5

YodaNN: BWN HW Acceleration

Neural Networks implementations on out-of-the-shelf embedded platforms, and on an application-specific ISA processor have been shown in the previous chapters. Still, these platforms do not meet the requirements for Edge IoT for state-of-the-art CNNs. We are, therefore focusing on full-custom accelerators for convolution neural networks in the following chapters. Recent developments in CNN accelerators for system-on-chip integration have reduced energy consumption significantly. Unfortunately, even these highly optimized devices are above the power envelope imposed by mobile and deeply embedded applications and face hard limitations caused by CNN weight I/O and storage. This prevents the adoption of novel CNNs in future ultra-low power Internet of Things end-nodes for near-sensor analytics. Recent algorithmic and theoretical advancements enable competitive classification accuracy even when quantizing neural networks to a single bit. Binary-Weight Neural Networks (BWNs) binarize the weights, and BNNs binarize both weights and activations. These new findings bring major optimization opportunities in the arithmetic core by removing the need for expensive multiplications, as well as reducing I/O bandwidth and storage.
### Table 5.1. Core- and Device-Level Efficiencies of BWN/BNN Accelerators presented in Chpt. 5-7

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>YodaNN (Chpt. 5)</td>
<td>BWN</td>
<td>65 nm</td>
<td>61.2</td>
<td>2.7</td>
</tr>
<tr>
<td>YodaNN (Chpt. 5)</td>
<td>BWN</td>
<td>22 nm(^a)</td>
<td>149.1</td>
<td>2.8</td>
</tr>
<tr>
<td>Hyperdrive (Chpt. 7)</td>
<td>BWN</td>
<td>22 nm</td>
<td>4.9</td>
<td>4.3</td>
</tr>
<tr>
<td>XNORBIN (Chpt. 6)</td>
<td>BNN</td>
<td>22 nm</td>
<td>204.9</td>
<td>26.9</td>
</tr>
</tbody>
</table>

\(^a\)Technology scaled based on Dreslinksi et al. [1].

In this chapter and the following chapters 6 and 7, we are introducing three accelerators for highly-quantized neural networks (i.e., Binary-Weight and Fully-Binary Neural Networks) whereas Tab. 5.1 gives an overview of core- and device-level efficiencies. The first chapter introduces the first BWN accelerator YodaNN\(^1\) achieving a core energy efficiency of 61.2 TOP/s/W (i.e., 7×7 kernels) or 149 TOP/s/W (i.e., scaled to 22 nm technology), followed by XNORBIN accelerating BNNs in Chpt. 6 reaching 205 TOP/s/W. As both accelerators are dominated by I/O energy costs on device-level (e.g., 2.8 TOP/s/W in YodaNN), and further reduced energy efficiencies when running novel CNNs with small-sized kernels (i.e., 1×1 or 3×3), we introduce Hyperdrive in Chpt. 7. Hyperdrive focuses on significantly reducing the I/O requirements, and is flexible to a large set of neural networks and scalable to any large-scale problems while scaling up units on-chip or exploiting a systolic array of chips.

The remainder of this chapter is organized as follows: Sec. 5.1 motivates more in detail the use of custom accelerators for neural networks inference and using highly-quantized neural networks. Sec. 5.2 summarizes the state-of-the-art approach of network design and hardware acceleration and introduces to Binary Weight Neural Networks. Sec. 5.3 introduces in the architecture design of YodaNN, followed

---

\(^1\)YodaNN named after the Jedi master known from StarWars – “Small in size but wise and powerful” [224]
5.1 Introduction

Even though, optimized software implementations on out-of-the-shelf compute platforms (presented in previous chapters and in current research [44,45,47]) reduce significantly the energy requirements for neural networks inference; They are still not able to fulfill the power constraints imposed by mobile and Internet of Things (IoT) end-node devices. The common approach of sourcing out all CNN computation from IoT end-nodes to data servers is exceptionally challenging and power-consuming, due to the large communication bandwidth required to transmit the data streams. This prompts the need for specialized architectures to achieve higher performance at lower power within the end-nodes of the IoT.

A few research groups exploited the customization paradigm by designing highly specialized hardware to enable CNN computation in the domain of embedded applications. Several approaches leverage FPGAs to maintain post-fabrication programmability while providing a significant boost in terms of performance and energy efficiency [54]. However, FPGAs are still two orders of magnitude less energy-efficient than ASICs [208]. Moreover, CNNs are based on a very reduced set of computational kernels (i.e., convolution, activation, pooling), but they can be used to cover several application domains (e.g., audio, video, biosignals) by simply changing weights and network topology, relaxing the issues with non-recurring engineering which are typical in ASIC design.

The AI HW accelerator research community has been developing specialized hardware architectures focusing on data re-use with limited resources and optimizing arithmetic precision [58,225], exploiting weight and feature map (FM) sparsity [68], and performing on-the-fly data compression to ultimately maximize energy efficiency [226,227].
Among CNN ASIC implementations, the precision of arithmetic operands plays a crucial role in energy efficiency. Several reduced-precision implementations have been proposed recently, relying on 16-bit, 12-bit, or 10-bit of accuracy for both operands and weights [58, 62, 208, 228, 229], exploiting the intrinsic resiliency of CNNs to quantization and approximation [225, 230]. Exploiting weight and feature map (FM) sparsity [68], and performing on-the-fly data compression to ultimately maximize energy efficiency [226, 227] reaching up to 5 TOp/s/W.

Recently, several methods to train neural networks to withstand extreme quantization have been proposed, yielding the notions of binary- and ternary-weight networks (BWNs, TWNs) and binarized neural networks (BNNs) [231–233]. BWNs and TWNs allow a massive reduction of the data volume to store the network and have been applied to recent and high-complexity networks with an almost negligible loss. In YodaNN and Hyperdrive (cf. Chpt. 7), we are exploiting BWNs, which quantize the weights to the extreme of two values -1 and 1. The binarization is done in the forward path, while the gradients are updated in high-precision to guarantee stability and convergence during training. This approach has the potential to bring great benefits to CNN hardware implementation by enabling the replacement of multipliers with much simpler complement operations and multiplexers, and by drastically reducing weight storage requirements. Interestingly, binary-weight networks lead to only small accuracy losses on several well-known CNN benchmarks [166, 234]. In XNORBIN (cf. Chpt. 6), we are going even further and quantize also the activations, which reduces not just the feature map memory footprint to a single bit per data item (e.g., a pixel), but also replaces multiplications with simple XNOR operations.

In this chapter, we present YodaNN the first optimized hardware design implementing a flexible, energy-efficient and performance scalable convolutional accelerator supporting binary-weight CNNs. We demonstrate that this approach improves the energy efficiency of the digital core of the accelerator by 5.1×, and the throughput by 1.3×, with respect to a baseline architecture based on 12-bit MAC units operating at a nominal supply voltage of 1.2 V. To extend the performance scalability of the device, we implement a latch-based Standard Cell Memory (SCM) architecture for on-chip
5.2. RELATED WORK

5.2.1 Co-Design of DNN Models and Hardware

Over the last few years, several approaches adapting DNNs to reduce the computational demand have been presented. One main direction was the reduction of the number of operations and model size. Specifically, the introduction of sparsity provides an opportunity to skip some operations. By pruning the weights a high sparsity can be achieved particularly for the fully-connected layers found at the end of many networks and the ReLU activations in most DNN models injects sparsity into the FMs, which can be exploited [68,69].

A different direction is the research into reduced precision computation. Standard fixed-point approaches work down to 10-16 bit number formats for many networks. It is possible to further reduce the precision to 8 bit with small accuracy losses (< 1%) when retraining the network to adapt to this quantization [230]. There are limitations to this:

1. for deeper networks higher accuracy losses (2-3% for GoogLeNet) remain, and
2. Typically, only the inputs to the convolutions are quantized in this format. Internal computations are performed at full precision, which implies that the internal precision is very high for large networks, e.g., for a $3 \times 3$ convolution layer with 512 input FMs, this adds 12 bits.

Considering the sparsity in deeper layers because of the ReLU activation function, multiplications with zeros can be skipped, reducing run time and saving energy. Moons et al. showed a power reduction of $30 \times$ without accuracy loss, or $225 \times$ with a 1% increase in error by skipping zero-multiplications and quantizing layers independently [236]. Further approaches include non-linearly spaced quantization in the form of mini-floats [230], and power-of-two quantization levels replacing multiplications with bit-shift operations (i.e., INQ [231]).

### 5.2.2 CNN Acceleration Hardware

There are several approaches to perform CNN computations on GPUs, which are able to reach a throughput up to 6 TOp/s with a power consumption of 250 W [47, 237]. On the other hand, there is a clear demand for low-power CNN acceleration. For example, Google exploits in its data-centers a custom-made neural network accelerator called Tensor Processing Unit (TPU) tailored to their TensorFlow framework. Google claims that they were able to reduce power by roughly $10 \times$ with respect to GP-GPUs [238]. Specialized functional units are also beneficial for low-power programmable accelerators which recently entered the market. A known example is the Movidius Myriad 2 which computes 100 GFLOPS and needs just 500 mW@600 MHz [239]. However, these low-power architectures are still significantly above the energy budget of IoT end-nodes. Therefore, several dedicated hardware architectures have been proposed to improve energy efficiency while preserving performance, at the cost of flexibility.

Several CNN systems were presented implementing the activation layer (mainly ReLU) and pooling (i.e., max-pooling) [56, 57, 63]. In this work, we focus on the convolution layer as this contributes most to the computational complexity [47]. Since convolutions typically rely on recent data for the majority of computations, sliding window schemes are typically used [57, 58, 62, 65] (e.g., in case of $7 \times 7$ kernels,
5.2. RELATED WORK

6×7 pixels are reused in the subsequent step). In this work, we go even further and cache the values, such that we can reuse them when we switch from one to the next tile. In this way, only one pixel per cycle has to be loaded from the off-chip storage.

As the filter kernel sizes change from problem to problem, several approaches were proposed to support more than one fixed kernel size. Zero-padding is one possibility: in Neuflow the filter kernel was fixed to 9 × 9, and it was filled with zeros for smaller filters [59]. However, this means that for smaller filters, unnecessary data has to be loaded and that the unused hardware cannot be switched off. Another approach was presented by Chen et al., who have proposed an accelerator containing an array of 14 × 12 configurable processing elements connected through a network-on-chip. The PEs can be adjusted for several filter sizes. For small filter sizes, they can be used to calculate several output channels in parallel, or they can be switched-off [65]. Even though this approach brings flexibility, all data packets have to be labeled, such that the data can be reassembled in a later step. Hence, this system requires a lot of additional multiplexers and control logic, forming a bottleneck for energy efficiency.

Another approach minimizes the on-chip computational complexity exploiting the fact that due to the ReLU activation layer, zero-values appear quite often in CNNs. In this way, some of the multiplications can be bypassed by means of zero-skipping [65]. This approach is also exploited by Reagon et al. [60] and Albericio et al. [61]. Another approach exploits that the weights’ distribution shows a clear maximum around zero. Jaehyeong et al. proposed a small 16-bit multiplier, which triggers a stall and calculation of the higher-order bits only when an overflow is detected, which gives an improvement of 56% in energy efficiency [57]. The complexity can be reduced further by implementing quantization scaling as described in Section 5.2.1. Even though most approaches work with fixed-point operations, the number of quantization bits is still kept at 24-bit [56,57], 16-bit [59,63,64], or 8-bit [58,62,65,208]. The peak compute energy efficiency for fixed-point CNN accelerators with precision bigger than 8 bit can be found at around 50 GOp/s/W for FPGAs, and 2 TOp/s/W in 65 nm.

Many of the sparsity-based optimizations mentioned in Sec. 5.2.1 have been implemented in hardware accelerators [68,70] and achieve
an up to 3× higher core energy efficiency and raise the device-level energy efficiency by around 70% through data compression. To improve throughput and energy efficiency, Hang et al. present compressed deep neural networks, where the number of different weights is limited, and instead of saving or transmitting full precision weights, the related indices are used [41]. They present a neural networks accelerator, called Efficient Inference Engine (EIE), exploiting network pruning, and weight sharing (deep compression). For a network with a sparsity as high as 97%, EIE reaches an energy efficiency of 5 TOp/s/W, and a throughput of 100 GOp/s, which is equal to a throughput of 3 TOp/W for the equivalent non-compressed network [68]. Even though this outperforms the previous state-of-the-art by 5×, we can still demonstrate a 12× more efficient design exploiting binary weights. Jaehyeong et al. used PCA to reduce the dimension of the kernels. Indeed, they showed that there is a strong correlation among the kernels, which can be exploited to reduce their dimensionality without major influence on accuracy [57]. This actually reduces the energy needed to load the chip with the filters and reduces the area to save the weights, since only a small number of bases and a reduced number of weight components need to be transmitted. On the other hand, it also increases the core power consumption, since the weights have to be reconstructed on-the-fly. With binary weights, we were able to reduce the total kernel data by 12×, which is similar to the 12.5× reported in Jaehyeong et al. [57]. On the other hand, YodaNN outperforms their architecture by 43× in terms of energy efficiency thanks to its simpler internal architecture that do not require on-the-fly reconstruction.

Some CNN accelerators have been presented exploiting analog computation: in one approach [66], part of the computation was performed partially on the camera sensor chip before transmitting the data to the digital processing chip. Another mixed-signal approach [240] looked into embedding part of the CNN computation in a memristive crossbar. Efficiencies of 960 GOp/s [66] and 380 GOp/s/W [67] were achieved.

### 5.2.3 Binary Weight Neural Networks

In Section 3.3.3, we have already presented BNNs, where both activations and weights are binarized to a value of -1 or 1. Unfortunately,
5.3. ARCHITECTURE

The performance gap between BNNs and their corresponding baseline networks is still significant with 12.3% (excluding methods adapting network topology), as presented in Tab. 3.1. Binary Weight Neural Networks or Ternary Weight Neural Networks quantize just the weights to a binary (+1/-1) or ternary (+1/0/-1) value while still computing the FMs in high-precision (e.g., FP32 or INT8). This massively compresses the data volume of the weights and has even been shown to be applicable to deep networks with an accuracy loss of approximately 0.3% for ResNet-50 [241] and thus less than the fixed point-and-retrain strategies.

BinaryConnect proposes to binarize (−1, +1) the weights \( w_{fp} \). During training of BWNs, the weights are stored as so-called shadow weights \( w_{fp} \) and updated in full precision, but binarized for forward propagation \( w_b \) [234]. The following formula shows the deterministic and stochastic binarization function, where a "hard sigmoid" function \( \sigma \) is used to determine the probability distribution:

\[
\begin{align*}
  w_{b,det} &= \begin{cases} 
    1, & \text{if } w_{fp} < 0 \\
    -1, & \text{if } w_{fp} > 0
  \end{cases}, \\
  w_{b,sto} &= \begin{cases} 
    1, & p = \sigma(w_{fp}) \\
    -1, & p = 1 - \sigma
  \end{cases}
\end{align*}

\[
\sigma(x) = \text{clip}\left(\frac{x + 1}{2}, 0, 1\right) = \max(0, \min(1, \frac{x + 1}{2}))
\]

Tab. 5.2 shows an overview of state-of-the-art BWNs (similar to Tab. 3.1 for BNNs). It can be seen, that training of these networks can reach network performance near to their full-precision baseline. Huang et al. introduced a new activation function (i.e., clamping rectified linear unit CReLU) and achieved 76.6% with ResNet-50 on ImageNet which is just 0.3% below its baseline.

5.3 Architecture

A CNN consists of several layers, usually they are convolution, activation, pooling or batch normalization layers. In this work, we focus on the convolution layers as they make up for the largest share of the total computation time. As can be seen in Fig. 5.1 from [47], convolution layers make up for the largest fraction of compute time in
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Zhu2016 [242]</td>
<td>AlexNet</td>
<td>1</td>
<td>32</td>
<td>69.3</td>
<td>69.6</td>
<td>98.8</td>
<td>90.8</td>
<td>83.0</td>
<td>79.7</td>
<td>88.3</td>
<td>66.0</td>
</tr>
<tr>
<td>Huang2019 [241]</td>
<td>ResNet-50</td>
<td>1</td>
<td>32</td>
<td>76.6</td>
<td>76.9</td>
<td>93.1</td>
<td>93.3</td>
<td>72.0</td>
<td>77.3</td>
<td>84.1</td>
<td>64.1</td>
</tr>
<tr>
<td>Huang2019 [241]</td>
<td>ResNet-18</td>
<td>2</td>
<td>32</td>
<td>67.5</td>
<td>67.8</td>
<td>89.0</td>
<td>89.2</td>
<td>75.0</td>
<td>75.0</td>
<td>89.0</td>
<td>75.0</td>
</tr>
<tr>
<td>Zhou2016 [168]</td>
<td>AlexNet</td>
<td>1</td>
<td>32</td>
<td>55.9</td>
<td>56.0</td>
<td>53.0</td>
<td>53.0</td>
<td>53.0</td>
<td>53.0</td>
<td>53.0</td>
<td>53.0</td>
</tr>
<tr>
<td>Li2016 [244]</td>
<td>ResNet-18</td>
<td>2</td>
<td>32</td>
<td>65.4</td>
<td>65.6</td>
<td>86.8</td>
<td>87.0</td>
<td>72.0</td>
<td>72.0</td>
<td>86.8</td>
<td>72.0</td>
</tr>
<tr>
<td>Hu2018 [245]</td>
<td>ResNet-18</td>
<td>1</td>
<td>32</td>
<td>69.3</td>
<td>69.6</td>
<td>89.2</td>
<td>89.4</td>
<td>75.0</td>
<td>75.2</td>
<td>89.0</td>
<td>75.0</td>
</tr>
<tr>
<td>Rastegari2016 [79]</td>
<td>GoogLeNet</td>
<td>1</td>
<td>32</td>
<td>71.3</td>
<td>72.0</td>
<td>90.0</td>
<td>90.1</td>
<td>72.0</td>
<td>72.0</td>
<td>90.0</td>
<td>72.0</td>
</tr>
<tr>
<td>Rastegari2016 [79]</td>
<td>ResNet-18</td>
<td>1</td>
<td>32</td>
<td>69.3</td>
<td>69.6</td>
<td>89.2</td>
<td>89.4</td>
<td>75.0</td>
<td>75.2</td>
<td>89.0</td>
<td>75.0</td>
</tr>
</tbody>
</table>
5.3. ARCHITECTURE

Figure 5.1: Overview of execution time in a convolution neural network for scene labeling executed on CPU and GPU [47]

Figure 5.2: A $32 \times 32$ CNN layer, with input channels $i_n$ and output channels $o_k$. 
CHAPTER 5. YODANN: BWN HW ACCELERATION

CPU and GPU implementations. A general convolution layer is drawn in Fig. 5.2 and it is described by Equation (5.2).

\[
o_k = C_k + \sum_{n \in I} i_n \cdot w_{k,n} \text{ (5.1)}
\]

\[
o_k(x, y) = C_k + \sum_{n \in I} \left( \sum_{a=0}^{b_k-1} \sum_{b=0}^{h_k-1} i_n (x + a, y + b) \cdot w_{k,n}(a, b) \right) \text{ (5.2)}
\]

A layer consists of \( n_{in} \) input channels, \( n_{out} \) output channels, and \( n_{in} \cdot n_{out} \) kernels with \( h_k \times b_k \) weights; we denote the matrix of filter weights as \( w_{k,n} \). For each output channel \( k \) every input channel \( n \) is convolved with a different kernel \( w_{k,n} \), resulting in the terms \( \tilde{o}_{k,n} \), which are accumulated to the final output channel \( o_k \).

We propose a hardware architecture able to calculate \( n_{ch} \times n_{ch} \) channels in parallel. If the number of input channels \( n_{in} \) is greater than \( n_{ch} \), the system has to process the network \( \lceil n_{in}/n_{ch} \rceil \times n_{in}/n_{ch} \) times and the results are accumulated off-chip. This adds only \( \lceil n_{in}/n_{ch} \rceil - 1 \) operations per pixel. In the following, we fix, for ease of illustration, the number of output channels to \( n_{ch} = 32 \) and the filter kernel size to \( h_k = b_k = 7 \). The system is composed of the following units (an overview can be seen in Fig. 5.3):

- The **Filter Bank** is a shift register which contains the binary filter weights \( w_{k,n} \) for the output channels \( k \in \mathbb{N}_{<32} \) and input channels \( n \in \mathbb{N}_{<32} \) \( (n_{in} \cdot n_{out} \cdot h_k^2 \cdot 1 \text{ bit} = 6.4 \text{ kB}) \) and supports column-wise left circular shift per kernel.

- The **Image Memory** saves an image stripe of \( b_k = 7 \) width and 1024 height (10.8 kB), which can be used to cache \( 1024/n_{in} = 1024/32 = 32 \) rows per input channel.

- The **Image Bank** (ImgBank) caches a spatial window of \( h_k \times b_k = 7 \times 7 \) per input channel \( n \) (2.4 kB), which are applied to the SoP units. This unit is used to reduce memory accesses, as the \( h_k - 1 = 6 \) last rows can be reused when we proceed in a
column-wise order through the input images. Only the lowest row has to be loaded from the image memory and the upper rows are shifted one row up.

- **Sum-of-Product (SoP) Units** (32, 1 per output channel): For every output channel $k$, the SoP unit $k$ calculates the sum terms $\tilde{o}_{k,n}$, where in each cycle the contribution of a new input channel $n$ is calculated.

- **Channel Summer (ChSum) Units** (32, 1 per output channel): The Channel Summer $k$ accumulates the sum terms $\tilde{o}_{k,n}$ for all input channels $n$.

- **1 Scale-Bias Unit**: After all the contributions of the input channels are summed together by the channel summers, this unit starts to scale and bias the output channels in an interleaved manner and streams them out.

