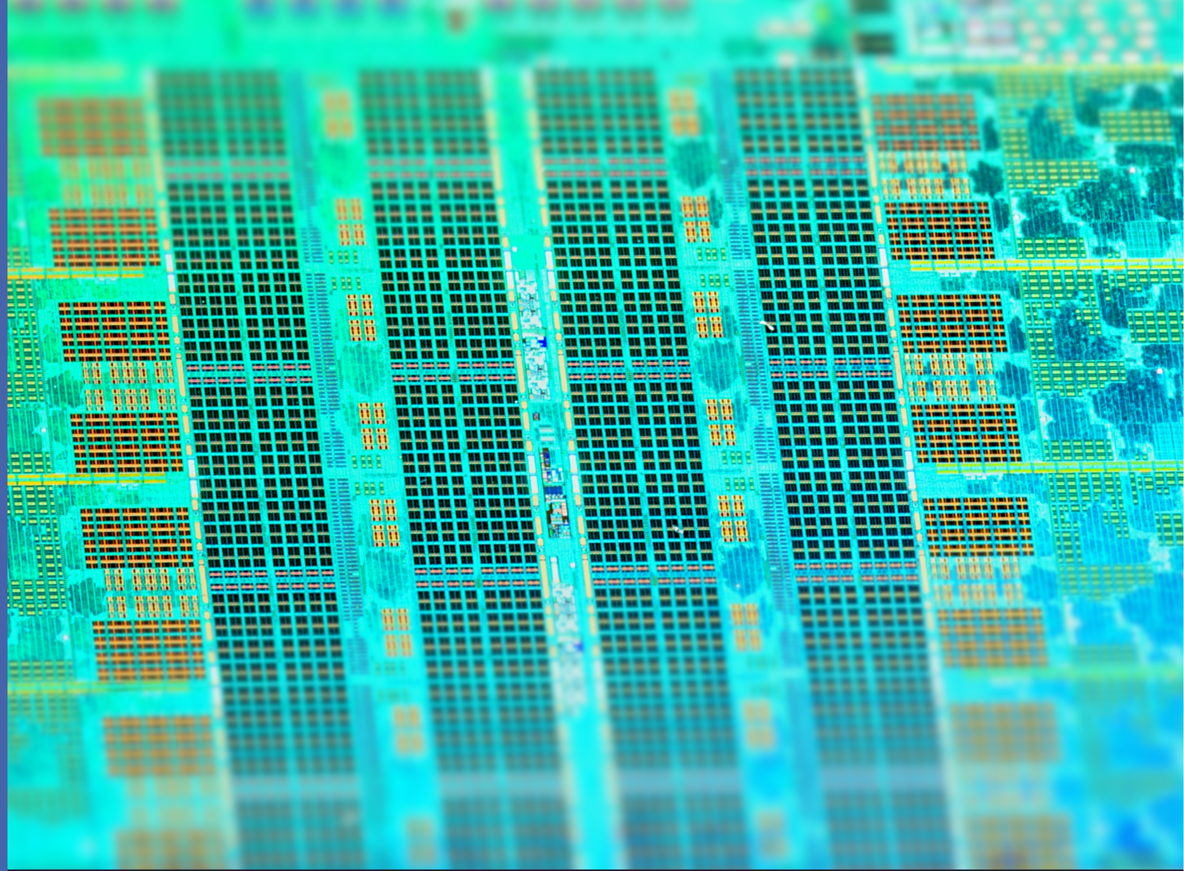


Tim Hotfilter

Efficient Optimization of Convolutional Neural Networks for Modern Embedded High-Performance Applications

Efficient Optimization of Convolutional Neural Networks for Modern Embedded High-Performance Applications
Tim Hotfilter

Efficient Optimization of Convolutional Neural Networks for Modern Embedded High-Performance Applications

Zur Erlangung des akademischen Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN (Dr.-Ing.)

von der KIT-Fakultät für Elektrotechnik und Informationstechnik des
Karlsruher Instituts für Technologie (KIT)
angenommene