- **I/O Interface**: Manages the 12-bit input stream (input channels) and the two 12-bit output streams (output channels) with a protocol based on a blocking ready-valid handshaking.
After accumulation:

Figure 5.4: Timing diagram of the operating scheme: Input Stream, SoP k’s operations, output stream after accumulation.


5.3. ARCHITECTURE

5.3.1 Dataflow

The pseudo-code in Algorithm 1 gives an overview of the main steps required for the processing of convolution layers, while Fig. 5.4 shows a timing diagram of the parallel working units. The input and output channels need to be split into blocks smaller than $32 \times 32$, while the image is split into slices of $1024/c_{in}$ height (lines 1–3). These blocks are indicated as *YodaNN chip block*. Depending on whether the border is zero-padded or not, $\lfloor (h_k - 1)/2 \rfloor$ or $h_k - 1$ columns need to be preloaded (just in case of $1 \times 1$ filters no pixels need to be preloaded) (Line 6). The same number of pixels are preloaded from one subsequent column, such that a full square of $h_k^2$ pixels for each input channel is available in the image bank (Line 7). After this preloading step, the SoPs start to calculate the partial sums of all 32 output channels while the input channel is changed every cycle (lines 15–20). When the final input channel is reached, the channel summers keep the final sum for all 32 output channels of the current row and column, which are scaled and biased by the Scale-Bias Unit, and the final results are streamed out in an interleaved manner (lines 27–33). In case of $n_{out} = n_{in}$ (e.g., $32 \times 32$) the same number of cycles are needed to stream out the pixels for all output channels as cycles are needed to sum all input channels for the next row, which means that all computational units of the chip are fully-utilized. Each row is processed sequentially, then the system switches to the next column, where again, the first pixels of the column are preloaded. The filters are circularly right-shifted to be aligned to the correct columns. Then, the next column of all output channels are calculated. This procedure is repeated until the whole image and blocks of input and output channels have been processed. Finally, the partial sums for each output channel need to be summed together for every block of input channels. (Line 37).

We are using a sliding window approach which is illustrated in Fig. 5.5 [47]. To avoid shifting all images in the image memory to the left for the next column, the rightmost pixels are inserted at the position of the obsolete pixel, and the weights are shifted instead. To illustrate this, Equation (5.3) shows the partial convolution for one pixel while the pixels are aligned to the actual column order and Equation (5.4) shows it when the next column is processed and the
Algorithm 4 Dataflow Pseudo-Code

Require: weights $w_{k,n}$, input feature map $i_k(x,y)$
Ensure: $o_n = \sum_k i_k * w_{k,n}$

1: for all $y_{block} \in \{1,..,\lceil h_{im}/h_{max} \rceil \}$ do
2:   for all $c_{out,block} \in \{1,..,\lceil n_{out}/n_{ch} \rceil \}$ do
3:     for all $c_{in,block} \in \{1,..,\lceil n_{in}/n_{ch} \rceil \}$ do
4:       YodaNN chip block
5:       Load Filters $w_{k,n}$
6:       Load $m$ columns, where
7:       $m = \begin{cases} 
        h_k - 1, & \text{if not zero-padded} \\
        \lceil \frac{h_k}{2} - 1 \rceil, & \text{if zero-padded}
        \end{cases}$
8:     Load $m$ pixels of the $(m+1)^{th}$ column.
9:     Parallel block 1
10:    for all $x$ do
11:      for all $y$ do
12:        $\tilde{o}(c_{out} := \cdot, x, y) := 0$
13:      for all $c_{in}$ do
14:        Single cycle block
15:        for all $(a,b) \in \{-\lceil \frac{h_k}{2} \rceil \leq a, b \leq \lceil \frac{h_k}{2} \rceil \}$ do
16:         $\tilde{o}_{c_{out}}(x,y) = \tilde{o}_{c_{out}}(x,y) + i_{c_{in}}(x+a, y+b) * w_{c_{out},c_{in}}(a,b)$
17:       end for
18:     end for
19:     Parallel block 2
20:    for all $x$ do
21:      wait until $\tilde{o}_0(x,0)$ is computed
22:    for all $y$ do
23:      Single cycle block
24:      for all $c_{out}$ do
25:        $o_{c_{out}}(x,y) = \alpha_{c_{out}} \tilde{o}_{c_{out}}(x,y) + \beta_{c_{out}}$
26:      output $o_{c_{out}}(x,y)$
27:    end for
28:  end for
29:  Sum the input channel blocks:
30: $o_{n,\text{final}} = \sum_{c_{in,blocks}} o_{n,\cdot}$
31: end for
32: end for
weights need to be aligned. To indicate the partial sum, the Frobenius inner product formalism is used, where: $\langle \mathbf{A}, \mathbf{B} \rangle_F = \sum_{i,j} a_{ij} b_{ij}$.

$$\bar{o}(2, 2) = \left\langle \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}, \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \right\rangle_F$$ (5.3)

$$\bar{o}(3, 2) = \left\langle \begin{bmatrix} x_{14} & x_{12} & x_{13} \\ x_{24} & x_{22} & x_{23} \\ x_{34} & x_{32} & x_{33} \end{bmatrix}, \begin{bmatrix} w_{13} & w_{11} & w_{12} \\ w_{23} & w_{21} & w_{22} \\ w_{33} & w_{31} & w_{32} \end{bmatrix} \right\rangle_F$$ (5.4)

Equation 5.4 shows the operands as they are applied to the SoP units. The 4th column which should be the most-right column is in the first column and also the other columns are shifted to the right, thus the weights also needs to be shifted to the right to obtain the correct result. The permutation in algebraic form is formulated in Equation (5.5):

$$\bar{o}(3, 2) = \left\langle \begin{bmatrix} x_{14} & x_{12} & x_{13} \\ x_{24} & x_{22} & x_{23} \\ x_{34} & x_{32} & x_{33} \end{bmatrix}, \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \right\rangle_F \cdot P$$ (5.5)

where $P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$ is the permutation matrix

5.3.2 BinaryConnect Approach

In this work, we present a CNN accelerator based on Binary-Weight Neural Networks [233]. With respect to an equivalent 12-bit version, the first major change in architecture are the weights which are reduced to a binary value $w_{k,n} \in \{-1, 1\}$ and remapped by the following equation:

$$f: \{-1, 1\} \rightarrow \{0, 1\}, \ y \mapsto \begin{cases} 0 & \text{if } z = -1 \\ 1 & \text{if } z = 1 \end{cases}$$ (5.6)
Figure 5.5: Sliding window approach of the image memory [47].

The size of the filter bank decreases thus from $n_{ch}^2 \cdot h_k^2 \cdot 12 = 37'632$ bit to $n_{ch}^2 \cdot h_k^2 \cdot 1 = 3'136$ bit in case of the 12-bit MAC architecture with $8 \times 8$ channels and $7 \times 7$ filters that we consider as baseline. The $12 \times 12$-bit multipliers can be substituted by two’s-complement operations and multiplexers, which reduce the "multiplier" and the adder tree size, as the products have a width of 12 bit instead of 24. The SoP is fed by a 12-bit and $7 \times 7$ pixel sized image window and $7 \times 7$ binary weights. Figure 5.6 shows the impact on area while moving from 12-bit MACs to the binary connect architectures. Considering that with the 12-bit MAC implementation, 40% of the total chip area is used for the filter bank, and another 40% is needed for the $12 \times 12$-bit multipliers and the accumulating adder trees, this leads to a significant reduction in area cost and complexity. In fact, the area of the conventional SoP unit could be reduced by $5.3 \times$ and the filter bank by $14.9 \times$ when moving from the Q$_{2,9}$ to the binary version. The impact on the filter bank is straightforward as 12 times fewer bits need to be saved compared to the Q$_{2,9}$, but also the SoP shrinks, as the $12 \times 12$-bit multipliers are replaced with 2’s complement operation
5.3. **ARCHITECTURE**

units, and multiplexers and the adder tree needs to support a smaller dynamic range, thanks to the smaller products, since the critical path is reduced as well. It is possible to reduce voltage while still keeping the same operating frequency and thus improving energy efficiency even further.

![Area Breakdown for fixed-point and several binary architectures](image)

Figure 5.6: Area Breakdown for fixed-point and several binary architectures.

### 5.3.3 Latch-Based SCM

An effective approach to optimizing energy efficiency is to adapt the supply voltage of the architecture according to the performance requirements of the application. However, the potential of this approach is limited by the presence of SRAMs for implementation of image memory, which bounds the voltage scalability to 0.8 V (in 65nm CMOS technology). To overcome this limitation, we replace the SRAM-based image memory with a latch-based SCMs taking advantage of the area savings achieved through the adoption of binary SoPs.

Indeed, although SCMs are more expensive in terms of area (Figure 5.6), they are able to operate in the whole operating range of the technology (0.6 V - 1.2 V) and they also feature significantly smaller read/write energy [235] at the same voltage. To reduce the area overhead of the SCMs and improve routability, we propose a
multi-banked implementation, where the image memory consists of a latch array organized in $6 \times 8$ blocks of 128 rows of 12-bit values, as described in Fig 5.7. A pre-decoding logic, driven by the controller of the convolutional accelerator, addresses the proper bank of the array every cycle, generating the local write and read enable signals, the related address fields, and propagating the input pixels to the banks and the current pixels to the SoP unit. During a typical CNN execution, every cycle, 6 SCMs banks are read, and one is written, according to the image memory access pattern described in Fig 5.5.

The SCMs are designed with a hierarchical clock gating and address/data silencing mechanisms as shown in Figure 5.8, so that when a bank is not accessed the whole latch array consumes no dynamic power. Every SCM block consists of a 12 bit $\times$ 128 rows array of latches, a data-in write path, and a read-out path. To meet the requirements of the application, the SCM banks are implemented with a two-ported, single-cycle latency architecture with input data and read address sampling. The write path includes data-in sampling registers, and a two-level clock gating scheme for minimizing the dynamic power of the clock path to the storage latches. The array write enable port drives the global clock gating cell, while the row clock gating cells are driven by the write address one-hot decoder. The readout path is implemented with a read address register with clock gating driven by a read enable signal, and a static multiplexer tree, which provides robust and low power operation and enables dense and low congestion layout.

Thanks to this optimized architecture based on SCMs, only up to 7 out of 48 banks of SCM banks consume dynamic power in every cycle, reducing power consumption of the memory by $3.25 \times$ at 1.2 V with respect to a solution based on SRAMs [208], while extending the functional range of the whole convolutional engine down to 0.6 V which is the voltage limit of the standard cells in UMC 65nm technology chosen for implementation [240].

5.3.4 Considering I/O Power in Energy Efficiency

I/O power is a primary concern of convolutional accelerators, consuming even more than 30% of the overall chip power [208]. As we decrease
the computational complexity by the binary approach, the I/O power gets even more critical. Fortunately, if the number of output channels is increased, more operations can be executed on the same data, which reduces the needed bandwidth and pad power consumption. The other advantage with having more SoP units on-chip is throughput which is formulated in (5.7):

\[ \Theta = 2 \cdot (n_{filt}^2 \cdot n_{ch}) \cdot f \]  

(5.7)

With this in mind, we increased the number of input and output channels from 8 × 8 to 16 × 16 and 32 × 32, which provides an ideal speed-up of throughput by 2× and 4×, respectively.

### 5.3.5 Support for Different Filter Sizes, Zero-Padding, Scaling and Biasing

Adapting filter size to the problem provides an effective way to improve the flexibility and energy efficiency of the accelerator when executing CNNs with different requirements. Although the simplest approach
Figure 5.8: Block diagram of one SCM bank.
is to zero-pad the filters, this is not feasible in the presented binary connect architecture, as the value 0 is mapped to −1. A more energy-efficient approach tries to re-use parts of the architecture. We present an architecture where we re-use the binary multipliers for two $3 \times 3$, two $5 \times 5$ or one $7 \times 7$ filters. In this work we limit the number of output channels per SoP unit to two as we are limited in output bandwidth. With respect to our baseline architecture, supporting only $7 \times 7$ filters, the number of binary operators and the weights per filter is increased from 49 to 50, such that two $5 \times 5$ or one $7 \times 7$ filter fits into one SoP unit. In case a filter size of $3 \times 3$ or $5 \times 5$ is used, the image from the image bank is mapped to the first 25 input image pixels, and the latter 25 are finally accumulated in the adjusted adder tree, which is drawn in Fig. 5.9. With this scheme, $n_{ch} \times 2n_{ch}$ channels for $3 \times 3$ and $5 \times 5$ filters can be calculated, which improves the maximum bandwidth and energy efficiency for these two cases. The unused 2’s complement-and-multiplex operands (binary multipliers) and the related part of the adder tree are silenced and clock-gated to reduce switching, therefore keeping the power dissipation as low as possible.

To support also different kernel sizes, we provide the functionality to zero-pad the unused columns from the image memory and the rows from the image bank instead of zeroing the weights which does not make sense with binary weights. This allows us to support kernels of size $1 \times 1$, $2 \times 2$, $4 \times 4$ and $6 \times 6$ as well. The zero-padding is also used to add zeros to image borders: e.g., for a $7 \times 7$ convolution the first 3 columns and first 3 rows of the $4^{th}$ column is preloaded. The 3 columns right to the initial pixel and the 3 rows on top of the pixel are zeroed the same way as described before and thus have not to be loaded onto the chip.

Finally, the system supports channel scaling and biasing, which are common operations (e.g., in batch normalization layer) in neural networks, which can be calculated efficiently. As described in the previous section up to two output channels are calculated in parallel in every SoP unit. Therefore the SoP also saves two scaling and two biasing values for these different output channels. As the feature maps are kept in maximum precision on-chip, the channel summers’ output $Q_{7,9}$ fixed-point values, which are then multiplied with the $Q_{2,9}$ formatted scaling factor and added to the $Q_{2,9}$ bias and finally the
Figure 5.9: The adder tree in the SoP unit: Different colors are showing the data paths for $3 \times 3$, $5 \times 5$, $7 \times 7$ kernels are indicated. The operands of the unused adders are silenced, but not indicated in the figure.
5.4 Results

5.4.1 Computational Complexity and Energy Efficiency Measure

Research in the field of deep learning is done on a large variety of systems, such that platform-independent performance metrics are needed. For computational complexity analysis, the total number of multiplications and additions has been used in other publications [47, 59, 228, 246]. For a CNN layer with $n_{in}$ input channels and $n_{out}$ output channels, a filter kernel size of $h_{k} \times w_{k}$, and an input size of $h_{im} \times w_{im}$, the computational complexity to process one frame can be calculated as follows:

$$\#Op = 2n_{out} n_{in} h_{k} w_{k} (h_{in} - h_{k} + 1)(w_{in} - h_{k} + 1)$$

(5.8)

The factor of 2 considers additions and multiplications as separate arithmetic operations (Op), while the rest of the equation calculates the number of multiply-accumulate operations MACs. The two latter factors $(h_{in} - h_{k} + 1)$ and $(w_{in} - h_{k} + 1)$ are the height and width of the output channels including the reduction at the border in case no zero-padding was applied. Memory accesses are not counted as additional operations. The formula does not take into account the amount of operations executed when applying zero-padding. In the following evaluation, we will use the following metrics:

- Throughput $\Theta = (\#Op \text{ based on (5.8)})/t \ [\text{GOp/s}]$
- Peak Throughput: Theoretically reachable throughput. This does not take into account idling, cache misses, etc.
- Energy Efficiency $H_{E} = \Theta/P \ [\text{TOp/s/W}]$
- Area Efficiency $H_{A} = \Theta/A \ [\text{GOp/s/MGE}]$
Furthermore, we introduce some efficiency metrics to allow for realistic performance estimates, as CNN layers have varying numbers of input and output channels, and image sizes vary from layer to layer.

\[ \Theta_{\text{real}} = \Theta_{\text{peak}} \cdot \prod_{i} \eta_{i} \]  

(5.9)

**Tiling:** The number of rows are limited by the image window memory, which accommodates \( h_{\text{max}} \cdot n_{\text{ch,in}} \) words of \( w_{k} \cdot 12 \) bit, storing a maximum of \( h_{\text{max}} \) rows per input channel. In case the full image height does not fit into the memory, it can be split into several image tiles, which are then processed consecutively. The penalty is the \((h_{k} - 1)\) rows by which the tiles need to overlap and thus are loaded twice vertically. The impact on throughput can be determined by the tiling efficiency

\[ \eta_{\text{tile}} = \frac{h_{\text{im}}}{h_{\text{im}} + \left( \lceil \frac{h_{\text{im}}}{h_{\text{max}}} \rceil - 1 \right) (h_{k} - 1)} . \]  

(5.10)

**(Input) Channel Idling:** The number of output and input channels usually does not correspond to the number of output and input channels processed in parallel by this core. The output and input channels are partitioned into blocks of \( n_{\text{ch}} \times n_{\text{ch}} \). Then the outputs of these blocks have to be summed up pixel-wise outside the accelerator.

In the first few layers, the number of input channels \( n_{\text{in}} \) can be smaller than the number of output channels \( n_{\text{out}} \). In this case, the output bandwidth is limiting the input bandwidth by a factor of \( \eta_{\text{chIdle}} \).

\[ \eta_{\text{ch Idle}} = \frac{n_{\text{in}}}{n_{\text{out}}} . \]  

(5.11)

Note that this factor only impacts throughput, not energy efficiency. Using less than the maximum available number of input channels only results in more cycles being spent idling, during which only a negligible amount of energy (mainly leakage) is dissipated.

**Border Considerations:** To calculate one pixel of an output channel, at least \( h_{k}^{2} \) pixels of each input channel are needed. This leads to a reduction of \( \frac{1}{2} (h_{k} - 1) \) pixels on each side. While in some cases this is acceptable, many and particularly deep CNNs perform
5.4. RESULTS

zero-padding to keep a constant image size, adding an all-zero halo around the image. In case of zero-padding, \( \frac{h_k-1}{2} \) columns need to be pre-loaded, this introduces latency, but does not increase idleness as the same number of columns need to be processed after the last column where in the meantime the first columns of the next image can be preloaded to the image and therefore \( \eta_{\text{border}} = 1 \). For non-zero padded layers, the efficiency is reduced by the factor

\[
\eta_{\text{border, non-zero-padded}} = \frac{h_k - 1}{w_{im}} \cdot \frac{h_k - 1}{h_{im}}. \tag{5.12}
\]

5.4.2 Experimental Setup

To evaluate the performance and energy metrics of the proposed architecture and to verify the correctness of the generated results, we developed a testbench, which generates the control signals of the chip, reads the filters and the input images from a raw file, and streams the data to the chip. The output is monitored and compared to the expected output feature maps, which are read from a file, too. To calculate the expected responses, we have implemented a bit-true quantized spatial convolution layer in Torch, which acts as a golden model. The power results are based on post place & route results of the design. The design was synthesized with Synopsys Design Compiler J-2014.09-SP4, while place-and-route was performed with Cadence Innovus 15.2. The UMC 65nm standard cell libraries used for implementation were characterized using Cadence Liberate 12.14 in the voltage range 0.6 V - 1.2 V, and in the typical process corner at the temperature of 25 °C. The power simulations were performed with Synopsys PrimePower 2012.12, based on Value Change Dump (VCD) files extracted from simulations of real-life workloads running on the post place and route netlist of the design. These simulations were done with the neural network presented in [208] on the Stanford backgrounds data set [247] (715 images, 320 × 240 RGB, scene-labeling for various outdoor scenes), where every pixel is assigned with one of 8 classes: sky, tree, road, grass, water, building, mountain and foreground object. The I/O power was approximated by power measurements on chips of the same technology [208] and scaled to the actual operating frequency of YodaNN.
The final floorplan of YodaNN is shown in Fig. 5.10. The area is split mainly among the SCM memory with 480 kGE, the binary weights filter bank with 333 kGE, the SoP units with 215 kGE, the image bank with 123 kGE and the area distribution is drawn in Fig. 5.6. The core area is 1.3 MGE (1.9 mm²). The chip runs at a maximum frequency of 480 MHz @ 1.2 V and 27.5 MHz@0.6 V.

5.4.3 Fixed-Point vs. YodaNN

In this section, we compare a fixed-point baseline implementation with a binary version with fixed filter kernel size of $7 \times 7$ and $8 \times 8$ channels including an SRAM for input image storage. The results are summarized in Table 5.3. The reduced arithmetic complexity and the replacement of the SRAM by a latch-based memory shortened the critical path delay. Three pipeline stages between the memory and the channel summers were used in the fixed-point baseline version could be reduced to one pipeline stage. The peak throughput could still be increased from 348 GOP/s to 377 GOP/s at a core voltage of 1.2 V and the core power was reduced by 79% to 39 mW, which leads to a 5.1× better core energy efficiency and 1.3× better core area efficiency. As UMC 65nm technology SRAMs fail below 0.8 V, we can get even better
results by reducing the supply voltage to 0.6 V thanks to our SCM implementation. Although the peak throughput drops to 15 GOp/s, the core power consumption is reduced to 260 µW, and core energy efficiency rises to 59 TOp/s/W, which is an improvement of 11.6× compared to the fixed-point architecture at 0.8 V.

Table 5.3. Fixed-Point Q2.9 vs. Binary Architecture 8×8

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Q2.9(^a)</th>
<th>Bin.</th>
<th>Q2.9(^a)</th>
<th>Bin.</th>
<th>Bin.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply (V)</td>
<td>1.2</td>
<td>1.2</td>
<td>0.8</td>
<td>0.8</td>
<td>0.6</td>
</tr>
<tr>
<td>Peak Throughput (GOp/s)</td>
<td>348</td>
<td>377</td>
<td>131</td>
<td>149</td>
<td>16</td>
</tr>
<tr>
<td>Average Power Core (mW)</td>
<td>185</td>
<td>39</td>
<td>31</td>
<td>5.1</td>
<td>0.26</td>
</tr>
<tr>
<td>Average Power Device (mW)</td>
<td>580</td>
<td>434</td>
<td>143</td>
<td>162</td>
<td>15.54</td>
</tr>
<tr>
<td>Core Area (MGE)</td>
<td>0.72</td>
<td>0.60</td>
<td>0.72</td>
<td>0.60</td>
<td>0.60</td>
</tr>
</tbody>
</table>

Efficiency metrics

<table>
<thead>
<tr>
<th></th>
<th>Q2.9(^a)</th>
<th>Bin.</th>
<th>Q2.9(^a)</th>
<th>Bin.</th>
<th>Bin.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy Core (TOp/s/W)</td>
<td>1.88</td>
<td>9.61</td>
<td>4.26</td>
<td>29.05</td>
<td>61.23</td>
</tr>
<tr>
<td>Energy Device (TOp/s/W)</td>
<td>0.60</td>
<td>0.87</td>
<td>0.89</td>
<td>0.92</td>
<td>0.98</td>
</tr>
<tr>
<td>Area Core (GOp/s/MGE)</td>
<td>487</td>
<td>631</td>
<td>183</td>
<td>247</td>
<td>25</td>
</tr>
<tr>
<td>Area Device (GOp/s/MGE)</td>
<td>161</td>
<td>175</td>
<td>61</td>
<td>69</td>
<td>7</td>
</tr>
</tbody>
</table>

\(^a\) A fixed-point version with SRAM is used as baseline comparison and 8×8 channels and 7×7 filters.

5.5 Latch-based memory vs SRAM

As discussed in Section 5.3.3, the choice to use a latch-based image memory improves core energy efficiency by an additional factor of 2.0× by the extended voltage range. On the other hand, considering SCM implementation for image bank, the area increases as well. In our architecture, we used a 1024-word wide memory of 6 × 12 bits. The area of the convolutional engine increases by 8.9× from 54 kGE to 480 kGE. This affects the core area efficiency; at a supply voltage of 1.2 V the binary version with SRAM outperforms the SCM with 2024 MOp/s/MGE compared to 624 MOp/s/MGE. Trivially this decreases even more with lower supply as throughput drops.

Fig. 5.11 shows the throughput and energy efficiency of YodaNN with respect to the baseline architecture for different voltage supplies, while Fig. 5.12 shows the breakdown of the core power an the operating
frequency of 400 MHz. Comparing the two 8×8 channels variants (fixed-point and binary weights), the power consumption was reduced from 185 mW to 39 mW, where the power could be reduced by 3.5× in the SCM, 4.8× in the SoP units and 31× in the filter bank. Although the power consumption of the core increases by 3.32× when moving from 8×8 to 32×32 channels, the throughput increases by 4×, improving energy efficiency by 20%. Moreover, taking advantage of more parallelism, voltage and frequency scaling can be exploited to improve energy efficiency for a target throughput. The support for different kernel sizes significantly improves the flexibility of the YodaNN architecture, but increases the core area by 11.2%, and the core power by 38% with respect to a binary design supporting 7×7 kernels only. The Scale-Bias unit occupies another 2.5 kGE area and consumes 0.4 mW at a supply voltage of 1.2 V and a operating frequency of 480 MHz. When I/O power is considered, increasing the number of channels is more beneficial, since we can increase the throughput while the total device power does not increase at the same

Figure 5.11: Comparison of core energy efficiency and throughput for the baseline architecture (fixed-point Q2.9, SRAM, 8×8 channels, fixed 7×7 filters) with final YodaNN (binary, SCM, 32×32 channels, supporting several filters).
rate. We estimate a fixed contribution of 328 mW for the the I/O power at 400 MHz. Table 5.4 provides an overview of the device energy efficiency for different filter kernel sizes at 1.2 V core and 1.8 V pad supply. The device energy efficiency raises from 856 GOp/s/W in the 8×8 architecture to 1611 in the 16×16 and to 2756 in the 32×32.

Table 5.4. Device Energy Efficiency for Different Filters and Architectures

<table>
<thead>
<tr>
<th>Archit.</th>
<th>Q_{2.9}</th>
<th>8×8</th>
<th>16×16</th>
<th>32×32</th>
<th>32^2 (fixed)</th>
<th>[GOp/s/W]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7×7</td>
<td>600</td>
<td>856</td>
<td>1611</td>
<td>2756</td>
<td>3001</td>
<td>[GOp/s/W]</td>
</tr>
<tr>
<td>5×5</td>
<td>611</td>
<td>1170</td>
<td>2107</td>
<td></td>
<td></td>
<td>[GOp/s/W]</td>
</tr>
<tr>
<td>3×3</td>
<td>230</td>
<td>452</td>
<td>859</td>
<td></td>
<td></td>
<td>[GOp/s/W]</td>
</tr>
</tbody>
</table>

5.5.1 Real Applications

For a comparison based on real-life CNNs, we have selected several state-of-the-art networks that exploit binary weights. This includes the CNNs from the BinaryConnect paper for Cifar-10 and SVHN [233], and the well-known networks VGG-13, VGG-19 [248], ResNet-18, ResNet-34 [8], and AlexNet [6], which were successfully implemented with binary weights by Rastegari et al. [79] (not XNOR-net). The layer
### Table 5.5: Several Widely-Known Convolutional Neural Networks in the High-Efficiency Corner

| Model          | ImageNet | Width | Height | Filters 1 | Filters 2 | Filters 3 | Filters 4 | Filters 5 | Filters 6 | Filters 7 | Filters 8 | Filters 9 | Filters 10 | Filters 11 | Filters 12 | Filters 13 | Filters 14 | Filters 15 | Filters 16 | Filters 17 | Filters 18 | Filters 19 | Filters 20 | Filters 21 | Filters 22 | Filters 23 | Filters 24 | Filters 25 | Filters 26 | Filters 27 | Filters 28 |
|----------------|----------|-------|--------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| AlexNet        | [6]      | 224   | 224    | 3         | 48        | 4         | 0.95      | 0.09      | 0.35      | 1.4       | 12.1      | 520       | 364.7     | 42.9      | 364.7     | 42.9      | 364.7     | 42.9      | 364.7     | 42.9      | 364.7     | 42.9      | 364.7     | 42.9      | 364.7     | 42.9      | 364.7     | 42.9      |
| ResNet-18/34   | [8]      | 224   | 224    | 3         | 64        | 1         | 0.86      | 0.09      | 0.35      | 4.4       | 15.1      | 236       | 53.3      | 15.7      | 106.7     | 31.1      | 106.7     | 31.1      | 106.7     | 31.1      | 106.7     | 31.1      | 106.7     | 31.1      | 106.7     | 31.1      |
| VGG-13/19      | [32]     | 224   | 224    | 3         | 64        | 1         | 0.95      | 1.00      | 1.00      | 19.1      | 56.2      | 231       | 11.9      | 4.0       | 11.9      | 4.0       | 11.9      | 4.0       | 11.9      | 4.0       | 11.9      | 4.0       | 11.9      | 4.0       | 11.9      | 4.0       | 11.9      | 4.0       |
Table 5.6. Several Widely-Known Convolutional Neural Networks (cont.)

<table>
<thead>
<tr>
<th>Network</th>
<th>L</th>
<th>$h_k$</th>
<th>w</th>
<th>h</th>
<th>$n_{in}$</th>
<th>$n_{out}$</th>
<th>$\times$</th>
<th>$\eta_{tile}$</th>
<th>$\eta_{Idle}$</th>
<th>$\bar{P}_{real}$</th>
<th>$\Theta_{real}$</th>
<th>$\text{EnEff}$</th>
<th>$#\text{MOp}$</th>
<th>t</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>px</td>
<td>px</td>
<td>px</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BinaryConnect</td>
<td>1</td>
<td>3</td>
<td>32</td>
<td>32</td>
<td>3</td>
<td>128</td>
<td>1</td>
<td>1.00</td>
<td>0.09</td>
<td>0.35</td>
<td>1.9</td>
<td>16.0</td>
<td>7</td>
<td>3.8</td>
<td>0.4</td>
</tr>
<tr>
<td>Cifar-10 [233]</td>
<td>2</td>
<td>3</td>
<td>32</td>
<td>32</td>
<td>128</td>
<td>128</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>20.1</td>
<td>59.2</td>
<td>302</td>
<td>15.0</td>
<td>5.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>16</td>
<td>16</td>
<td>128</td>
<td>256</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>20.1</td>
<td>59.2</td>
<td>151</td>
<td>7.5</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>16</td>
<td>16</td>
<td>256</td>
<td>256</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>20.1</td>
<td>59.2</td>
<td>302</td>
<td>15.0</td>
<td>5.1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>256</td>
<td>512</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>20.1</td>
<td>59.2</td>
<td>151</td>
<td>7.5</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>512</td>
<td>512</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>20.1</td>
<td>59.2</td>
<td>302</td>
<td>15</td>
<td>5.1</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>FC</td>
<td>4</td>
<td>4</td>
<td>512</td>
<td>1024</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>FC</td>
<td>1</td>
<td>1</td>
<td>1024</td>
<td>1024</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>SVM</td>
<td>1</td>
<td>1</td>
<td>1024</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>BinaryConnect</td>
<td>1</td>
<td>3</td>
<td>32</td>
<td>32</td>
<td>3</td>
<td>128</td>
<td>1</td>
<td>1.00</td>
<td>0.09</td>
<td>0.35</td>
<td>1.9</td>
<td>16.0</td>
<td>7</td>
<td>3.8</td>
<td>0.4</td>
</tr>
<tr>
<td>SVHN [233]</td>
<td>2</td>
<td>3</td>
<td>16</td>
<td>16</td>
<td>128</td>
<td>256</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>20.1</td>
<td>59.2</td>
<td>151</td>
<td>7.5</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>256</td>
<td>512</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>20.1</td>
<td>59.2</td>
<td>151</td>
<td>7.5</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>FC</td>
<td>4</td>
<td>4</td>
<td>512</td>
<td>1024</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Legend:  
- L: layer, $h_k$: kernel size, w: image width, h: image height, $n_i$: input channels, $n_o$: output channels, $\times$: quantity of this kind of layer,  
- $\eta_{tile}$: tiling efficiency, $\eta_{ch.Idle}$: channel idling efficiency, $\bar{P}_{real}$ Normalized Power consumption in respect to active convolving mode,  
- $\Theta_{real}$: actual throughput, EnEff: Actual Energy Efficiency, $\#\text{MOp}$: Number of operations (additions or multiplications, in millions), t: time, E: needed processing energy  
- a The 11 $\times$ 11 kernels are split into two 6 $\times$ 6 and two 5 $\times$ 5 kernels as described in Section 5.5.1.
configurations and the related metrics are summarized in Table 5.5. As described in Section 5.3.1, the layers are split into blocks of $n_{in} \times n_{out} = 32 \times 32$ channels in case of a kernel size of $h_k^2 = 7^2$ and $n_{in} \times n_{out} = 32 \times 64$ elsewhere. The first layers have a high idle rate, but the silenced SoP units consume roughly no power. To account for this, we introduce $P_{real} = P_{eff}/P_{max}$, which is calculated.

The first layer of AlexNet uses $11 \times 11$ filters and needs to be split into smaller kernels. We split them into 2 filters of $6 \times 6$ (top-left, bottom-right) and 2 filters of $5 \times 5$ (bottom-left, top-right), where the center pixel is overlapped by both $6 \times 6$ kernels. By choosing the value for the overlapping weight appropriately, it is possible to prevent the need of additional $1 \times 1$ convolutions: if the original weight is 1, the overlapping weight of both $6 \times 6$ kernels are chosen to be 1. Otherwise, $-1$ is assigned to one of them and 1 to the other. Instead of $1 \times 1$ convolutions, just the sum of the identities of all input channels needs to be subtracted. The summing of the contributions and subtracting of the identities is done off-chip. Table 5.7 gives an overview of the energy efficiency, throughput, actual frame rate, and total energy consumption for calculating the convolutions, including channel biasing and scaling in the energy-optimal configuration (at 0.6 V). Table 5.8 shows the same metrics and CNNs for the high-throughput setting at 1.2 V. It can be noticed that in the energy-optimal operating point, the achieved throughput is about half of the maximum possible throughput of 55 GOp/s for most of the listed CNNs. This can be attributed to the smaller-than-optimal filter size of $3 \times 3$, which is frequently used and limits the throughput to about 20 GOp/s. However, note that the impact on peak energy-efficiency is only minimal with 59.20 instead of 61.23 GOp/s/W.

The average energy efficiency of the different networks is within the range from 48.1 to 56.7 TOp/s/W, except for AlexNet which reaches 14.1 TOp/s/W due to the dominant first layer which requires a high computational effort while leaving the accelerator idling for a large share of the cycles because of the small number of input channels. The fourth column in tables 5.7 and 5.8 shows the frame rate, which can be processed by YodaNN, excluding the fully connected layers and the chip configuration. In the throughput optimal case, the achieved frame rate is between 13.3 (for VGG-19) and 1428 FPS (for
the BinaryConnect-SVHN network) with a chip power of just 153 mW. In the maximum energy efficiency corner YodaNN achieves a frame rate between 0.5 and 53.2 FPS at a power of 895 µW.

Table 5.7. Overview of Several Networks in an Energy Optimal Use Case ($V_{core} = 0.6 \text{ V}$) on a YodaNN Accelerator

<table>
<thead>
<tr>
<th>Network</th>
<th>img size $h_{in} \times w_{in}$</th>
<th>Avg. EnEff $\bar{\Theta}$</th>
<th>$\Theta$</th>
<th>Energy $\mu J$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC-Cifar-10</td>
<td>32×32</td>
<td>56.7</td>
<td>19.1</td>
<td>15.8</td>
</tr>
<tr>
<td>BC-SVHN</td>
<td>32×32</td>
<td>50.6</td>
<td>16.5</td>
<td>53.2</td>
</tr>
<tr>
<td>AlexNet</td>
<td>224×224</td>
<td>14.1</td>
<td>3.3</td>
<td>0.5</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>224×224</td>
<td>48.1</td>
<td>16.2</td>
<td>4.5</td>
</tr>
<tr>
<td>ResNet-34</td>
<td>224×224</td>
<td>52.5</td>
<td>17.8</td>
<td>2.5</td>
</tr>
<tr>
<td>VGG-13</td>
<td>224×224</td>
<td>54.3</td>
<td>18.2</td>
<td>0.8</td>
</tr>
<tr>
<td>VGG-19</td>
<td>224×224</td>
<td>55.9</td>
<td>18.9</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 5.8. Overview of Several Networks in a Throughput Optimal Use Case ($V_{core} = 1.2 \text{ V}$) on a YodaNN Accelerator

<table>
<thead>
<tr>
<th>Network</th>
<th>img size $h_{in} \times w_{in}$</th>
<th>Avg. EnEff $\bar{\Theta}$</th>
<th>$\Theta$</th>
<th>Energy $\mu J$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC-Cifar-10</td>
<td>32×32</td>
<td>8.6</td>
<td>525</td>
<td>435</td>
</tr>
<tr>
<td>BC-SVHN</td>
<td>32×32</td>
<td>7.7</td>
<td>454</td>
<td>1429</td>
</tr>
<tr>
<td>AlexNet</td>
<td>224×224</td>
<td>2.2</td>
<td>90</td>
<td>14</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>224×224</td>
<td>7.3</td>
<td>446</td>
<td>125</td>
</tr>
<tr>
<td>ResNet-34</td>
<td>224×224</td>
<td>8.0</td>
<td>495</td>
<td>68</td>
</tr>
<tr>
<td>VGG-13</td>
<td>224×224</td>
<td>8.3</td>
<td>502</td>
<td>22</td>
</tr>
<tr>
<td>VGG-19</td>
<td>224×224</td>
<td>8.5</td>
<td>520</td>
<td>13</td>
</tr>
</tbody>
</table>

5.5.2 Comparison with State-of-the-Art

In Section 5.2, the literature from several software and architectural approaches have been described. The 32×32 channel YodaNN is able to reach a peak throughput of 1.5 TOp/s which outperforms NINEX [63] by a factor of 2.7. In core energy efficiency the design outperforms k-Brain, NINEX by 5× and more. If the supply voltage
is reduced to 0.6 V, the throughput decreases to 55 GOp/s but the energy efficiency rises to 61.2 TOp/s, which is more than an order-of-magnitude improvement over the previously reported results [56, 57, 63]. The presented architecture also outperforms the compressed neural network accelerator EIE in terms of energy efficiency by 12× and in terms of area efficiency by 28×, even though they assume a very high degree of sparsity with 97% zeros [68]. Fig. 5.13 gives a quantitative comparison of the state-of-the-art in energy efficiency and area efficiency. For the sweep of voltages between 1.2 V and 0.6 V, YodaNN builds a clear pareto front over the state of the art.

5.6 Conclusion

We have presented a flexible, energy-efficient, and performance scalable CNN accelerator. The proposed architecture is the first ASIC design exploiting recent results on binary-weight CNNs, which greatly simplifies the complexity of the design by replacing fixed-point MAC units with simpler complement operations and multiplexers without
negative impact on classification accuracy. To further improve energy efficiency and extend the performance scalability of the accelerator, we have implemented latch-based SCMs for on-chip data storage to be able to scale down the operating voltage even further. To add flexibility, we support seven different kernel sizes: $1 \times 1$, $2 \times 2$, ..., $7 \times 7$. This enables efficient evaluation of a large variety of CNNs. Even though this added flexibility introduces a 29% reduction in energy efficiency, outstanding overall energy efficiency of 61 TOp/s/W is achieved. The proposed accelerator surpasses state-of-the-art CNN accelerators by 2.7× in peak performance with 1.5 TOp/s, by 10× in peak area efficiency with 1.1 TOp/s/MGE and by 32× peak energy efficiency with 61.2 TOp/s/W. YodaNN’s power consumption at 0.6 V is 895 µW with an average frame rate of 11 FPS for state-of-the-art CNNs and 16.8 FPS for ResNet-34 at 1.2 V.
Chapter 6

XNORBIN: BNN Hardware Acceleration

In the previous chapter, we have introduced the first Binary-Weight Neural Networks accelerator. In this chapter, we go to the extreme case of quantization and binarize also the activations like we have done in the embedded domain in Chapter 3. BNNs reduce not just the weights memory requirements, but also the feature map volume by 8-12×. As introduced in Chapter 3, the energy-intensive sum-of-product can be converted to XNOR and binary accumulation. We present XNORBIN, among the first accelerators for binary CNNs. Thanks to the efficient data re-use and optimized memory hierarchy and exploiting efficient latch-based memories XNORBIN achieves an energy efficiency of 205 TOP/s/W in globalfoundries 22 nm FDX technology.

6.1 Introduction

Binary Weight Neural Networks have been shown to work with a negligible loss on simple ML tasks and with little loss on challenging tasks (i.e., ImageNet object recognition task), and enable very efficient hardware accelerator like YodaNN. Still, the intermediate feature maps occupy a large amount of memory. BNNs, introduced in Section 3.3.3,
quantize not just the weights, but also the intermediate feature maps, which reduces the overall memory size and bandwidth constraints by up to 32×.

General hardware accelerators have been introduced in Section 5.2. Recently, there have been presented few BNN accelerators exploiting the extreme reduction in arithmetic complexity where full-precision multiply-accumulate become binary XNOR-popcount. Conti et al. presented a 46 TOPS/W accelerator tightly-connected to a general-purpose processor (without considering off-accelerator memory and I/O costs) [249], UNPU is a stand-alone accelerator for flexible weights and feature maps and reaches 51 TOPS/W for fully-binary NN [72], and BinarEye presents a full-custom accelerator for BNN with 64 channels and 2×2 kernels and reached a peak core energy-efficiency of 230 TOPS/W.

In this chapter, we present XNORBIN, a hardware accelerator targeting fully-binary CNNs to address both the memory and computation energy challenges. The key operations of BNNs are 2D-convolutions of multiple binary (+1/-1) input feature maps and binary (+1/-1) filter kernel sets, resulting in multiple integer-valued feature maps. These convolutions can be formulated as many parallel XNOR-and-popcount operations with the potential for intensive data reuse. The activation function and the optional batch normalization can then be collapsed to a re-binarization on a pre-computed per-output feature map threshold value and can be applied on-the-fly. An optional pooling operation can be enabled after the re-binarization.

6.2 BNN and related HW optimization

We have introduced Binary Neural Networks in Sections 3.3.3 and 3.3.2 in the context of microcontrollers. In the following, we are shortly introducing BNNs again, but focus more on the implications for hardware acceleration. The BNNs are a subset of Neural Networks, whereas the intermediate feature maps and the weights are quantized to a single bit, and thus $\mathbf{I} \in \{-1, 1\}^{n_{in} \times h \times b}$ $\mathbf{W} \in \{-1, 1\}^{n_{out} \times n_{in} \times k_y \times k_x}$. While calculating the output feature maps, the full resolution is
preserved and is re-binarized after all input channel contributions have been summed together. Typically, the signum function

\[
\text{sgn}(x) = \begin{cases} 
-1 & x < 0 \\
1 & \text{else} 
\end{cases}
\]

is used as the activation function for re-binarization. Training of BNNs is not trivial, as gradients are not smooth anymore due to the high non-linearity of the parameter space. The most common approach bases on shadow weights in high precision (e.g., FP32). These weights are binarized during the forward-propagation. During back-propagation, the gradients are applied to the shadow weights. Even though the binarization itself is not derivable, it can be modeled as the identity function. This can be interpreted as propagating the stochastic expected value of the gradient to the weights (i.e., straight-through estimator) [179]. The \( k \)-th output feature map \( o_k \) is the sum of convolutions of every binarized input feature map \( \hat{i}_n \) with the corresponding binary weights \( \hat{w}_{k,n} \) and the bias \( C_k \):

\[
o_k = \text{sgn} \left( C_k + \alpha \sum_{n \in I} \text{sgn}(i_n) \ast \text{sgn}(w_{k,n}) \right) \tag{6.1}
\]

BNNs have much potential for optimizations: First, the memory footprint can be reduced up to \( 32 \times \) (i.e., in case of FP32), and secondly, multiply-accumulate can be simplified to \texttt{xnor}-accumulate. In the following, we will shortly introduce the mathematical optimization for direct HW benefits in more detail.