DISSERTATION

von

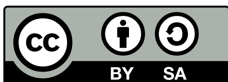
M.Sc. Tim Hotfilter
geb. in Mettingen

Tag der mündlichen Prüfung: 13. Mai 2024

Hauptreferent: Prof. Dr.-Ing. Dr. h.c. Jürgen Becker
Korreferent: Prof. Dr. Holger Fröning

Efficient Optimization of Convolutional Neural Networks
for Modern Embedded High-Performance Applications

First edition: July 26, 2024
Tim Hotfilter



This work is licensed under CC BY-SA 4.0. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/

Abstract

Since the invention of modern computers, people have dreamed of making them able to learn and think autonomously. Modern machine learning (ML) algorithms are bringing this dream within reach. A major milestone in this endeavor was the invention of Deep Neural Networks (DNNs). Numerous applications such as autonomous driving or robotics would be unthinkable today without DNNs. Despite their versatility and very good prediction accuracy for problems that were impossible to describe with traditional algorithms, they are very computationally intensive. Convolutional Neural Networks (CNNs), a subcategory of DNNs that are indispensable for image processing, are particularly challenging. For example, to perform high-resolution image processing, tens of billions of computations must be performed, which could only be realized thanks to the modern and powerful computing hardware of recent years.

Despite the progress made in hardware solutions for CNNs, there are still many unanswered questions, specifically with regard to embedded systems and their requirements. Not only performance, but also energy efficiency, latency, reliability (safety) and algorithmic accuracy play an important role here. This becomes clear by exemplary means of autonomous vehicles: Environment perception must be efficient, accurate, and safe.

Over the last two decades, various hardware accelerators, tools and optimization strategies have been introduced to cope with the increasing complexity of CNNs. The search for optimal CNN topologies and accelerators has been automated by Neural Architecture Search (NAS). In addition, quantization and pruning were investigated in order to achieve higher energy efficiency. The first method involves the reduction of computing precision, while the second involves the omission of computing operations that have little impact on the overall result. Such approaches work particularly well in the context of CNNs, as they can inherently tolerate certain computational inaccuracies. Great progress has also been made in the robustness of CNNs. Despite the large amount of previous work, there are still many open questions and challenges to deploy CNNs in an energy-efficient and safe way, especially with regard to demanding applications such as autonomous driving.

In this thesis, we therefore introduce three new approaches that simplify the design of a CNN accelerator and algorithm, as well as increase their energy efficiency and performance. First, we introduce the cycle-accurate simulation tool FLECSim-SoC and

an analytical counterpart. These tools make it possible to systematically investigate the influence of various architecture parameters of the CNN accelerator on its energy efficiency, chip area, and performance. The two tools are not opposing but complement each other. A cycle-accurate simulation is highly accurate, but time-consuming, and the analytical modeling is slightly less accurate but significantly faster. In two case studies we showed that the two simulation tools provide valuable insights into the impact of different accelerator configurations and thus help to guide the design process.

To further optimize the energy efficiency of the designed accelerators, we conceived, developed, and evaluated a novel pruning method subsequently. It analyzes regular regions in intermediate results of CNNs. Such regions can be pruned efficiently by the CNN accelerator if they match the dimensions of the accelerator. Since these regions rarely occur in CNNs, we designed, in addition to the hardware architecture, a tool, which determines thresholds that decide whether a region is pruned. Our pruning method is able to reduce the operations by 20 % and saves up to 19 % of energy. Thereby, the corresponding hardware extension requires significantly less chip area than comparable work.

In the context of robust and efficient execution of CNNs, we finally investigated mixed-precision quantization, which allows to set the precision of operands in a fine-granular manner. This can optimize the compromise between model accuracy and energy savings. Since CNN accelerators, which support mixed-precision, often have high latency, we first introduce a novel and area-efficient hardware architecture. With the help of mixed-precision inference, we cannot only reduce energy consumption, but also increase the robustness of the CNN, for example when an environment perception has to cope with perturbed input images. By dynamically increasing the operand's precision in such a case, we could show that the original model accuracy of modern workloads could be restored. At the same time, about 45 % of the energy-intensive memory accesses were saved.