Bipolar activations and weights \( i_n, w_{k,n} \in \{-1, 1\} \) are mapped to the binary representation of \( \hat{i}_n, \hat{w}_{k,n} \in \{0, 1\} \), enabling the replacement of the multiplication with a binary \texttt{xnor} operation. The function mapping the bipolar values to the binary values is \( b(x) = \frac{1}{2}(x + 1) \) and needs to be compensated after accumulation by applying the following equation \( \sum_i \zeta_i = -i + 2 \sum_i \left[ \frac{1}{2}(\zeta + 1) \right] \) and by merging and rearranging, the formula can be turned into the same form as in Eq. 6.1, where as the multiplication within the convolutions are replaced by \texttt{xnor} operations indicated by \( \ast^{\oplus} \):
$o_k = sgn\left(C_k + \alpha \sum_{n=0}^{n_{in}-1} \left(2 \cdot \hat{i}_n \ast \tilde{w}_{k,n} - k_y k_x\right)\right)$

$= sgn\left(C'_k + \alpha' \sum_{n=0}^{n_{in}-1} \hat{i}_n \ast \tilde{w}_{k,n}\right)$

$o_k = sgn\left(\sum_{n=0}^{n_{in}/16-1} \sum_{(\Delta x, \Delta y)} 2\text{popcnt}\left(\hat{i}_{16n+16} \oplus \tilde{w}_{k,16n+16}\right) - 16\right)$

(6.2)

Even though the weights and feature maps stay binary, there are still non-binary intermediate values: On one hand side the accumulation itself and on the other hand side learned bias/scaling factors and batch normalization and can be written as follows:

$\hat{o}_k = sgn\left(\hat{C}_k + \hat{\alpha}_k \sum_{n=0}^{n_{in}-1} \hat{i}_n \ast \tilde{w}_{k,n} - \mu_k\right)$

(6.3)

This formula can be reformulated s.t. the signum function becomes a more general threshold function:

$o_k = \begin{cases} -1, & \sum_{n=0}^{n_{in}-1} \hat{i}_n \ast \tilde{w}_{k,n} < \theta_k \\ 1, & \text{else} \end{cases}$

(6.4)

Whereas $\theta_k = \frac{\hat{C}_k \sigma_k}{|\hat{\alpha}_k|} + \mu_k$. While re-ordering the inequality, sign inversion has to be taken into account (i.e., while multiplying with a negative number). The standard variation $\sigma_k$ is positive by definition, but the learned scaling factor $\alpha_k$ can be negative. To counteract against the implied sign change, all weights can be inverted in case of negative $\alpha_k$, and just a single quantized threshold $\theta_k$ needs to be stored per feature map.

Pooling is applied after the convolution, scaling, and batch normalization, but before the re-binarization and, therefore, non-binary. However, the threshold comparison operation that follows the pooling
operation is a monotonic function. The binarization and pooling function exhibit the commutative property enabling reversing the order of operation. Pooling operations can therefore be calculated as boolean operation. As such, max-pooling is implemented with \text{AND} reduction, min-pooling with \text{OR} reduction, and average-pooling with boolean majority voting.

\[
Pool(o_k(x, y)) = \begin{cases} 
-1, & \max_{m, n \in \{0, 1\}} (o_k(2x + m, 2y + n)) < \theta_k \\
1, & \text{else}
\end{cases} 
\] (6.5)

\[
= \begin{cases} 
-1, & \bigwedge_{m, n \in \{0, 1\}} (o_k(2x + m, 2y + n) < \theta_k) \\
1, & \text{else}
\end{cases} 
\] (6.6)

6.3 Architecture

The architecture of XNORBIN is illustrated in Fig. 6.1 and is presented in the following:

**BPU Cluster**

The BPU Cluster is the core of the accelerator and consists of 7 Binary Processing Units (BPUs). Each BPU is performing a 1-D convolution
Figure 6.2: Architecture of BPU

Figure 6.3: A Pipelined Cluster of BPUs
of an image row with a kernel row. The multiplication operation is replaced with an XNOR gate and 16 XNOR gates are parallelizing the accumulation of 16 input channels, as described in Eq. 6.2.

The binary accumulation units are attached to the 16 XNOR gates; these units are replicated 7 times in each BPU in order to support convolution kernels sized up to $7 \times 7$. The outputs of all these instances are then added together to return the corresponding 2D inner product and pipelined to increase throughput. Each of the XNOR-sum instances is fed with the image and weight data through a controlled shift-register to enable data reuse (i.e., FM and weights).

**Multi-level Memory Hierarchy**

XNORBIN comes with three levels of memory and data buffering hierarchy:

L3) The feature map memory FMM stores the feature maps and the partial sums of the convolutions. The memory is divided into two blocks, where one serves as the data source (i.e., current input feature maps), and the other serves as data sink (i.e., partial or final output feature maps), and is swapped for every layer. If the FMM is dimensioned to fit the largest intermediate FMs, no energy-costly off-chip memory accesses are needed to store and load intermediate FMs.

L2) The row banks are used to buffer rows of the input feature maps for frequent accesses. Since these row banks need to be rotated when shifting the convolution window down, they are connected to the BPU array through a crossbar. The weight bank and the parameter buffer stores the weights, the binarization threshold, and the configuration parameters and is sized to fit the largest network to be supported but is also implemented as a cache-like buffer for an external flash memory storing these parameters.

L1) The crossbar connects the registers inside the BPUs, the *controlled shift registers* (CSRs, as illustrated in Fig. 6.2) containing kernel input feature map elements and the filter weight elements. These are shifted when the convolution window is moved forward.
All the data words in the CSRs are accessible in parallel and applied to the \texttt{xnor\_sum} units.

**DMU**

The Data Management Unit (DMU) moves data independently within the memory hierarchy. E.g., filling the Row Bank Memories (i.e., L2) with frequently reused features and weights from the main memory (i.e., L3). It is also responsible for storing back partial sums, and output feature maps back to the feature map memory.

**Scheduler**

According to a given layer configuration of a CNN, the Scheduler instructs the crossbar how to route feature map and weight data from banks to the BPUs in order to compute row-wise partial sums for each member in the batch.

**Near Memory Compute Unit CU**

An in-loop compute unit closer to the memories carries out one to one type of operations that do not exhibit data reuse. This means the data needs to be accessed from the main memory. The in-loop compute unit takes care of the following operations.

- Partial sum accumulation: When the BPU cluster sends the stream of partial sums related to a certain batch, the partial sum is accumulated until the last batch should be retrieved, accumulated, and stored back. In the first iteration, a precomputed initial threshold is added.

- Binarization. In this process, integer data are streamed into the compute unit and binarized. After the entire row tile has been binarized, the result is written back to the FMM.
6.3. ARCHITECTURE

As shown in Figure 6.4, datapath resources are shared among these different operations to gain area efficiencies. In addition, data from BPU Cluster is packed to match memory data width.

![Diagram of Datapath Near memory CU](image)

**Figure 6.4: Datapath Near memory CU**

### 6.3.1 Data Organization and Data Reuse

To support up to $7 \times 7$ kernel sizes, the processing core of XNORBIN is composed of an array (shown in Fig. 6.2) of 7 BPUs (Basic Processing Units), where every BPU includes a set of 7 $\text{xnor}_\text{sum}$ units (i.e., Fig. 6.2. These units calculate the XNOR-and-popcount result on 16 bit vectors, containing values of 16 feature maps at a specific pixel. The outputs of all 7 $\text{xnor}_\text{sum}$ units in a BPU are added-up, computing one output value of a 1D convolution on an image row each cycle. On the next higher level of hierarchy, the results of the BPUs are added up to produce one output value of a 2D convolution (illustrated in Fig. 6.2). Cycle-by-cycle, a convolution window slides horizontally over the image. The resulting integer value is forwarded to the DMU controller via the near-memory compute unit (CU). The CU accumulates the partial results by means of a read-add-write operation since the feature maps are processed in tiles of 16. After the final accumulation of partial results, the unit also performs the thresholding/re-binarization operation (i.e., activation and batch normalization). When binary results have to be written back to memory, the DMU also handles packing them into 16 bit words.
The bank memories have to be fixed to finite size, but this does not limit the supported networks as the channels are tiled into fixed blocks of $\tilde{c}_i$ input channels and $\tilde{c}_o$ output channels (i.e., batch size). XNORBIN exploits the following data-reuse patterns:

- **Kernel-level filter reuse (KLFR):** The same kernel is slid across the image tensor to calculate contributions towards adjacent outputs.
- **Kernel-level feature reuse (KLIR):** The input feature maps are reused for several output channels.
- **Row-level input channel reuse (ICR):** The same set of input image rows are reused when processing a single row of output channels within a batch. Even moving from one row of output to the next row can lead to the reuse of several input image rows due to vertical kernel window overlap.
- **Row-level filter reuse (RLFR):** The same kernels are used to produce different spatial output rows of a given channel.

### 6.3.2 Scheduling

The scheduling is determined with the objective of maximizing the data reuse at different levels of the memory hierarchy. The memory transfers between memories and operational units is illustrated in Fig. 6.5 and the scheduling algorithm is explained more in detail in Alg. 5 based on the filter dimensions $k_w$ and $k_h$, the spatial input dimensions $i_w$ and $i_h$, the depths (i.e., input channels $c_i$, and output channels $c_o$) and the channels tile sizes $\tilde{c}_i$ and $\tilde{c}_o$. Parallel execution is indicated in line 6, 8 and 10. In order to maximize kernel-level reuse, filter weights are retained in BPUs while streaming selected image rows through BPUs. Partial sums of several parallel output channels are computed to maximize row-level image reuse.

XNORBIN operates in parallel over the kernel-sized tile $k_w \times k_h$ and $\tilde{c}_i$ input channels in every BPU cluster, whereas every BPU cluster is calculating the contributions for one single output channels within the output channel tile $\tilde{c}_o$. Then the values binarized and pooling is applied as described in Sec. 6.2. This procedure is then repeated
Algorithm 5 High-level scheduling of BNN Calculation on XNORBIN

Require: \(k_w, k_h, i_w, i_h, c_i, c_o, \tilde{c}_i, \tilde{c}_o\)

1: for \(n_o \leftarrow 0\) to \(c_o/\tilde{c}_o\) do
2:     for \(n_i \leftarrow 0\) to \(c_i/\tilde{c}_i\) do
3:         for \(n_{row} \leftarrow 0\) to \(i_h\) do
4:             for \(b_o \leftarrow 0\) to \(\tilde{c}_o\) (per BPU cluster in parallel) do
5:                 pass kernels of channel \(b_o\) to Bank memory
6:                 // parallize in HW
7:                 for \(k_{row} \leftarrow -(k_h/2)\) to \((k_h/2)\) do
8:                     // parallize in HW
9:                     for \(k_{col} \leftarrow -(k_w/2)\) to \((k_w/2)\) do
10:                        // parallize in HW
11:                         for \(b_i \leftarrow 0\) to \(\tilde{c}_i\) do
12:                             pass input feature map pixel \((b_i, n_{row}, n_{col})\) and
13:                             weight \((b_o, b_i, k_{col}, k_{row})\) to BPU Cluster \(b_o\)
14:                             calculate xnor-popcount and accumulate
15:                         end for
16:                     end for
17:                 end for
18:             end for
19:         end for
20:     end for
21: end for
22: Binarize final partial sums
23: Pool operation (if applicable)
24: end for
Figure 6.5: Schedule and Illustration of Memory Transfers. In case of $c_n = c_{\text{in}}$...
for all rows and columns of the entire input feature map and for all tiles of output feature maps. XNORBIN supports CNNs of arbitrary depths by streaming the network parameters from external memory. However, the size of the BNN’s layer with the largest pair of input and output feature maps has to fit into the FMM (i.e., 250 kbit for the actual implementation of XNORBIN). The succession of CNN layers is configurable. XNORBIN supports adjustable feature map dimensions (height, width, channel length), as long as the volume of the largest intermediate feature map fits in FMM. It can handle convolution windows of up to $7 \times 7$ and configurable stride. Furthermore, any convolution layer with a filter size larger than $7 \times 7$ would need to be split into smaller convolutions due to the number of parallel working BPUs, xnor_sum units per BPU, the number of row banks, and the size of the CSRs, thereby introducing a large overhead. Convolution window size, stride, and order of layers are configurable, and arbitrarily deep networks are possible. A Python compiler is used to create the data stream to the accelerator from a high-level software description of the trained CNN.

6.4 Scalability

XNORBIN is not limited to $7 \times 7$ convolutions. It can be configured to handle any smaller filter sizes down to $1 \times 1$. However, the size of BNN’s largest pair of input and output feature maps (pixels $\times$ number of maps for both) has to fit into the main memory (i.e., 404 kbit for the actual implementation of XNORBIN). Furthermore, any convolution layer with a filter size larger than $7 \times 7$ would need to be split into smaller convolutions due to the number of parallel working BPUs, xnor_sum units per BPU, the number of row banks, and the size of the CSRs, thereby introducing a large overhead. There are no limitations to the depth of the network when streaming the network parameters from external flash memory.
Figure 6.6: Throughput vs. Core Energy Efficiency for various timing constraints at 0.4 V supply voltage (GF22 7.5T 0.4 V,TT,25C) for the average performance of convolution layers with different kernel sizes.

Figure 6.7: Floorplan of XNORBIN
6.5 Results

6.5.1 Physical Implementation

XNORBIN has been implemented with a 7.5 track standard-cell libraries in Globalfoundries 22nm FDX technology, synthesized with Synopsys Design Compiler 2018.06. Cadence Innovus 18.11 was used for back-end design and power simulation, and Questa Modelsim 10.6b has been used for verification and extraction of switching activities for power simulation. To reach the highest energy efficiency, we are using the lowest supply voltage corner available, which is 0.4 V and with a forward body-bias voltage of 0.1 V\(^1\).

SRAM memory accesses have high energy costs and are inevitably frequent in Neural Networks. Furthermore, SRAMs typically do not scale down to the same voltage as standard cells. Therefore we implement all memories with latch-based standard-cell memory SCM. The data is stored in latch arrays, which can be accessed by logarithmically arranged clock gates and are otherwise silenced, thus do not consume any dynamic power [81]. One SCM bank is organized of 256 32-bit words. The feature map memories have been dimensioned to fit for the two largest consecutive layers of AlexNet and have therefore 16, and 32 banks, the weight memory 2 banks, and the 7 bank memories consist of 1 SCM bank each. The final floorplan is shown in Fig. 6.7. It can be seen that a large part of the chip (i.e., 97%) are memories, whereas the compute units just occupy 1% of the total chip area.

6.5.2 Experimental Results

We have synthesized (colored, i.e., \(\textcolor{red}{\circ \circ \circ \circ} \)) and run back-ends at various speed constraints, to find the best efficiency and throughput trade-off at 0.4 V. The results are shown in Fig. 6.6. Due to the large SCM memories, the chip has a comparably high amount of leakage which limits the core energy efficiency to 205 TOPS/W at a throughput of 241 GOPS for the full-utilized case of 7 \(\times\) 7 kernel sizes.

We have tested our design running the binary AlexNet model from XNOR-net [79], which comes pre-trained on the ImageNet dataset. The

\(^1\)All numbers presented are in typical corner at room temperature.
### Table 6.1. Key Figure of XNORBIN

<table>
<thead>
<tr>
<th>Physical Characteristics</th>
<th>Technology</th>
<th>GF 22 nm FDX (7.5 track)</th>
</tr>
</thead>
<tbody>
<tr>
<td># Pads</td>
<td></td>
<td>40 (i: 18, o:6, clk/test: 6, pwr: 8)</td>
</tr>
<tr>
<td>Core Area w/o SCM</td>
<td></td>
<td>0.025 mm²</td>
</tr>
<tr>
<td>Circuit Complexity w/o SCM</td>
<td></td>
<td>126 kGE</td>
</tr>
<tr>
<td>Core Area w/ SCM</td>
<td></td>
<td>0.70 mm²</td>
</tr>
<tr>
<td>Circuit Complexity w/ SCM</td>
<td></td>
<td>3'518 kGE</td>
</tr>
<tr>
<td>On-chip SCM</td>
<td></td>
<td>404 kbit</td>
</tr>
<tr>
<td><strong>Performance &amp; Efficiency @0.4V</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max Clock Frequency</td>
<td>core: 154 MHz</td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td>1.2 mW (core) + 7.8 mW (pad)</td>
<td></td>
</tr>
<tr>
<td>Peak Throughput</td>
<td>241.2 GOp/s</td>
<td></td>
</tr>
<tr>
<td>Core Power-Efficiency</td>
<td>204.9 TOp/s/W @ 0.4 V</td>
<td></td>
</tr>
<tr>
<td>Device Power Efficiency</td>
<td>26.9 TOp/s/W @ 0.4 V</td>
<td></td>
</tr>
<tr>
<td>FPS AlexNet binary layers</td>
<td>6.72 fps</td>
<td></td>
</tr>
</tbody>
</table>
6.5. RESULTS

Table 6.2. Comparison of various SoA accelerators

<table>
<thead>
<tr>
<th>Design</th>
<th>Power [mW]</th>
<th>Efficiency [GOp/s/W]</th>
<th>Freq. [MHz]</th>
<th>Core Area [mm$^2$]</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINN (FPGA)</td>
<td>2.3k</td>
<td>685</td>
<td>200</td>
<td>-</td>
<td>Z-7045</td>
</tr>
<tr>
<td>NeuFlow (FPGA)</td>
<td>10k</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td>IBM45</td>
</tr>
<tr>
<td>NeuFlow</td>
<td>600</td>
<td>490</td>
<td>400</td>
<td>13</td>
<td>TSMC65</td>
</tr>
<tr>
<td>Eyeriss</td>
<td>278</td>
<td>246</td>
<td>250</td>
<td>16</td>
<td>TSMC65</td>
</tr>
<tr>
<td>ShiDianNao</td>
<td>320</td>
<td>400</td>
<td>1000</td>
<td>5</td>
<td>TSMC65</td>
</tr>
<tr>
<td>EIE</td>
<td>590</td>
<td>5000</td>
<td>800</td>
<td>41</td>
<td>TSMC45</td>
</tr>
<tr>
<td>Origami (@0.8V)</td>
<td>core: 93</td>
<td>core: 803</td>
<td>189</td>
<td>3.09</td>
<td>UMC65</td>
</tr>
<tr>
<td>pads: 144</td>
<td>device: 220</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>YodaNN (@1.2V)</td>
<td>core: 39</td>
<td>core: 9610</td>
<td>480</td>
<td>1.91</td>
<td>UMC65</td>
</tr>
<tr>
<td>pads: 395</td>
<td>device: 870</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>YodaNN (@0.6V)</td>
<td>core: 0.26</td>
<td>core: 61k</td>
<td>28</td>
<td>1.91</td>
<td>UMC65</td>
</tr>
<tr>
<td>pads: 15.54</td>
<td>device: 980</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XNORBIN (@0.4V)</td>
<td>core: 1.2</td>
<td>core: 205k</td>
<td>154</td>
<td>0.70</td>
<td>gf22</td>
</tr>
<tr>
<td>pads: 7.8</td>
<td>device: 27k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

throughput and energy consumption per layer are shown in Tab. 6.3. The results are implicitly bit-true—there is no implementation loss such as going from FP32 to a fixed-point representation since all intermediate results are integer-valued or binary. A throughput of 241 GOp/s at an energy efficiency of 205 TOp/s/W has been achieved. The system consumes 1.18 mW@0.4V from which 56.1% are in the memory, 13.0% in the DMU and crossbar and 17.6% in the BPUs. The key performance and physical characteristics are presented in Tab. 6.1. The implementation parameters, such as memory sizes, have been chosen to support BNN models up to the size of binary AlexNet. We compare the energy efficiency of XNORBIN to state-of-the-art CNN accelerators in Tab. 6.2. To the best of our knowledge, this is the first hardware accelerator for binary neural networks. The closest comparison point are the FPGA-based FINN results [250] with a 299× higher energy consumption when running BNNs. The strongest competitor is YodaNN [81], which is a binary-weight CNN accelerator strongly limited by I/O energy, requiring 27× more energy per operation than XNORBIN.
### Table 6.3. Layer-wise performance on AlexNet at 0.4 V supply.

<table>
<thead>
<tr>
<th>L.</th>
<th>Func.</th>
<th>Ops</th>
<th>Cyc.</th>
<th>$P_{I/O}$</th>
<th>$P_{core}$</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Conv</td>
<td>447.9 M</td>
<td>2.1 M</td>
<td>14</td>
<td>33</td>
<td>14.4</td>
</tr>
<tr>
<td></td>
<td>Pooling</td>
<td>173.1 k</td>
<td>93.3 k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Binarize</td>
<td>43.3 k</td>
<td>21.6 k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Conv</td>
<td>149.5 M</td>
<td>1.8 M</td>
<td>18</td>
<td>27</td>
<td>12.0</td>
</tr>
<tr>
<td></td>
<td>Binarize</td>
<td>13.8 k</td>
<td>6.9 k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Conv</td>
<td>224.3 M</td>
<td>2.8 M</td>
<td>27</td>
<td>41</td>
<td>18.0</td>
</tr>
<tr>
<td></td>
<td>Binarize</td>
<td>13.8 k</td>
<td>6.9 k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Conv</td>
<td>149.5 M</td>
<td>2.4 M</td>
<td>18</td>
<td>36</td>
<td>15.8</td>
</tr>
<tr>
<td></td>
<td>Pooling</td>
<td>36.9 k</td>
<td>21.6 k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Binarize</td>
<td>9.2 k</td>
<td>4.6 k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Conv</td>
<td>37.7 M</td>
<td>9.4 M</td>
<td>755</td>
<td>140</td>
<td>61.3</td>
</tr>
<tr>
<td></td>
<td>Binarize</td>
<td>4.1 k</td>
<td>2.0 k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Conv</td>
<td>16.8 M</td>
<td>4.2 M</td>
<td>336</td>
<td>62</td>
<td>27.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Sigma$</td>
<td></td>
<td>1’026.0 M</td>
<td>22.9 M</td>
<td>1’166</td>
<td>339</td>
<td>148.7</td>
</tr>
</tbody>
</table>
6.6 Analysis Summary

Thanks to the binarization of the neural networks, the memory footprint of the intermediate results as well as the filter weights could be reduced by 8-32×, making XNORBIN capable of fitting all intermediate results of a simple, but realistic BNN, such as binary AlexNet, into on-chip memory with a mere total accelerator size of 0.7 mm². Furthermore, the computational complexity decreases significantly as full-precision multiplier-accumulate units are replaced by XNOR and pop-count operations. Furthermore, we have introduced thresholding to combine batch-normalization and the sigmoid activation function. Due to these benefits—smaller compute logic, keeping intermediate results on-chip, reduced model size, optimized latch-based memories—XNORBIN outperforms the overall energy efficiency of accelerators existing at the time of development by more than 27×.

6.7 Conclusion

Recently, there have been presented several new BNN accelerators with energy efficiencies from 20 to 51 TOP/s/W [72, 251–253]. BinarEye from Moons et al. [74] is currently leading the efficiency ranking with 230 TOP/s/W for fully-digital BNN accelerator, with 1.12× slightly better than XNORBIN. Comparing these numbers has to be taken with a grain of salt, as technology, supply voltages, and evaluation methods differ. In this case, the technologies are very similar (i.e., 22 nm for XNORBIN vs. 28 nm), but BinarEye has a much higher supply voltage of 0.66 compared to 0.4 V. It could be argued that BinarEye would get a ≈ 2× higher efficiency, but it has to be considered that BinarEye uses SRAM memories, which do not scale down to 0.4 V. In contrast, XNORBIN is entirely built upon latch-based SCM memories and can be scaled to the limit. Some loss in efficiency can be explained through the extensive memory-to-memory data transfers which have been introduced to use smaller memories in the compute units and to preload data in parallel. In the retrospect, multi-banked memories with direct access would have reduced the overall power consumption significantly. Furthermore, we use a very flexible datapath for kernels from 1×1 to 7×7 with any number of channels. On the other hand,
BinarEye restricts to $2 \times 2$ filters, which are very uncommon in common neural nets and fix the number of input channels to multiples of 64. At the time of this design, most common kernel sizes were $7 \times 7$ and are, therefore, the most efficient corner of our design, while $3 \times 3$ kernels introduce more idling of the compute units, and thus the energy efficiency reduces to $43.2 \text{ TOp/s/W}$. Both designs use latch-based memory for often accessed data, which uses $3.5 \times$ less access energy compared to normal SRAM memories (as shown in Sect. 5.3.3). Both designs exploit an efficient datapath for the binary multiply-accumulate with $\text{xnor}$ and binary adder tree.

If we take into account the accuracy trade-off, it turns out that binary neural networks are still having a huge gap in accuracy for challenging tasks. Notably, in the important ImageNet classification challenge the Top-1 accuracy drop is between 26.6 and 12.3 percentage points [79, 170, 181, 254], a significant decrease in performance. Recent approaches suggest to duplicate the binary neural network layers, and calculate them in parallel. This approach indeed brings the accuracy down to 3-4 percentage points comparable to recent ternary-weight neural networks, but increases on the other hand linearly the computational complexity in number of operations up to 8 times. YodaNN (introduced in Chapter 5 scaled to 22 nm) has therefore still an energy advantage at the same accuracy performance, as its energy efficiency of $145 \text{ TOp/s/W}$ in 22 nm technology$^2$ is just 38% lower than XNORBIN.

$^2$scaled from 65 nm to 22 nm, based on Dreslinksi et al. [1]
### Table 6.4. Storage elements in memory hierarchy.

<table>
<thead>
<tr>
<th></th>
<th>CSRs</th>
<th>Row Banks</th>
<th>Img Mem1</th>
<th>Img Mem2</th>
<th>Param Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem Size</td>
<td>1.5 kbit</td>
<td>16.8 kbit</td>
<td>128 kbit</td>
<td>256 kbit</td>
<td>3.2 kbit</td>
</tr>
<tr>
<td>Data width</td>
<td>16 bit</td>
<td>16 bit</td>
<td>32 bit</td>
<td>32 bit</td>
<td>32 bit</td>
</tr>
<tr>
<td>Type</td>
<td>register</td>
<td>2-port SCM</td>
<td>1-port SCM</td>
<td>1-port SCM</td>
<td>2-port SCM</td>
</tr>
<tr>
<td>Peak rd/cycle</td>
<td>7</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Peak wr/cycle</td>
<td>1</td>
<td>0.14</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Chapter 7

Hyperdrive: Solving the I/O Bottleneck in BWN HW Accelerators

Up to 205 TOp/s/W in energy efficiency have been achieved with binary-weight and binary neural networks, introduced in the two recent Chapters 5 and 6. Nevertheless, these and state-of-the-art accelerators do not take into account off-chip communication. If off-chip power is included, we have shown in Tab. 5.4, that the device energy efficiency of YodaNN drops from 62 TOp/s/W to 2.7, 2.1 or 0.86 TOp/s/W for the kernel sizes $7 \times 7$, $5 \times 5$ and $3 \times 3$, respectively. Especially for the recently used small kernel sizes (i.e., $3 \times 3$ and $1 \times 1$), the accelerators become I/O dominated. The common approach to keep the entire feature map on-chip limits the accelerator to a limited feature map volume and therefore excludes a set of machine learning tasks like smart surveillance cameras.

In this chapter, we present Hyperdrive: a novel BWN accelerator dramatically reducing the I/O bandwidth exploiting a novel binary-weight streaming approach, which can be used for arbitrarily sized convolutional neural network architecture and input resolution. Hyperdrive exploits the natural scalability of the compute units both
at chip-level and system-level. Hyperdrive chips can be systolically in a 2D mesh while processing the entire feature map together in parallel. Hyperdrive achieves 4.3 TOp/s/W system-level efficiency (i.e., including I/Os), 3.1× higher than state-of-the-art BWN accelerators, even if its core uses resource-intensive FP16 arithmetic for increased robustness.

7.1 Introduction

We have shown in chapter 5 that binarizing weights in DNNs simplifies the computations significantly and has shown the biggest impact on core compute-only energy with an energy efficiency of 60 TOp/s/W in 65 nm.

Recently, new BWN accelerators have been presented: QUEST [71] or UNPU [72], the latter have reached an energy efficiency of 50.6 TOp/s/W at a throughput of 184 GOp/s with 1-bit weights and 16-bit activations on 16 mm² of silicon in 65 nm technology, 1.2× less in core energy efficiency, and 8.1× in throughput compared to YodaNN.

However, state-of-the-art accelerators (introduced in section 5.2.2), YodaNN and XNORBIN fall into one of two categories:

1. They stream the entire or even partial FMs into and out of the accelerator ending up in a regime where I/O energy is far in excess of the energy spent on computation, hitting an energy efficiency wall: YodaNN has a core energy efficiency of 61 TOp/s/W, but including I/O power it is limited to 2.7 TOp/s/W; or

2. they assume to store the entire network’s weights and intermediate FMs on-chip. This severely constrains the DNN’s size that can be handled efficiently by a small low-cost IoT-end node class chip. It also prevents the analysis of high-resolution images, thus precluding many relevant applications such as object detection.

The main contributions of this work are:

1. A new and highly optimized yet flexible core architecture systolically scalable to high-resolution images to enable applications such as object detection.
2. A new computational model, which exploits the reduced size of the weights due to the binarization in BWNs. As the size of the weights becomes much smaller than the intermediate feature maps, Hyperdrive streams the weights instead of the intermediate feature maps. With this new method, Hyperdrive enables execution of state-of-the-art BWNs on tiny, power-constrained chips, while overcoming the I/O energy-induced efficiency wall.

3. An in-depth analysis of this architecture in terms of memory requirements, I/O bandwidth, and scalability including measurements of the chip implemented in GF 22 nm FDX technology, showing a $1.8 \times$ and $3.1 \times$ gain in energy efficiency in image classification and object detection, respectively, even though our core uses resource-intensive FP16 arithmetic for increased robustness.

4. We show that the system is systolically scalable to multiple chips with the elementary chip size fixed to a maximum area constraint arranged in a 2D mesh operating on tiles of the entire feature map. The extension is also implemented in GF 22 nm FDX technology and is evaluated on layout simulations, showing that even with the overhead of exchanging the border pixels, the I/O energy can be reduced up to $5.3 \times$ compared with state-of-the-art accelerators.

The remainder of this chapter is organized as follows. Sec. 7.2 and Sec. 7.3 introduce the Hyperdrive architecture and computational model, respectively, mainly focusing on its key innovation aspect: stationary feature-map and streaming binary-weights for reduced I/O bandwidth and improved system-level energy efficiency. Sec. 7.4 describes the extensions to the presented architecture enabling a systolic-scalable system composed of Hyperdrive chips. Sec. 7.5 presents the results of the chip implemented in 22nm FDX technology, providing details about its characterization, benchmarking, and comparison with respect to the state-of-the-art of binary-weight CNN accelerators. Finally, Sec. 7.6 closes this chapter with some final remarks.
7.2 Hyperdrive Architecture

Hyperdrive not only exploits the advantages of reduced weight memory requirements and computational complexity, but fundamentally differs from previous BWN accelerators [72, 73] and YodaNN presented in chapter 5. The main concepts can be summarized as:

1. Feature Maps are stored entirely on-chip, instead the weights are streamed to the chip (i.e., feature map stationary). Thanks to the binary nature of the weights the overall I/O demand is reduced dramatically.

2. Through its hierarchically systolic-scalable structure it allows to efficiently scale to any sized feature map and even with silicon area restriction it is still scalable by tailing on a 2D mesh of Hyperdrive chips.

Hyperdrive is a scalable and flexible binary-weight neural networks accelerator that can be parametrized to fit a wide range of networks targeting a variety of tasks from classification to object detection. Fig. 7.1 shows a block diagram of Hyperdrive, where $M \times N$ indicate the spatial parallelism (i.e., size of the FM), while $C$ the output channel parallelism. It is composed of the following components:

- **Feature Map Memory (FMM):** Is a multi-banked memory storing input and output FMs.

- **Array of $C \times M \times N$ Tile Processing Units (TPUs):** A single Tile-PU is illustrated in Fig. 7.2. It contains
  
  1. a half-precision float adder/subtractor to accumulate partial sums of the output pixels, bias and the bypass input FM (in case of residual blocks),
  2. a half-precision multiplier for the FM-wise batch-normalization shared among the Tile-PUs of the same tile, and
  3. a ReLU activation unit.

Each Tile-PU$_{(c,x,y)}$ is operating on the spatial tile $(x, y)$ of the $M \times N$ tiles and on the output channel $c$ from $C$. Each Tile-PU
Figure 7.1: System overview with $C \times M \times N = 4 \times 3 \times 3$ tiles. Marked in blue are hardware block for the multi-chip systolic extension including the border interface which orchestrates any write and read to the border and corner memories and distributes it to the Data Distribution Units (DDUs). Furthermore, it sends and receives calculated pixels to and from the chip neighbors.
is connected to its 8 spatial neighboring Tile-PUs (i.e., directly adjacent Tile-PUs) to quickly access neighboring pixels.

- **Weight Buffer (WBuf):** Stores the weights of the current $C$ output FMs.

- **Data Distribution Units (DDUs):** Distributes the data from the memories to the corresponding Tile-PU units or manages zero-padding.

- **Border and Corner Memory BM, CM:** Storage for pixels which are part of neighboring chips.

- **Border Interface (BI/F):** Sends and receive border pixels from/to neighboring chips and stores pixels into Border and Corner Memories.

The superior efficiency of Hyperdrive is achieved exploiting data re-use at different levels:

- **Output FM level:** The output FMs are tiled into blocks of $C$ FMs which are calculated at the same time in the depth-wise parallel Tile-PUs which allows to load the input FMs just once for $C$.

- **Spatial level:** The input FM is tiled into $M \times N$ equally-sized image patches and calculated in parallel in the $M \times N$ spatial processing units illustrated in Fig. 7.3. Weights are read once from off-chip memory only and used to calculate all $M \times N$ partial sums for the corresponding tiles.

- **Weight re-use:** Weights are stored in the weight buffer, which is implemented as a latch-based standard cell memory for optimal energy efficiency [81].

- **Border re-use:** Border pixels are transmitted only once to the corresponding neighbor chip and stored in its Border and Corner Memory instead of reading every time.
Figure 7.2: *Tile Processing Units (TPUs)* of same spatial tile (*Tile-PU*\(_{(x,y)}\))*: Every single Tile-PU (i.e., 4 shown in figure) provides a FP16 adder, accumulation register and ReLU activation unit. There is one time-shared FP16 multiplier per spatial tile and shared among the \( C = 4 \) Tile-PUs in the depth dimension, indicated by the dots. The FMs are calculated in a interleaved way for all \( C \) output dimensions. The (single-bit) binary weight is applied as the sign input for the FP16 adder.

Figure 7.3: The feature maps are tiled and processed in parallel Tile-PUs.
7.3 Computational Model

State-of-the-art CNNs like ResNet-34 impose high demands in computational complexity and memory for the large space of parameters and intermediate Feature Maps. However, for BWNs, streaming the weights rather than the FMs or both is particularly attractive due to the compression by $16 \times$ (i.e., from FP16).

CNNs are composed of several neural network layers, whereas the main building block are Convolution Layers which can be formulated as a mapping from the 3D input Feature Map space (i.e., $\text{FM}^{\text{in}}$) of $n_{in}$ channels with $h_{in} \times w_{in}$ sized spatial dimensions to the 3D output Feature Map space (i.e., $\text{FM}^{\text{out}}$) of $n_{out} \times h_{out} \times w_{out}$ size and can be described as follows:

$$\mathbb{R}^{n_{in} \times h_{in} \times w_{in}} \xrightarrow{\text{CNN}} \mathbb{R}^{n_{out} \times h_{out} \times w_{out}}$$

$$\text{FM}^{\text{out}} \mapsto \text{FM}^{\text{in}} \text{ s.t.}$$

$$\text{FM}^{\text{out}}(c_{out}, \cdot, \cdot) = \beta_{c_{out}} + \alpha_{c_{out}} \sum_{c_{in} \in I_{n_{i}}} \text{FM}^{\text{in}}(c_{in}, \cdot, \cdot) * k_{c_{out}, c_{in}}(\cdot, \cdot)$$

Every single output channel $c_{out}$ is calculated by convolving all input feature maps $c_{in}$ with the corresponding filter kernel $k_{c_{out}, c_{in}} \in \mathbb{R}^{h_{k} \times w_{k}}$, scaled by the factor $\alpha_{c_{out}}$ and accumulated to a bias term $\beta_{c_{out}}$. It should be noted here, that Batch normalization which are quite common after convolution layers, can be merged with biasing and scaling, as the coefficients stay constant after training.

7.3.1 Binary Weights for Residual Networks

Residual Networks have been introduced to allow better generalization in deep neural networks and showed the first super-human performance on the challenging ImageNet challenge [8]. The networks are composed of either basic residual blocks built from two subsequent $3 \times 3$ CNN layers or bottleneck blocks built from subsequent CNN layers of kernel sizes $1 \times 1$, $3 \times 3$, and $1 \times 1$. Both blocks have in common to have an additional datapath bypassing the CNN layers and being accumulated
7.3. **COMPUTATIONAL MODEL**

Figure 7.4: Early block of layers of ResNet-34 and transition to next type of layer block for C-Type Bypasses. Activation and batch normalization layers are not indicated separately. Dashed rectangles imply on-the-fly addition to eliminate the need for additional memory.

with the output feature maps from the CNN layers. Furthermore, within the blocks max-pooling layers are introduced, which half the feature maps in both spatial dimensions (e.g., $x$ and $y$), while the number of channels is doubled. As the max-pooling is done within the CNN layers, their FM dimensions differ from the bypass datapath and has to be treated specially. He et al. [8] introduces therefore three bypass version A, B, C and we suggest another version D, which are as follows:

A: Identity (if same dimension), (Strided) Identity + Zero-Padding in FM dimension

B: Identity (if same dimension), Spatial 1x1 Convolution Layer

C: Spatial 1x1 Convolution Layer

D: Identity (if same dimension), (Strided) Identity + Spatial 1x1 CNN layer

We trained several versions of ResNet with bypass types A to D with binary weights and 16-bit fixed-point activation values with the Stochastic Gradient Descent (SGD) algorithm with a momentum of 0.9 and a learning rate of 0.01 for the first 18 epochs, and fixed decaying scheme afterwards, and the results are shown in Tab. 7.1.

Fig. 7.5 shows the training curve of BWN-ResNet-34 with bypass type D; It can be seen, that the best result has been shown with the
Table 7.1. Comparison of different Bypass Variants for Binary-Weight ResNets

<table>
<thead>
<tr>
<th>Network</th>
<th>act./wght.</th>
<th>Test Top-1</th>
<th>Test Top-5</th>
<th>Train Top-1</th>
<th>Train Top-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resnet-18B</td>
<td>INT16/1</td>
<td>57.20</td>
<td>80.84</td>
<td>61.48</td>
<td>83.77</td>
</tr>
<tr>
<td>Resnet-34A</td>
<td>INT16/1</td>
<td>58.90</td>
<td>81.91</td>
<td>64.33</td>
<td>85.85</td>
</tr>
<tr>
<td>Resnet-34B</td>
<td>INT16/1</td>
<td>61.66</td>
<td>83.93</td>
<td>66.98</td>
<td>87.43</td>
</tr>
<tr>
<td>Resnet-34C</td>
<td>INT16/1</td>
<td>61.57</td>
<td>83.97</td>
<td>66.77</td>
<td>87.29</td>
</tr>
<tr>
<td>Resnet-34D</td>
<td>INT16/1</td>
<td>63.09</td>
<td>85.11</td>
<td>70.94</td>
<td>89.44</td>
</tr>
<tr>
<td>Resnet-50B</td>
<td>INT16/1</td>
<td>61.61</td>
<td>84.03</td>
<td>65.66</td>
<td>86.72</td>
</tr>
<tr>
<td>Resnet-34B [8]</td>
<td>FP32</td>
<td>78.16</td>
<td>94.29</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Resnet-50B [8]</td>
<td>FP32</td>
<td>79.26</td>
<td>94.75</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 7.5: Accuracy plot for BWN-ResNet-34D
D-type ResNet-34 which reaches a 85.1% of Top-5 accuracy which is a 9.2 drop in accuracy, but more interestingly it reaches better 1.2% better results, than with the commonly used B-type ResNet, whereas the D-type even needs less computations as the Convolution is just applied to half of the feature map and the other half part is identical to the input feature map.

7.3.2 Principles of Operation

The operations scheduling is summarized in Algorithm 6 and illustrated in Tbl. 7.2 for an implementation of the architecture featuring $C \times M \times N = 16 \times 7 \times 7$ Tile-PU with 8×8 sized spatial tiles $\tilde{p}$ and for a 3×3 convolution layer with 16×64 FMs, whereas the output channels are tiled into blocks $\tilde{c}_{out}$ of $C = 16$ channels. After the entire input feature map is loaded into the FMM, the system starts inferring the network. The output FM-level and spatial parallelism is indicated in lines 2 and 3, whereas every Tile-PU is working on its assigned spatial tile $\tilde{p}$ and output channel tile $\tilde{c}$.

Then in the inner loop, the contribution for all pixels from the corresponding tile and output channel are calculated. From the streaming approach, a logical approach would be to load the weights and apply it to the entire patch for every Tile-PU, unfortunately, the patches can be large, and this introduces frequent writes and reads to random-access memory (FMM), instead the weights streamed to the chip are stored in a weight buffer (Line 11) which can be implemented in a small memory (i.e., latch-based memory for low energy) and where the weights for the current $C$ output channels (of all input channels) are stored. In this way, we avoid writing and re-fetching intermediate FM values.

The pixels are then calculated by iterating through all filter points (e.g., 9 in 3×3 kernels) and input channels $c_{in}$ (lines 7 and 8). On each cycle one binary weight per parallel feature map dimension $\#\tilde{c}_{out}$ is loaded from the weight buffer (Line 14) and input Feature Map pixel per spatial tile ($\#\tilde{p} = \#\{\text{Tile-PU} s\} = M \cdot N$) are loaded from the FMM (Line 16). All the Tile-PU's access either their own FMM bank in case that the feature $p + \Delta$ (for the filter tap $\Delta$, e.g., (-1,-1) for the top-left weight of a 3×3 filter) lies in the same tile $\tilde{p}$ or from the
Table 7.2. Time schedule for a 16 input FM and 64 output FM 3×3 convolution. Notation for filter weights: $f_{\text{filter tap}(\Delta y, \Delta x)}^{\text{input FM}, \text{output FM}}$.

<table>
<thead>
<tr>
<th>cycle</th>
<th>1</th>
<th>2</th>
<th>16</th>
<th>17</th>
<th>144</th>
<th>145</th>
<th>288</th>
<th>9216</th>
<th>9217</th>
<th>36.8k</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight input</td>
<td>$f_{-1,-1}$, $f_{-1,0}$, ...</td>
<td>$f_{1,-1}$, $f_{1,0}$, ...</td>
<td>$f_{-16,-1}$, $f_{-16,0}$, ...</td>
<td>No I/O (loaded from weight buffer)</td>
<td>$f_{16,-1}$, $f_{16,0}$, ...</td>
<td>No I/O</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>input FM</td>
<td>1</td>
<td>2</td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>filter tap pos.</td>
<td>-1,-1</td>
<td>-1,0</td>
<td>...</td>
<td>+1,+1</td>
<td>-1,-1</td>
<td>...</td>
<td>+1,+1</td>
<td>-1,-1</td>
<td>...</td>
<td>+1,+1</td>
</tr>
<tr>
<td>outp. pixel pos.</td>
<td>1,1</td>
<td>1,2</td>
<td>...</td>
<td>8,8</td>
<td>1,1</td>
<td>...</td>
<td>8,8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>output FM</td>
<td>1-16 (in parallel)</td>
<td>17-32</td>
<td>...</td>
<td>63-64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

corresponding FMM bank of the corresponding neighboring Tile-PU. All these accesses are aligned (e.g., all the Tile-PUs are reading the FMM bank of their corresponding top-left neighbor) and therefore no access conflicts occur. The weights are multiplied with the binary weights: this is implemented as a change of sign and then accumulated with the previous partial sum $v$ (Line 17). When all contributions for all input channels and filter taps have been accumulated, a scaling factor (e.g., from batch normalization) is applied to it (Line 21), bypass is added (Line 22) and finally the channel bias is added (Line 23), before it is written back to the feature map memory (Line 24).

Bypass paths are common in several CNNs like ResNet-34 and are shown in Fig. 7.4. As will be explained in the next section, the bypass can be read, added to the partial sum and stored back to the same memory address avoiding additional memory for the bypass FM. Unfortunately, this does not work in the same cycle, therefore adding the bias (Line 21) has been moved after the bypass (Line 20) and stalling can be avoided.

### 7.3.3 CNN Mapping

The size of the on-chip memory for intermediate FM storage has to be selected depending on the convolution layer with the largest memory footprint of the network, hereinafter referred as **Worst-Case Layer**. Typically, the Worst-Case Layer is at the beginning of the network, since a common design pattern is to double the number of FMs after a few layers while performing at the same time a $2 \times 2$ strided operation, thereby reducing the number of pixels by $4 \times$ and the total FM volume by $2 \times$. To perform the computations layer-by-layer, avoiding usage of power hungry dual-port memories, we leverage a
Algorithm 6 Hyperdrive Execution-Flow

Require: All input feature maps in FMM$^\text{in}$
Require: Weight Stream
1: for all $M \times N$ pixel tiles $\tilde{p}$ (in parallel HW units) do
2:   for all $C$ output channel tiles $\tilde{c}_{\text{out}}$ (in parallel HW units) do
3:       Tile-PU for output channel tile $\tilde{c}_{\text{out}}$ and pixel tile $\tilde{p}$
4:       def readFMfromMemory
5:       for all output channel $c_{\text{out}}$ in tile $\tilde{c}_{\text{out}}$ do
6:           $v = 0$
7:           for all pixel $p = (y, x)$ in tile $\tilde{p}$ do
8:               for all filter points $\Delta = (\Delta y, \Delta x)$ with
9:                   $\Delta y = -\lfloor \frac{h_k}{2} \rfloor, \ldots, -1, 0, 1, \ldots, \lfloor \frac{h_k}{2} \rfloor,$
10:                  $\Delta x = -\lfloor \frac{w_k}{2} \rfloor, \ldots, -1, 0, 1, \ldots, \lfloor \frac{w_k}{2} \rfloor$ do
11:                     for all input channel $c_{\text{in}}$ do
12:                         if $w[c_{\text{in}}, c_{\text{out}}, \Delta] \notin \text{WBuf}$ then
13:                             $k_{c_{\text{out}}, c_{\text{in}}}(\Delta) = \text{wghtStrm}$
14:                         end if
15:                         $w = \text{WBuf}[c_{\text{in}}, c_{\text{out}}, \Delta]$ (read of $\#\tilde{c}_{\text{out}}$ bit)
16:                         // Aligned read of FMM$^\text{in}[p + \Delta, c_{\text{in}}]$ from //
17:                         // corresponding memory bank (either from // its own
18:                         // memory bank or the correspond- // ing neighbor’s
19:                         // bank).
20:                         $x = \text{FMM}^\text{in}[p + \Delta, c_{\text{in}}]$ (read of $\#\tilde{p}$ words)
21:                         $v = (v + x \cdot w) = \begin{cases} v + x & \text{if } w = 1 \\ v - x & \text{otherwise} \end{cases}$
22:                     end for
23:                 end for
24:             end for
25:         end for
26:     end for
27: end for
ping-pong buffer mechanism reading from one memory bank and writing the results to a different memory bank. Hence, for a generic CNN the amount of memory required by the Worst-Case Layer is: \( \max_{\text{layers in CNN}} n_{\text{in}}h_{\text{in}}w_{\text{in}} + n_{\text{out}}h_{\text{out}}w_{\text{out}} \) words, since all input and output FMs have to be stored to implement the described ping-pong buffering mechanism.

However, many networks have bypass paths, hence additional intermediate FMs have to be stored, as described in Fig. 7.4a for the potential Worst-Case Layers of ResNet-34. This aspect has two implications:

1. In order to avoid additional memory (+50%), we perform an on-the-fly addition of the bypass path after the second 3\( \times \)3 convolution (i.e., the dashed rectangle is a single operation). This is done by performing a read-add-write operation on the target memory locations.

2. To avoid adding a stall cycle when reading and writing to the same memory area within the same cycle, the bias adding is moved after the bypass such that the following order is followed: convolution, scale, bypass, bias, store back. In this way, the data can be read from memory address and stored back to the same address with one cycle latency.

3. The common transition pattern with the 2\( \times \)2-strided convolution does not require additional memory. It temporarily needs three memory segments, but two of them are 2\( \times \) smaller and can fit into what has been a single memory segment before (M2 is split into two equal-size segments M2.1 and M2.2).

In the following section, the calculation of the Worst-Case Layer for ResNet-like networks with basic bypass blocks is discussed in detail and numbers are presented for ResNet-34, but does not limit the execution of networks with smaller Worst-Case Layer. To reduce off-chip data communication to a minimum, we will split the Worst-Case Layer in memory segments M1, M2, ... to indicate which data needs to be kept in on-chip memories at the same time. Hyperdrive always operates on a single convolutional layer at a time and is iterating
7.3. **COMPUTATIONAL MODEL**

several times over the entire input FM which therefore needs to be stored on-chip in memory section $M_1$. The same is valid for the output FM which is calculated and stored in $M_2$, respectively.

There are $n_{out}$ output channels which have a $h_{out} \times w_{out}$ sized output FM. These output FMs are calculated as sum of convolutions of every $n_{in}$ input channel (with FMs size of $h_{in} \times w_{in}$) on the $h_k \times w_k$ sized filter kernels $w_{k,n}$.

For a normal convolution layer,

$$M = M_1 + M_2 = n_{in} \cdot h_{in} \cdot w_{in} + n_{out} \cdot h_{out} \cdot w_{out} \ [\text{words}]$$

need to be stored, because the entire input FM is needed to calculate every single output FM.

In a next step, the special case with residual bypasses is evaluated like in ResNet [8] and similar residual networks. ResNet has two different types of residual blocks: the basic building block and the bottleneck building block. The basic building block is presented in Fig. 7.4.

Within the basic building block, there are two different cases, the first is $n_{in} = n_{out}$ where there is no striding, thus also $h_{in} = h_{out}$ and $w_{in} = w_{out}$. The input FM to the residual block will then be placed in the virtual memory section $M_1$ and Hyperdrive computes the first $3 \times 3$ convolution layer and writes the results into section $M_2$, secondly Hyperdrive calculates the second convolutions layer reading from $M_2$ and accumulating the output FM with the bypassed values in $M_1$ on-the-fly and writing them back to $M_1$. A total amount of 401 kwords need to be stored.

$$M = M_1 + M_2 = 2 \cdot M_1 = 2n_{in} \cdot h_{in} \cdot w_{in}$$

$$M_1 = M_2 = n_{in} \cdot h_{in} \cdot w_{in}$$

$$M_{max} = 2n_{in} \cdot h_{in} \cdot w_{in} = 2 \cdot 64 \cdot 56 \cdot 56 = 401k\text{words}$$

In case of down-sampling the number of output channels is doubled $n_{out} = 2n_{in}$ and the image sizes are reduced by $4 \times$ to $h_{out} \times w_{out} = \frac{1}{2} h_{in} \times \frac{1}{2} w_{in}$. Also, the bypass needs to be strided. He et al. suggest to either use the strided identity or to perform $1 \times 1$ strided convolution,
we will consider this case as it is more memory critical than with subsampling [8]. The input FM is read from M1 and the $3 \times 3$ strided convolution is performed and saved in M2, then the $1 \times 1$ strided convolution on the bypass is evaluated and saved in M3, finally the 2nd convolution layer is performed on the data in M2 and accumulated to the strided bypass in and to M3. It can be shown, that M2 and M3 are a quarter of the size of M1 and 301 kwords are needed for the three memory sections.

\[
M = M1 + M2 + M3 = 1.5 \cdot M1 \\
M1 = 1.5n_{in} \cdot h_{in} \cdot w_{in} \\
M2 = M3 = 2n_{in} \cdot 0.5 \cdot h_{in} \cdot 0.5 \cdot w_{in} = 0.5 \cdot M1
\]

Due to the reduced size of the FM after every subsampling, just the first residual block need to be considered for dimensioning the memories. For ResNet-18 and ResNet-34, this translates to 401 kwords which are 6.4 Mbit with FP16.

Deeper residual networks (e.g., ResNet-50) are composed of the bottleneck building block (illustrated in Fig. 7.4b), to evaluate the Worst-Case Layer, there are two cases to consider: with and without subsampling. In the first case, the input FM is stored in M1 and needs to be stored for the entire bottleneck block. The output FM for the first $1 \times 1$ convolution layer is stored in M2 and is $4 \times$ smaller due to the $4 \times$ smaller number of channels, then the $3 \times 3$ convolution layer calculates its features from M2 to M3 and the second $1 \times 1$ convolution layer is calculated on-the-fly adding to the bypass FM.

\[
M = M1 + M2 + M3 = 1.5 \cdot M1 \\
M1 = 1.5n_{in} \cdot h_{in} \cdot w_{in} \\
M2 = M3 = \frac{n_{in}}{4} \cdot h_{in} \cdot w_{in} = 0.5 \cdot M1
\]
7.3. COMPUTATIONAL MODEL

In total 1.5× of the input FM size is needed to evaluate the bottleneck block without subsampling. In case with subsampling, already after the 1×1 convolution, the bypass needs to be evaluated which is another 1×1 convolution which we can map into M4 memory. Instead of writing the feature map for the 3×3 convolution to M3, it can be written to M1, because this data is not needed any more. The 2nd 1×1 convolution is then calculated on the fly from M1 and M4 back to M1.

\[
M = M1 + M2 + M4 = 1.675 \times M1
\]
\[
= \frac{13}{8} n_{in} \cdot h_{in} \cdot w_{in} = 1.2M \text{ words}
\]
\[
M1 = \max \left( n_{in} \cdot h_{in} \cdot w_{in}, \frac{2n_{in}}{4} \cdot \frac{h_{in}}{2} \cdot \frac{w_{in}}{2} \right)
\]
\[
= n_{in} \cdot h_{in} \cdot w_{in}
\]
\[
M2 = \frac{2n_{in}}{4} \cdot \frac{h_{in}}{2} \cdot \frac{w_{in}}{2} = 0.125 \cdot M1
\]
\[
M4 = \frac{2n_{in}}{2} \cdot \frac{h_{in}}{2} \cdot \frac{w_{in}}{2} = 0.5 \cdot M1
\]

This leads to a Worst-Case Layer of 1.2 Mword or 19.2 Mbit (Conv2) for ResNet-50/-152/... independently of the depth which would be 6.3 mm² of SRAM (0.3 µm²/bit in GF 22nm FDX).

7.3.4 Supported Neural Network Topologies

In the previous section, we have discussed the requirements to map the different ResNet-style networks onto Hyperdrive. For its implementation, we have parameterized the architecture to fit the feature maps of ResNet-34 on-chip. Nevertheless, Hyperdrive is neither restricted to these networks nor these applications—in fact, its scalability to multiple chips to process high-resolution images for object detection and image segmentation is a key feature of its architecture. For example, running the feature extraction for object detection using
Table 7.3. Data Comparison for various typical networks with binary-weights and 16-bit FMs for single-chip implementation considering single-chip implementation (Top: Image Recognition, Bottom: Object Detection)

<table>
<thead>
<tr>
<th>Network</th>
<th>Resolution</th>
<th>Weights [bit]</th>
<th>All FMs [bit]</th>
<th>WC Mem. [bit]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-18</td>
<td>224×224</td>
<td>11M</td>
<td>36M</td>
<td>6.4M</td>
</tr>
<tr>
<td>ResNet-34</td>
<td>224×224</td>
<td>21M</td>
<td>61M</td>
<td>6.4M</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>224×224</td>
<td>21M</td>
<td>156M</td>
<td>21M</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>224×224</td>
<td>55M</td>
<td>355M</td>
<td>21M</td>
</tr>
<tr>
<td>ResNet-34</td>
<td>2048×1024</td>
<td>21M</td>
<td>2.5G</td>
<td>267M</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>2048×1024</td>
<td>55M</td>
<td>14.8G</td>
<td>878M</td>
</tr>
</tbody>
</table>

YOLOv2 [255] is supported by Hyperdrive. For the worst-case layer in terms of memory when processing 448×448 pixel frames, we would need to be able to store 3.2 M words—scaling up the memory by 2× over the ResNet-34 parameterization would be sufficient to even run it even on a single chip, and for higher resolutions the workload and memory for the feature maps could be split across multiple chips as described in Sec. 7.4. Also, the Fire module of the size-optimized SqueezeNet [37] and SqueezeDet [9] topologies is supported by Hyperdrive. The grouped convolutions and shuffling operations present in MobileNetV2 [38] and ShuffleNet [39] can also be applied with the presented architecture. Also the not very common depth-wise separable convolutions present in some layers of MobileNetV2 can be computed using Hyperdrive, although not at maximum performance due to limited bandwidth of the on-chip SRAMs (no local re-use of the input feature map data possible).

The only limitation is that several networks feature a first convolution layer with an exceptionally large kernel size (e.g., 7×7 convolution for both ResNet and YOLOv2, but making up less than 2% of all operations). As Hyperdrive supports only 1×1 and 3×3 convolution layers, this first layer has to be computed off-chip before loading the data into Hyperdrive, or a small dedicated on-chip accelerator for the
first layer could be included, which would perform these operations as the feature maps are streamed into the device. Networks optimized for compute effort, such as TinyYOLO [256] or MobileNetV2 [38], are often only composed of $3 \times 3$ and $1 \times 1$ convolution layers and do not have such a first filter with an exceptionally large kernel size.

7.4 Scalability to Multiple Chips

Even though, we could show that the architecture is in theory scalable to any sized networks, the Worst-Case Layer is setting a real-world limit to it. Already ResNets with bottleneck layer require 19.2 Mbit\(^1\) to perform inference on small $224 \times 224$ sized images and larger images (e.g., in typical object detection tasks) need 10s or 100s of Mbit. This clearly exceeds the area constraints of few Mbit in low-cost chip fabrication due to high production costs and diminished production yield. A very natural solution is to extend the systolic architecture to multiple chips, in this way the feature map is first tiled on an array of $m \times n$ Hyperdrive chips and further tiled within each chip on their $M \times N$ Tile Processing Units, such that $M \cdot m \times N \cdot n$ tiles are operated in parallel.

\(^1\)Note that the Worst-Case Layer for ResNet-like networks does not depend on depth, but on the size of the images (e.g., $224 \times 224$) and the building blocks (basic bypass in Fig. 7.4a or bottleneck in Fig. 7.4b). See also Tbl. 7.3 for a comparison of the Worst-Case Layers.
CHAPTER 7. HYPERDRIVE: I/O BOTTLENECK IN BWN

Figure 7.7: Multi-chip Considerations: a) Intra-chip connection: 1 output interface and 4 inputs from/to 4 direct neighbors, b) Border Memory and Corner memory access with address block \( (c_{in} = 1, h_k = w_k = 3) \) for every single cycle c) Access pattern in case of a corner access: two reads from Border Memory (top and left) and one read from Corner Memory d) Chip Types in a systolic chip setting (North West to South East and Center chip)

Figure 7.6: Memory Allocation in the multi-chip setup with \( 1 \times 1 \) sized tiles for \( 3 \times 3 \) sized kernels. The \( M \times N \) “core” tiles and pixels are stored in the FMM and the pixels located and calculated in the chip neighbor are stored in Border and Corner Memory. The Border Memory stores these \( M \times \) or \( N \times \) pixels (i.e., \( 7 \times 16 \) – bit) which can be accessed in the same cycle.

Similarly to the single-chip setup, the Tile Processing Units need to access neighboring pixels, but in the multi-chip setup they might
even lie on a different chip instead of just another tile memory. Three solutions are possible in this case:

1. the input feature maps of all chips are padded with the missing pixels, but this is not reasonable for deep networks as the padding increases steadily with the number of layers.

2. The border pixels are read from the neighboring chips when they are used, but this introduces high bandwidth requirement, as these pixels are needed several times or

3. the border pixels are sent once after they have been calculated to the neighboring chips and stored locally there.

Hyperdrive implements option 3 which introduces two additional memories: A Border Memory BM and Corner Memory CM and have been added to the general architecture of Hyperdrive in Fig. 7.1.

Fig. 7.6 illustrates the locations of the pixels from a chip perspective and Fig. 7.7a shows the perspective of a single chip connected to its neighboring chips which are overall arranged in a systolic way. Pixels residing in the border of the neighboring chips are stored in the Border Memory and pixels residing in the corners of the diagonal neighboring chips are stored in the Corner Memory and are read from there in case border pixels are requested by the computational model.

### 7.4.1 Access Pattern and Storing Scheme of the Border Memories

Fig. 7.7c illustrates the pixels and their memory location which are read in case of a corner pixel and Fig. 7.7b for all cases of access top border pixels. When border pixels but not corner pixels have to be accessed, one pixel per corresponding border Tile-PUs is read and stored into the same memory block. In case of a corner, actually $M - 1$ and $N - 1$ pixels from two border sides (i.e., one vertical and one horizontal) and one corner pixel. Therefore, the border memory is split into two physically separated memory blocks allowing to read from both sides without the need of two-port memories or introducing any latency or stalls. Furthermore, chips are assigned a location chip type,
which indicates which part of the feature map the chip is working on. They have been named according to cardinal orientation: corner chips (NW, NE, SW, SE), border chips (N, W, E, S) and Center like illustrated in Fig. 7.7d. All chips sharing the same orientation work identically and synchronized, thus the exact position does not matter.

### 7.4.2 Border and Corner Exchange

Whenever a border pixel (e.g., N border) has been calculated, it is sent to the corresponding neighbor (i.e., south neighbor) and a flag is set indicating that it is waiting the same kind of pixel from its opposite neighbor (i.e., north neighbor).

When a corner pixel (e.g., NW) is calculated, the pixel needs to be sent to all three neighboring chips in the corresponding direction (N, W, NW). As the number of these pixels is small and to keep the inter-chip wiring small, no additional diagonal interfaces are introduced, but these pixels are forwarded by the corresponding vertical neighbor (N) to the diagonal chip (NW). Additionally, there are for every corner 2 additional flags which are set in the Border Interface: one for the forwarding chip (sending, N) and the receiving chip (NW).

### 7.4.3 Border and Corner Memory

There are two different access patterns. If a corner pixel is accessed, the corner pixel, \(N-1\) vertical pixels (left or right) and \(M-1\) horizontal pixels (top or bottom) and one pixel need to be read from the corner memory, which is illustrated in Fig. 7.7c. In the other border cases, they are either \(N\) vertical pixels or \(M\) horizontal pixels (e.g., in Fig. 7.7b at \(t \in \{1, 2\}\)). Therefore, the border memory can be seen as a horizontal or vertical extension to the FMM and \(N\) and \(M\) words can be read in a single cycle. As for the FMM, splitting the border memory into two physically separated memory blocks allows to read from both in the same cycle without introducing any additional latency. The memory needs to fit the overlapping border of the Worst-Case Layer whereas the width depends on the kernel width of the current and next layer. The overlapping rows or columns are \(\lfloor \frac{h_k}{2} \rfloor\) or \(\lfloor \frac{w_k}{2} \rfloor\) wide.
and can be determined directly from the Worst-Case Layer evaluation for FMM by dividing the spatial area and multiplying by the sum of all overlapping border rows or columns (which might differ for input and output FM). In case of ResNets with the basic building block (e.g., ResNet-34). The required memory for the left, right, top and bottom border (i.e., $M_{b,left}$, $M_{b,right}$, $M_{b,top}$, $M_{b,bottom}$) can therefore be calculated as follows:

$$M_{border} = M_{b,left} + M_{b,right} + M_{b,top} + M_{b,bottom} = M \cdot \frac{2h_{in} + 2w_{in}}{h_{in} \cdot w_{in}} = M \cdot \frac{2 \cdot 56 + 2 \cdot 56}{56 \cdot 56} = 459 \text{kbit}$$

$$M_{b,left} = M_{b,right} = 2 \left( n_{in} w_{in} \left\lfloor \frac{w_{k,l}}{2} \right\rfloor + n_{out} w_{out} \left\lfloor \frac{w_{k,l+1}}{2} \right\rfloor \right)$$

$$M_{b,top} = M_{b,bottom} = 2 \left( n_{in} h_{in} \left\lfloor \frac{h_{k,l}}{2} \right\rfloor + n_{out} h_{out} \left\lfloor \frac{h_{k,l+1}}{2} \right\rfloor \right)$$

which is an increase of 7% of overall memory.

The Border Memory (as indicated in Fig. 7.1) is then implemented with 4 high-density single-port SRAMs with 1024 lines of $7 \cdot 16 = 112$.

The Corner Memory needs to store the diagonally overlapping pixels, which are $\left\lfloor \frac{h_{k}}{2} \right\rfloor \cdot \left\lfloor \frac{w_{k}}{2} \right\rfloor$ sized patches. In contrary to the discussions regarding the FMM and BM, the Corner Memory does not profit from striding such that for ResNet typed networks the last layer has the highest memory demand. Overall it can be dimensioned for ResNet-34 as $(n_{in} + n_{out}) \cdot 4 \left\lfloor \frac{h_{k}}{2} \right\rfloor \cdot \left\lfloor \frac{w_{k}}{2} \right\rfloor = 2 \cdot 512 \cdot 4 \cdot 1 \cdot 1 \cdot 16 \text{bit} = 64 \text{kbit}$ which is another 1% increase of overall memory. This memory has been implemented with a single-port memory of 4096 of 16-bit words.

### 7.4.4 Interface Implementation

During the computation of border pixels, every border Tile-PU sends and receive the pixels to/from the respective Border Interfaces. The border interfaces, placed on the 4 sides (as illustrated in Fig. 7.7a) of the accelerator, are responsible for buffering and transmitting pixels from/to the neighboring chips, synchronizing execution of the Tile-PUs as well. For vertical and horizontal borders there is one $m \cdot C = 7 \cdot 16 =$
112 entries buffer. When the buffer is non-empty, the border interface sends these pixels in an interleaved way and split into blocks of 4 bits and 1 valid bit to the neighbors. Every chip itself has 4 in-coming serial interfaces from the directly adjacent neighbors (i.e., N, S, W, E). When data is received, it is de-serialized, recovered in its original 16-bit format and stored in the border/corner memories. The interfaces are also responsible for calculating the addresses of pixels received and transmitted from/to neighboring chips in the border memory. Fig. 7.2 shows in blue the extension needed for exchanging the borders between the chips with 1 out-going and 4 in-going intra-chip interfaces.

7.5 Experimental Results

The number of tiles has been chosen to be $M \times N = 7 \times 7$, which allows for $4 \times$ striding on $112 \times 112$ sized input FMs (like in common ResNet-like networks), while keeping all the TPUs busy with at least one single spatial pixel during the entire network. We use the half-precision floating point (FP16) number format for the FMs as a conservative choice to ensure loss-less inference even for deeper networks [257, 258]. Fixed-point or other alternative formats [259] could be used to reduce the energy cost of the arithmetic operations. Fixed-point arithmetic units featuring a smaller number of bits (e.g., 8) would linearly impact the size of the on-chip memory for the FMs. By using FP16, the final accuracy is determined by the selected network and the corresponding BWN training algorithm. A ResNet-18 trained on the ImageNet dataset can run on Hyperdrive with a 87.1% top-5 accuracy using the SBD-FQ training method [260] (full-precision top-5 accuracy: 89.2%).

The on-chip memory was sized to fit the Worst-Case Layer of ResNet-34 with 6.4 Mbit (400 kword) and is implemented with $M \times 8 = 7 \times 8$ high-density single-port SRAMs with 1024 lines of $N \cdot 16 = 7 \cdot 16 = 112$-bit words, whereas the memories are assigned to the $(M \times N)$ tiles. The output FM parallelism has been fixed to $C = 16$. The weight buffer has been implemented to fit up to 512 (max. #input FMs) $h_k \times w_k = 3 \times 3$ kernels for $16 \times$ depth-wise parallelism. If more input FMs are needed, they can be tiled to 512 blocks and partial output FM can be calculated and summed up on-the-fly using the bypass mode. The frequently-accessed weight buffer has been implemented as
7.5. EXPERIMENTAL RESULTS

![Floorplan with Weight Buffer, Feature Map Memory and Tile Processing Units](image)

Figure 7.8: Floorplan with Weight Buffer, Feature Map Memory and Tile Processing Units (left) and photograph of the taped-out multi-project chip Poseidon\(^1\) with Hyperdrive on the bottom side (right).

...a latch-based standard cell memory (SCM) composed of 5×8 blocks of 128 rows of 16-bit words, reducing the access energy to SRAM memories by 43× [81]. It should be noted that even though the energy efficiency of SCMs are much better than SRAMs, they are also up to 8× larger in area which limits this kind of memories to comparably small buffers (i.e., weight buffer), but not for the feature map memory.

### 7.5.1 Implementation Results

Hyperdrive was designed in GF 22 nm FDX technology using an 8 track low voltage threshold (LVT) standard cell library. This flavor of the technology allows to apply up to 1.8V of forward body biasing (FBB), increasing the operating frequency of the chip at the cost of higher leakage power. Synthesis was performed with Synopsys Design Compiler 2017.09, while place & route was performed with Cadence Innovus 17.11.

The chip has an effective core area of 1.92 mm\(^2\) (=9.6 MGE)\(^2\), where 1.24 mm\(^2\) are SRAM memories (6.4 Mbit), 0.115 mm\(^2\) are SCM memory (74 kbit) and 0.32 mm\(^2\) arithmetic units. Fig. 7.8 shows on

---

\(^1\)Hyperdrive was taped-out alongside of two different projects (Kerbin and Quentin) on the same die to share costs, details can be found on [http://asic.ethz.ch/2018/Poseidon.html](http://asic.ethz.ch/2018/Poseidon.html)

\(^2\)One 2-input NAND gate equivalents (GE) is 0.199 µm\(^2\) in GF22.
the right side a photograph of the actual chip and on the left side Hyperdrive’s floorplan.

Testing and characterization (frequency, power) of silicon prototypes were performed on the industry-standard ASIC tester Advantest SoC V93000 and core power are based on the real chip measurements. The I/O energy was determined on the basis of an LPDDR3 PHY implemented in 28 nm technology [208], estimated as 21 pJ/bit, as in context of our research no low-swing interface IP blocks were available. It should be noted that this has to be considered as quite optimistic bound for I/O energy in a low-cost chip (the LPDDR3 PHY is quite complex and expensive), hence pessimistic for the proposed architecture focusing on system-level energy efficiency and specifically I/O bandwidth reduction. If we use low-cost low-complexity full-swing I/O interfaces (used for the implementation of this prototype, and of the other state-of-the-art accelerator [68, 72, 73, 81]) would further magnify the system-level energy gain of Hyperdrive with respect to other architectures, but would probably give too much advantage to our solution with respect to industrialized, production-ready scenario where low-swing I/O interfaces would be used [75].

Fig. 7.11 provides an overview of the Hyperdrive’s blocks power consumption at the operating voltage of 0.5 V and 58 MHz. The power consumption of memory arrays, memory periphery and logic were measured on the silicon prototype, available through the multi-power rails implementation strategy. On the other hand, the breakdown of the remaining standard cell logic power contributions is split into Tile-PUs, Weight Buffer and Others and has been estimated with post-layout simulations. It is interesting to note that a considerable amount of the power is consumed into the arithmetic units, while only a small overhead comes from memory accesses and I/Os, due to the efficient exploitation of Feature Map stationary (i.e., temporal locality) of the Hyperdrive architecture, explaining its superior system-level energy efficiency with respect to the other BWN accelerators in Tbl. 7.6. The main features of the chip in other operating points is reported in Tbl. 7.5

In order to characterize the best energy point of the chip we swept the body bias of the system along the available range (i.e., from 0 V to 1.8 V), as shown in Fig. 7.9. It is interesting to note that both
performance and energy efficiency increase together with body biasing, due to the favorable ratio between leakage power and dynamic power (4% at 0.5 V with no body biasing) and that even if the memory arrays are not body biased (i.e., leakage does not increase) the operating frequency increases significantly. This makes the operating points at 1.5 V FBB the most energy efficient ones for all performance targets. The best energy point occurs at 0.5 V VDD and 1.5 V FBB, featuring a throughput of 88 TOPs/s and an energy efficiency of 3.6 TOPS/W running ResNet-34.

Fig. 7.10 shows the Energy Efficiency sweep vs. VDD. As mentioned before, the peak energy efficiency is achieved at 0.5V. Below this operating voltage, the relatively small operating frequency (i.e., 60 MHz) makes the leakage dominate, hence efficiency drops. It is interesting to note that, as opposed to other architectures implemented in scaled technologies, where the IO energy is dominating Tbl. 7.6, in Hyperdrive the system level energy drops by only 25% when introducing the I/O energy into the analysis.
Figure 7.10: Energy Efficiency and Throughput vs. supply voltages

Table 7.4. Overview of Cycles, Throughput for ResNet-34

<table>
<thead>
<tr>
<th>layer type</th>
<th>#cycles</th>
<th>#Op</th>
<th>#Op/cycle</th>
<th>#Op/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv</td>
<td>4.52 M</td>
<td>7.09 G</td>
<td>1568</td>
<td></td>
</tr>
<tr>
<td>bnorm</td>
<td>59.90 k</td>
<td>2.94 M</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>bias</td>
<td>59.90 k</td>
<td>2.94 M</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>bypass</td>
<td>7.68 k</td>
<td>376.32 k</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>4.65 M</td>
<td>7.10 G</td>
<td>1.53 k</td>
<td>431 G</td>
</tr>
</tbody>
</table>
7.5. EXPERIMENTAL RESULTS

7.5.2 Benchmarking

The main evaluation of Hyperdrive has been performed on ResNet-34, whose network structure have been used in plenty of applications. This network features a good trade-off between depth and accuracy, i.e., ResNet-50 outperforms ResNet-34 by just 0.5% (Top-1) in terms of classification accuracy on the ImageNet dataset, but is roughly 50% more compute-intensive and the memory footprint is even $3.3\times$ higher (see Sec. 7.4).

The first and the last layer need to stay in full-precision to keep a satisfactory accuracy and are not implemented on Hyperdrive, but they contribute just 3% of the computation (226 MOp of 7.3 GOp) and can therefore also be evaluated on low-power compute platforms [214].

Tbl. 7.4 provides an overview of the number of operations, number of cycles and throughput while Hyperdrive is evaluating ResNet-34. In case of batch normalization, the throughput is reduced since just 49 multipliers are available and the normalization does take more cycles. In the layers where the bypass has to be added, Hyperdrive can also just calculate one output FM at a time, because the memory bandwidth is limited to 49 half-precision words. Fortunately, the non-convolution operations are comparably rare and a real throughput
Table 7.5. Overview of HYPERDRIVE (measured numbers)

<table>
<thead>
<tr>
<th>Operating Point [V]</th>
<th>0.5</th>
<th>0.65</th>
<th>0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power [mW]</td>
<td>22</td>
<td>72</td>
<td>134</td>
</tr>
<tr>
<td>Throughput [Op/cycle]</td>
<td>1568</td>
<td>1568</td>
<td>1568</td>
</tr>
<tr>
<td>Throughput [GOp/s]</td>
<td>88</td>
<td>212</td>
<td>248</td>
</tr>
<tr>
<td>Core Energy Eff. [TOp/s/W]</td>
<td>4.9</td>
<td>3.0</td>
<td>1.9</td>
</tr>
<tr>
<td>Core Area [mm²]</td>
<td>1.92</td>
<td>1.92</td>
<td>1.92</td>
</tr>
<tr>
<td>Memory [Mbit]</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
</tr>
</tbody>
</table>

of 1.53 kOp/cycle or 221.9 GOp/s @ 0.65 V is achieved leading to a very high utilization ratio of 97.5% of the peak throughput. Tbl. 7.7 provides an overview of the utilization (i.e., actual throughput normalized to theoretical peak throughput) for several networks. It can be seen that both ResNet-34 and ShuffleNet have very high utilization since the feature maps tile equally onto the Tile-PUs. In the other case, where the intermediate feature maps are not sized by a multiple of $M \times N$ (i.e., $7 \times 7$), the feature maps are padded with zeros and the last row and column of Tile-PUs is idle during calculation of these zero pixels. Nevertheless, also in these cases, utilization is well above 80% (e.g., YOLOv3 [261] on a 320×320 with 82.8%), which confirms the high flexibility of the proposed architecture with respect to different flavors of network topologies.

7.5.3 I/O in Multi-Chip Setup

Having multiple-chips introduces implicitly more I/O as the border pixels have to be sent to the neighboring chips. To illustrate the relation between the feature map size to the amount of I/O, Fig. 7.12 compares the common weight stationary approach (green) to the feature map stationary approach of Hyperdrive (red). The evaluation is done with ResNet-34 with the taped-out accelerator dimensioned to fit the Worst-Case Layer for $3 \times 224 \times 224$ sized images. By scaling the spatial dimensions evenly, the amount of I/O stays constant for
## EXPERIMENTAL RESULTS

### Table 7.6. Comparison with State-of-the-Art BWN Accelerators (Top: Image Recognition, Bottom: Object Detection)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Image Classification</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>YodaNN (layout) [81]</td>
<td>umc65</td>
<td>ResNet-34</td>
<td>224²</td>
<td>Bin./Q12</td>
<td>1.20</td>
<td>490</td>
<td>0.9</td>
<td>3.6</td>
<td>4.5</td>
<td>1.6</td>
<td>1.3</td>
</tr>
<tr>
<td>YodaNN (layout) [81]</td>
<td>umc65</td>
<td>ResNet-34</td>
<td>224²</td>
<td>Bin./Q12</td>
<td>0.60</td>
<td>18</td>
<td>0.1</td>
<td>3.6</td>
<td>3.7</td>
<td>2.0</td>
<td>1.3</td>
</tr>
<tr>
<td>Wang w/ 25 Mbit SRAM SMIC130</td>
<td>65 nm</td>
<td>ResNet-34</td>
<td>224²</td>
<td>Bin./ENQ6</td>
<td>1.08</td>
<td>876</td>
<td>5.4</td>
<td>1.7</td>
<td>7.2</td>
<td>1.0</td>
<td>11.1</td>
</tr>
<tr>
<td>UNPU (chip)</td>
<td>GF22</td>
<td>ResNet-34</td>
<td>224²</td>
<td>Bin./FP16</td>
<td>0.50</td>
<td>88</td>
<td>1.4</td>
<td>0.5</td>
<td>1.9</td>
<td>3.6</td>
<td>9.0</td>
</tr>
<tr>
<td>Hyperdrive (chip)</td>
<td>GF22</td>
<td>ResNet-34</td>
<td>224²</td>
<td>Bin./FP16</td>
<td>1.00</td>
<td>263</td>
<td>6.5</td>
<td>0.5</td>
<td>7.0</td>
<td>1.0</td>
<td>9.0</td>
</tr>
<tr>
<td>Wang w/ 25 Mbit SRAM SMIC130</td>
<td>GF22</td>
<td>ShuffleNet</td>
<td>224²</td>
<td>Bin./ENQ6</td>
<td>1.08</td>
<td>876</td>
<td>0.3</td>
<td>0.4</td>
<td>0.7</td>
<td>0.5</td>
<td>9.9</td>
</tr>
<tr>
<td>UNPU (chip)</td>
<td>65 nm</td>
<td>ShuffleNet</td>
<td>224²</td>
<td>Bin./Q16</td>
<td>0.77</td>
<td>346</td>
<td>0.1</td>
<td>1.0</td>
<td>1.1</td>
<td>0.3</td>
<td>11.1</td>
</tr>
<tr>
<td>Hyperdrive (chip)</td>
<td>GF22</td>
<td>ShuffleNet</td>
<td>224²</td>
<td>Bin./FP16</td>
<td>0.50</td>
<td>91</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
<td>2.1</td>
<td>9.0</td>
</tr>
</tbody>
</table>

| **Object Detection**          |        |         |            |           |           |                  |                |               |                |                  |            |
| Wang w/ 25 Mbit SRAM SMIC130  | GF22   | YOLOv3(COCO) | 320²     | Bin./ENQ6 | 1.08      | 876              | 40.9           | 4.2           | 45.1           | 1.2              | 9.9        |
| UNPU (chip)                   | 65 nm  | YOLOv3   | 320²      | Bin./Q16  | 0.77      | 346              | 17.2           | 9.1           | 26.4           | 2.0              | 11.1       |
| Hyperdrive (chip)             | GF22   | YOLOv3   | 320²      | Bin./FP16 | 0.50      | 75               | 13.1           | 1.4           | 14.5           | 3.7              | 9.0        |
| Wang w/ 25 Mbit SRAM SMIC130  | GF22   | ResNet-34 | 2k×1k     | Bin./ENQ6 | 243.4     | 40.5             | 283.9          | 1.0           |                |                  |            |
| UNPU (chip) [72]              | 65 nm  | ResNet-34 | 2k×1k     | Bin./Q16  | 0.77      | 346              | 97.7           | 105.6         | 203.3          | 1.4              | 11.1       |
| Hyperdrive (10×5)             | GF22   | ResNet-34 | 2k×1k     | Bin./FP16 | 0.50      | 4547             | 61.9           | 7.6           | 69.5           | 4.3              | 50×9.0     |
| Hyperdrive (20×10)            | GF22   | ResNet-152| 2k×1k     | Bin./FP16 | 0.50      | 18189            | 185.2          | 21.6          | 206.8          | 4.4              | 200×9.0    |

Improvement over state-of-the-art for image classification (ResNet-34): 3.5× 1.8× 1.8×

Improvement over state-of-the-art for object detection: (ResNet-34): 5.3× 3.1× 3.1×
Table 7.7. Utilization of Hyperdrive

<table>
<thead>
<tr>
<th>Network (Resolution)</th>
<th>#Op</th>
<th>#cycles</th>
<th>#Op/cycle</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (Peak Perf.)</td>
<td>1.57 k</td>
<td>100.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResNet-34 (224²)</td>
<td>7.10 G</td>
<td>4.65 M</td>
<td>1.53 k</td>
<td>97.5%</td>
</tr>
<tr>
<td>ShuffleNet (224²)</td>
<td>140 M</td>
<td>90.3 k</td>
<td>1.55 k</td>
<td>98.8%</td>
</tr>
<tr>
<td>YOLOv3 (320²)</td>
<td>53.1 G</td>
<td>33.9 M</td>
<td>1.30 k</td>
<td>82.8%</td>
</tr>
</tbody>
</table>

Figure 7.12: Number of bits to be transmitted with the weight stationary approach compared to the output stationary approach adopted in the Hyperdrive architecture (including border exchange).
the weights of 21.6 Mbit until the maximum dimension of 224×224 is reached. After that the FM is tiled onto several chips, starting with 2×2. This introduces the need exchange two entire rows and columns per output channel and layer to transmit and increases linearly with the FM size until the FM is not fitting anymore onto the 2×2 chips, and tiling is done on 3×3, etc. In case of a systolic array of 2×2 chips, the I/O can be reduced by up to 2.7× and 2.5× for a 3×3 array while accounting for the border exchanges.

7.5.4 Comparison with State-of-the-Art

Tbl. 7.6 compares Hyperdrive with state-of-the-art binary weight CNN accelerators. The upper part of the table compares the SoA accelerators running image recognition applications (i.e., ResNet-34, VGG-16 and ShuffleNet on 224×224 sized images), while the lower part compares key metrics coming from object detection applications with images available in autonomous driving data sets [9,262] (i.e., ResNet-34 on 2048×1024, and YOLOv3 on 320×320 images). At 0.65 V, Hyperdrive achieves a frame rate of 46.7 for ResNet-34, and, most important, the performance is independent of the image resolution thanks to the systolically scalable capabilities of the architecture.

While the totality of previous works is dominated by I/O energy, especially for spatially large feature maps, in Hyperdrive the I/O energy is only a small factor of the total energy (7% to 30%, depending on the application). Thanks to this feature, Hyperdrive outperforms other architectures by up to 1.8× on image classification applications and up to 3.1× in object detection applications, in terms of energy efficiency. More precisely, if we compare with the architecture presented in [73], Hyperdrive is 3.1× more energy efficient, despite the number of bits used for the FMs in ENQ6 is only 6 [73], hence higher energy efficiency is achieved with much less aggressive reduction of HW precision. It should also be mentioned here, that previous work has estimated that for equi-precision results, highly discretized networks need to be just slightly larger (e.g., a ternary-weight (INT2 or $Q_{1,0}$) ResNet-18 is about 12% larger than a full-precision GoogLeNet while both achieve the same accuracy when trained with precision-aware algorithms [231]), whereas the core energy efficiency would improve significantly from stronger
quantization and therefore Hyperdrive is expected to outperform the state-of-the-art even more than the $3.1 \times$ factor reported here when using fixed-point representation and stronger quantization.

Furthermore, we compare our work with UNPU [72], which is the only silicon implementation adopting fixed-point arithmetic with adaptable precision (16, 8, 4, 2, 1) for the feature maps. We compare with the 16-bit mode, as this is the most similar with respect to accuracy. Our approach uses up to $5.3 \times$ less energy for I/O and increases overall energy efficiency by up to $3 \times$ since just the first input FM and the weights need to be streamed to the chip, but not the intermediate FMs. ShuffleNet is a challenging task for all the three accelerators analyzed, as the feature maps are very deep, but spatially small. This implies a low compute intensity relative to the number of weights, which is an adverse pattern for Hyperdrive, and for most accelerators. On the other hand, grouping implies that for every group of output channels, just the subset of assigned input channels is filtered, which reduces the compute complexity while keeping the same feature map volume and is therefore an aspect in Hyperdrive’s favor. Thus Hyperdrive still outperforms the state-of-the-art by $4.2 \times$.

The previous state-of-the-art accelerators are designed in less advanced technologies than Hyperdrive (GF 22nm compared to 65nm and 130nm), thus their core energy efficiency would be improved by using an advanced technology. Nevertheless, Hyperdrive’s core energy efficiency is $12.2 \times$ worse than YodaNN’s and just 1.6 or 3.7× better than UNPU and Wang et al. One of the reasons is that we use FP16 operators which are more robust than Q12 or ENQ6 in [73,81] and were shown to work with the most challenging deep networks. Using floating-point feature maps directly impacts the energy for the accumulation operations as well as memory and register read/write operations. ENQ on the other side has been shown to introduce an accuracy drop of 1.6% already on CIFAR-100 [73], which is more than the difference between running ResNet-34 instead of ResNet-110 on CIFAR-10. It thus implies that a deeper network has to be computed to achieve a comparable accuracy. Furthermore, optimizations such as approximate adders and strong quantization have not been implemented, but can be combined with Hyperdrive’s concepts, coupling core efficiency gains with the removal of the non-scalable I/O bottleneck. For instance,
moving from FP16 to INT12 would lead to an energy efficiency boost that can be estimated to be around $3\times$ for the core, which would translate to a system efficiency boost of $6.8\times$ for high accuracy object detection with ResNet-34 features.

7.6 Conclusion

We have presented Hyperdrive: a systolically scalable hardware architecture for binary-weight neural networks, which dramatically minimizes the I/O energy consumption to achieve outstanding system-level energy efficiency. Hyperdrive achieves an energy efficiency of 4.3 TOp/s/W on object detection task which is more than $3.1\times$ better than prior state-of-the-art architectures, by exploiting a binary-weight streaming mechanism while keeping the entire FMs on-chip. Furthermore, while previous architectures were limited to some specific network sizes, Hyperdrive allows running networks not fitting on a single die, by arranging multiple chips in an on-board 2D systolic array, scaling-up the resolution of neural networks, hence enabling a new class of applications such as object detection on the edge of the IoT.
Chapter 8

Summary and Conclusion

At the begin of this Ph.D. studies in late 2015, the machine learning era had just started, but was dominated by “not so deep”, but memory-intensive neural networks consisting of large fully-connected layers (e.g., VGG-16 needs 40 million multiply-accumulate and has a 40 MByte weight footprint for just the fully-connected layers [248]). They seemed not to be feasible for embedded devices, as already, the weights could not fit in available microcontrollers. Furthermore, running these networks imposed extensive computational requirements, as billions of complex floating-point operations have to be executed. To make matter worse, there were no frameworks or algorithms available to train quantized neural networks. Therefore, just tiny neural networks and for very simple ML tasks (e.g., MNIST) were presented in the embedded community. On the hardware side, very few accelerators were presented, and their energy efficiency was not optimized for energy-constrained devices like it is needed for embedded and IoT end nodes. But it was the time when the first binary-weight and binary neural networks were shown to work with decent performance. Impressed by these results and by the potential savings in energy and memory footprint, we have presented the first accelerator optimized for binary weights neural networks YodaNN (Chapter 5). Furthermore, in
Chapter 2, we have presented an efficient embedded systems application for context recognition on our smartwatch with a light-weight classifier, and sound event detection with the help of binarized neural networks. For the RRM domain, we have implemented state-of-the-art neural networks on the RISC-V-based PULP platform and shown how to improve their efficiencies with the existing Xpulp extension and with newly inserted instructions. Then, besides YodaNN, we have presented Hyperdrive in Chapter 7, a novel accelerator which solves the I/O bottleneck identified in YodaNN (and similar accelerators), and among the first fully-binary accelerator XNORBIN (Chapter 6).

8.1 Overview of the Main Results

In the following, the main results and contributions are summarized:

Embedded Design and Context Recognition

We presented a new smartwatch for context recognition, based on low-power sensors, a low-power microcontroller MSP430, and a multi-core ultra-lower power processor PULPv3. We showed that with an energy-wise light-weight decision tree based on the C4.5 algorithm, and including a small neural network for the ultra-low-power camera, a classification accuracy of 84% had been achieved at the cost of 2.2 mJ total cost. Replacing the visual features (from the initial smartwatch design) with a neural network, and running it on the multi-core platform, gave a speedup of 500× for the camera feature extraction. Complete autonomy can be reached if classification every 14 minutes while using the tiny on-board solar cell and the thermal electric generators.

Binary Neural Networks for Sound Event Detection

The state-of-the-art CNN classifier for Sound Event Detection presented by Meyer et al. does not fit on an out-of-the-shelf microcontroller, as 6.3 MByte of memory would be required. Thanks to extreme compression through binary weight and activations, we have shown that the binarized CNN fits into a 230 kB of RAM (28× less compared to
8.1 OVERVIEW OF THE MAIN RESULTS

FP32), which fits on the low-power and DSP-enhanced GAP8 platform. We have trained the network and achieved a classification accuracy of 77.9% with 28 different classes, which is 7.2 percent points below the full-precision baseline. The system has a peak energy efficiency of 134 GOp/s/W or 69 GOp/s/W on average at a frame rate of 2 FPS. A 10× faster and 51× more energy-efficient performance on the BNN inference is achieved compared to the same implementation on an ARM Cortex-M4F platform, which comes from multi-core capabilities (i.e., 7.2/2.6×), the build-in popcount instruction (i.e., 2.5/2.9×), and other low-power capabilities (e.g., latch-based memories).

RNN ISA-Extensions for a general-purpose RISC-V processor optimized for the RRM domain

In modern 5G Radio Resource Management (RRM), RNN and in general Neural Networks are more and more used to model and predict the physical property of wireless transmission, and for smart resource allocation (i.e., frequency bands or bandwidth). A full-custom accelerator is not flexible enough to cope with the fast-changing network developments, and FPGAs are too expensive for large distribution of mobile base stations. We have presented an extended RISC-V processor, which supports single-cycle hyperbolic tangent and sigmoid instructions, and the prefetch-load compute instruction to parallelize compute and load within the same cycle. Hardware loops, post-increment load and store, and SIMD instructions from the RI5CY extensions [214] give a 4.4× improvement in energy and throughput, another 1.13× improvement give the tanh and sig extensions with a little 3% increase in core area, and no deterioration of classification accuracy. With efficient tiling and date reuse with the general-purpose register, another 1.9× is achieved. Finally, the prefetch-load and compute instruction gives a 1.8× better throughput with a 1.2× better energy efficiency.

Custom ASIC Accelerators

A custom accelerator has to be designed to reach the highest performance in energy efficiency. A significant part of an efficient accelerator is careful data management, as memory access costs are higher than the
computational logic costs itself, notably off-chip memory access. We have been focusing on highly-quantized neural networks, and we have presented two binary-weight neural networks accelerators YodaNN and Hyperdrive, and a fully-binary accelerator XNORBIN. The core energy efficiencies have been shown to be up to 61.2 TOp/s/W for BWNs, and 205 TOp/s/W for BNNs, respectively. Efficient data reuse, has been implemented in YodaNN with a sliding window approach. All input feature map pixels are loaded just once (per column) from the feature map memory and reused for all the output feature map pixels and as long as needed through processing the output columns. Furthermore, thanks to the extreme simplification of multiply-accumulate to sign inversion and accumulate (i.e., BWN), or XNOR and accumulate (i.e., BNN) and efficient design of the corresponding adder trees has shown very high compute energy efficiencies.

In YodaNN, we have shown that the energy efficiency of the computational units was reduced by $4.8\times$ and the access cost of the weight memory even $31\times$. By designing a latch-based memory for the feature map memory, we could decrease the access cost to the memory by another $3.5\times$, which led to an overall $4.6\times$ improvement of energy efficiency at the same voltage corner, at the cost of $8.9\times$ larger memory. Furthermore, using latch-based memory showed to be very beneficial, as we could reduce further the supply voltage down to 0.6 V. With a core energy efficiency of 61.2 TOp/s/W, YodaNN outperforms the state-of-the-art neural network accelerators by $32\times$. Nevertheless, it has to be remarked that in YodaNN and XNORBIN we have optimized for $7 \times 7$ kernels, which at that time was the common kernel size. By adding flexibility in the adder tree, the energy efficiency of the $7 \times 7$ convolutions was already reduced by 29% (to 61.2 TOp/s/W). But more significant, the $3 \times 3$ and $1 \times 1$ have a utilization of 36%, or 4%, which leads to a drastic reduction in energy efficiency. Fortunately, the throughput can be further increased by parallelizing in the input channel domain. Nevertheless, YodaNN and XNORBIN have quite limited memory footprint (i.e., even if scaled) and have therefore high I/O bandwidth requirements to load and store feature maps for large-scale image tasks, limiting the system energy efficiency to 2.8 TOp/s/W.
8.1. OVERVIEW OF THE MAIN RESULTS

Multi-Chip Systolic binary CNN Accelerator

With Hyperdrive, we have shown a novel scalable design, which is not just scalable on a chip-level, but also on multiple chips on board-level. By keeping the entire feature maps on the chips, just the weights need to be streamed. Thus, the design is also independent of the depth of the networks, but on the size of two consequent feature map volumes (including a potential bypass volume). With this approach, we showed an overall energy efficiency of 4.3 TOp/s/W, which is more than 3.1× improvement over the state-of-the-art BWN accelerators, due to the high reduction in I/O bandwidth (e.g., up to 58×).

BNN vs. BWN

Neural Networks can be quantized down to 8 bits in activations and weights, and binarizing the weights led to an insignificant decrease in accuracy performance. Recent works have shown that BWNs achieve state-of-the-art performance in small and simple tasks (e.g., MNIST, Cifar-10, ...), and reach the performance of challenging tasks slowly (e.g., 4 points gap in ILSVRC [263]). The most efficient method works by iterative binarization and keeping a few significant weights with higher precision (i.e., 0.5% of the weights). BNN’s on the other hand, still have a large performance gap (≈10 points in accuracy on ILSVRC) in very important challenges. We have presented the first BWN and BNN accelerators. Scaling YodaNN down to 22 nm technology based on Dreslinksi et al. [1], a peak performance of 149 TOP/s/W can be achieved, while 205 TTOP/s/W was achieved with the BNN accelerator XNORBIN, which is 38% higher. Considering the much larger gap in accuracy from BNNs and the comparably small gain in energy efficiency, BWNs seem to be favorable over BNN, while achieving 32× gain compared to fixed-point alternatives.
8.2 Outlook

Ternary Weights

We have been focusing on binary weights, but recent research [231, 244, 263, 264] are halving the accuracy gap between binary weights and full-precision networks, by adding zero-weights. Using these ternary weights introduces roughly no extra energy costs to binary weights acceleration hardware: First, the compute units stay nearly the same: Instead of having a multiplexer for the positive $x$ and negative input $-x$, another port is added with 0 contribution. On the other hand side, the weight footprint will be increased by $n \geq \log_2(3) \approx 1.58$ (e.g., 5 ternary weights can be packed into 8 bits; 1.625 bits/weight).

Sparsity

Sparsity is common in neural networks, already as the commonly used Rectified Linear Unit (ReLU) activation function introduces zero-values in the output feature maps. Furthermore, in full-precision neural networks, a substantial amount of weights is around zero. Previous work (e.g., the prominent EIE accelerator [68]), has been exploiting this for quantized weights and activations, by first skipping the multiplications with zero and secondly using pruning and retraining to favor even higher sparsity. Ternary weights instead of binary weights allow again to skip zero-weights. Using zero-weight-skipping also comes with some disadvantages, as the control and data flow of parallel units start to differ, which has to be evaluated and handled carefully.

Heterogeneous Accelerators

Zhou et al. have shown that quantized or even highly quantized neural networks can be trained iteratively. A very small decrease in accuracy is achieved, while always some of the weights are fixed to a binary value and keeping a tiny amount of weights in higher precision [231]. Furthermore, in highly quantized neural networks like BNNs, typically, the first and the last layer have to be kept in higher precision to avoid a substantial drop in accuracy. On the one hand side, it needs to be better understood how to quantize these networks, whereas
some weights are binarized or quantized or whether to use different quantization schemes among different network layers; on the other hand, the hardware needs to be flexible to cope with these networks. E.g., a system could be optimized for ternary weights, but support for a few weights with higher precision.

**New Advances in the ML Field**

The ML field is changing rapidly; adapting to these changes is important. E.g., For machine learning tasks based on sequential data (e.g., audio or video), LSTMs have been out-performed by Gated Recurrent Networks. Gated Recurrent Networks have a very similar structure and could be calculated by flexible accelerators able to run matrix-vector multiplication and adaptable data graphs. Furthermore, Temporal Convolutional Networks have shown to be trained much faster due to the lack of recurrent paths and seem, therefore, to replace LSTMs in the near future. In the last years, also optimized network topologies have been suggested, which have to be supported in future hardware accelerators:

1. Reducing the memory footprint by using CNNs instead of fully-connected layers [6, 36]

2. By replacing large convolution kernel like $11 \times 11$ or $7 \times 7$ with $3 \times 3$ with $1 \times 1$ kernels and

3. Reducing input channels for $3 \times 3$ CNN layers in SqueezeNet [37]

4. Using depth-wise convolutions and reduced intermediate feature map volumes in MobileNet [38]

5. Point-wise group convolutions in ShuffleNet [39].
CHAPTER 8. SUMMARY AND CONCLUSION
Appendix A

Notations and Acronyms

Operators

| \cdot | absolute value
| \ceil{\cdot} | ceil: smallest integer value equal to or larger as argument
| \floor{\cdot} | floor: largest integer value equal to or smaller as argument
| \hat{a} | normalized value, i.e., \( \hat{a}_i = \frac{a_i}{\max_i a_i} \)
| s_i | i-th entry of vector \( s \)
| h_i | i-th column of matrix \( H \)
| H_i | i-th row of matrix \( H \)
| H_{i,j} | entry of the i-th row and j-th column of matrix \( H \)
| \| \cdot \|_1 | \ell_1\text{-norm}, i.e., \( \sum_i^n |x_i| \) for \( x \in \mathbb{R}^n \)
| \| \cdot \| | \ell_2\text{-norm or Euclidean norm}, i.e., \( \sqrt{\sum_i^n |x_i|^2} \) for \( x \in \mathbb{C}^n \)
| \| \cdot \|_\infty | \ell_\infty\text{-norm}, i.e., \( \sum_i^n \|x_i\| \) for \( x \in \mathbb{R}^n \)
| \mathcal{E}\{ \cdot \} | expectation operator
| \log_2 | base-2 logarithm
Operators

\[ \log_{10} \] base-10 logarithm
\[ \text{ReLU}(\cdot) \] Rectified Linear Unit, i.e., \(1_{x>0} \cdot x\)
\[ \text{sgn}(\cdot) \] signum operator
\[ \text{sig}(\cdot) \] sigmoid operator, i.e., \(1/(1 + e^{-x})\)
\[ \text{tanh}(\cdot) \] Hyperbolic tangent operator
\[ 1_{\text{cond}} \] Conditional Function, i.e., returns 1 in case of \(\text{cond}\) is true, else 0

4G  4th mobile communication generation
5G  5th mobile communication generation (advanced wireless technology)

ADC  analog-to-digital converter
AI   Artificial Intelligence
ASIC application-specific integrated circuit
AWGN Additive White Gaussian Noise

billion \(= 10^9\) (short scale)
BNN Binary Neural Network
BPU Binary Processing Unit
BWN Binary-Weight Neural Network

CCI Co-channel interference
CDMA Code-division multiple access
CMOS complementary metal-oxide semiconductor
CNN Convolutional Neural Network
CSI Channel State Information
CT  Computed Tomography

DAC digital-to-analog converter
DCT Discrete Cosine Transform
DFT discrete Fourier transform
DL Deep Learning
DMA Direct Memory Access
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNN</td>
<td>Deep Neural Network</td>
</tr>
<tr>
<td>DQN</td>
<td>Deep Q-Network</td>
</tr>
<tr>
<td>DRL</td>
<td>Deep Reinforcement Learning</td>
</tr>
<tr>
<td>DSA</td>
<td>Dynamic Spectrum access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>FC</td>
<td>Fabric Controller</td>
</tr>
<tr>
<td>FC</td>
<td>Fully-Connected (Neural Network Layer)</td>
</tr>
<tr>
<td>FFT</td>
<td>fast Fourier transform</td>
</tr>
<tr>
<td>FM</td>
<td>Feature Map</td>
</tr>
<tr>
<td>FP16</td>
<td>Half-Precision Floating-Point</td>
</tr>
<tr>
<td>FP32</td>
<td>Full-Precision Floating-Point</td>
</tr>
<tr>
<td>FPGA</td>
<td>field-programmable gate array</td>
</tr>
<tr>
<td>GE</td>
<td>Gate Equivalent (Area unit equivalent to a two-input NAND [55])</td>
</tr>
<tr>
<td>GOp</td>
<td>Billion Operation Per Second, 1 MAC is considered to be equal to 2 Op</td>
</tr>
<tr>
<td>GPIO</td>
<td>General Purpose Input/Output</td>
</tr>
<tr>
<td>GPR</td>
<td>General-Purpose Register</td>
</tr>
<tr>
<td>GRU</td>
<td>Gated Recurrent Unit</td>
</tr>
<tr>
<td>HMM</td>
<td>Hidden Markov Model</td>
</tr>
<tr>
<td>HVT</td>
<td>High-Voltage Threshold</td>
</tr>
<tr>
<td>HWCE</td>
<td>Hardware Convolution Engine</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IFM</td>
<td>Input Feature Map</td>
</tr>
<tr>
<td>i.i.d.</td>
<td>independent and identically distributed</td>
</tr>
<tr>
<td>IIS</td>
<td>Integrated Systems Laboratory</td>
</tr>
<tr>
<td>ILSVRC</td>
<td>ImageNet Large-Scale Visual Recognition Challenge</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IPS</td>
<td>Instructions Per Second</td>
</tr>
</tbody>
</table>
ISA Instruction Set Architecture

KD Knowledge Distillation

LSTM Long Short-Term Memory

LSU Load-Store Unit

LTE long term evolution

LTE-U Long Term Evolution in unlicensed spectrum

LUT Look-Up Table

LVT Low-Voltage Threshold

MAC Multiply-ACcumulate

MAC Media Access Control

MAE Maximum Absolute Error, i.e., $\max_i(|x_i|)$

MCU MicroController Unit

MFCC Mel Frequency Cepstral Coefficients

MIMO multiple-input multiple-output

MIPS Million Instructions Per Second

ML Machine Learning

MLP Multi-Layer Perceptron

MMSE minimum mean squared error

MOp Million Operation Per Second, 1 MAC is considered to be equal to 2 Op

MSE Mean Squared Error, i.e., $\frac{1}{n} \sum_{i=1}^{n}(Y_i - \bar{Y})^2$

NN Neural Network

OFM Output Feature Map

Op Operations Per Second (Unit for Computational Throughput)

OPS Operations Per Second (Unit for Computational Throughput)

Op/s/W Operations Per Energy (Unit for Energy Efficiency)
PDF Probability Density Function
phoneme Smallest sound unit in a language.
PHY physical layer
PLA Piecewise Linear Approximation
PULP Parallel Ultra Low Power

QoS Quality of Service

ReLU Rectified Linear Unit
RISC Reduced Instruction Set Computer
RNN Recurrent Neural Network
RRM Radio Resource Management
RSRP Reference Signal Received Power
RTC Real Time Clock
RV32IMFC RISC-V ISA with Integer, Integer Multiplications, single-precision floating-point, and compressed instructions.

SBS Small (cell) Base Stations
SCM Standard Cell Memory
SDK Software Development Kit
SED Sound Event Detection
SGD Stochastic Gradient Descent
SIMD Single Instruction Multiple Data
SNR signal-to-noise ratio
SoC System-on-a-chip
STFT Short Time Fourier Transform
SVD Singular Value Decomposition

TCDM Tightly-Coupled Data Memory
TDMA Time Division Multiple Access
TNN Ternary Neural Network
TOp Trillion Operation Per Second, 1 MAC is considered to be equal to 2 Op

trillion \(= 10^{12}\) (short scale)
<table>
<thead>
<tr>
<th>TWN</th>
<th>Ternary-Weight Neural Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLSI</td>
<td>very large scale integration</td>
</tr>
<tr>
<td>WCS</td>
<td>Wireless Communication Systems</td>
</tr>
<tr>
<td>WMMSE</td>
<td>Weighted Minimum Mean Square Error</td>
</tr>
</tbody>
</table>
Bibliography


[113] “Ambiq Apollo Data Brief .”


[131] “NXP LPC54100 Datasheet.”


[137] A. Teman, D. Rossi, P. Meinerzhagen, L. Benini, and A. Burg, “Controlled placement of standard cell memory arrays for high density and low power
in 28nm FD SOI ,” in Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific, 1 2015, pp. 81–86.


Curriculum Vitae

Renzo Andri was born on 17 July 1990 in Brig, Valais, Switzerland. He received the B.Sc. and M.Sc. degree from ETH Zurich in 2013 and 2015, respectively. He has been pursuing his Ph.D. degree under the supervision of Prof. Dr. Luca Benini since November 2015 and has been working as a teaching and research assistant at the Integrated Systems Laboratory at ETH Zurich. His main research interests involve the design of low-power machine learning hardware accelerators and the design of hardware-software systems for efficient machine learning systems for the embedded and IoT domain. Mr. Andri has won the Donald O. Pederson Award for the paper YodaNN: An Architecture for Ultra-low Power Binary-Weight CNN Acceleration, published in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.