Zusammenfassung

Seit der Erfindung von Computern träumt der Mensch davon, ihnen selbstständiges Lernen und Denken beizubringen. Moderne Algorithmen des maschinellen Lernens (ML) rücken diesen Wunsch in greifbare Nähe. Die Erfindung der tiefen neuronalen Netze (Deep Neural Networks, DNNs) ist dabei ein wichtiger Meilenstein. Zahlreiche Anwendungen wie autonomes Fahren oder Robotik sind heute ohne DNNs undenkbar. Trotz ihrer Vielseitigkeit und ihrer sehr guten Vorhersagegenauigkeit für bisher algorithmisch schwer beschreibbare Probleme sind sie sehr rechenintensiv. Besonders rechenintensiv sind die Convolutional Neural Networks (CNNs), eine Unterkategorie der DNNs, die für die Bildverarbeitung unentbehrlich sind. Beispielsweise erfordert die Verarbeitung von hochauflösenden Bildern Milliarden von Berechnungen, die erst durch die modernere und leistungsfähigere Hardware der letzten Jahre in die Praxis umgesetzt werden konnten.

Trotz der Fortschritte bei den Hardwarelösungen für CNNs gibt es noch viele offene Fragen, insbesondere in Bezug auf eingebettete Systeme und deren Anforderungen. Hierbei spielen nicht nur die Performance, sondern auch die Energieeffizienz, die Latenz, die Ausfallsicherheit (Safety) sowie die algorithmische Genauigkeit (Accuracy) eine Rolle. Dies wird am Beispiel autonomer Fahrzeuge deutlich: Eine Umgebungserkennung muss effizient, genau und zuverlässig durchgeführt werden.

In den letzten zwei Jahrzehnten wurden verschiedene Hardwarebeschleuniger, Tools und Optimierungsstrategien vorgestellt, um der zunehmenden Komplexität von CNNs gerecht zu werden. Die Suche nach optimalen CNN-Strukturen und Beschleunigern wurde durch Neural Architecture Search (NAS) automatisiert. Zudem wurden Quantisierung und Pruning untersucht, um eine höhere Energieeffizienz zu erzielen. Beim ersten Verfahren handelt es sich um die Reduzierung der Rechengenauigkeit, beim zweiten um das Auslassen von Rechenoperationen, die das Gesamtergebnis nur wenig beeinflussen. Solche Ansätze funktionieren insbesondere im Umfeld von CNNs sehr gut, da sie gewisse Ungenauigkeiten inhärent tolerieren können. Auch im Rahmen der robusten Ausführung von CNNs wurden in den letzten Jahren große Fortschritte erzielt. Trotz der Vielzahl an vorausgegangenen Arbeiten, gibt es noch viele offene Fragen und Herausforderungen, um CNNs energieeffizient und sicher einsetzen zu können, speziell im Hinblick auf anspruchsvolle Anwendungen wie dem autonomen Fahren.

Die vorliegende Arbeit präsentiert drei neue Ansätze, die den Entwurf von CNN-Beschleunigern und Modellen vereinfachen und gleichzeitig deren Energieeffizienz und Leistung während der Ausführung steigern. Zunächst wird das zyklenakkurate Simulationswerkzeug FLECSim-SoC vorgestellt, sowie eine analytische Alternative. Diese Werkzeuge ermöglichen es, den Einfluss verschiedener Architekturparameter des CNN-Beschleunigers auf dessen Energiebedarf, Chip-Fläche und Performanz systematisch zu untersuchen. Die beiden Werkzeuge ergänzen sich gegenseitig und sollten nicht als gegensätzlich betrachtet werden. Eine zyklenakkurate Simulation ist hochgenau, aber langwierig, während die analytische Modellierung leicht ungenauer, dafür aber deutlich schneller ist. In zwei Fallstudien wurde gezeigt, dass Simulationstools wertvolle Einblicke in die Auswirkungen verschiedener Beschleunigerkonfigurationen geben und somit den Entwurfsprozess vereinfachen können.

Um die Energieeffizienz der Beschleuniger weiter zu optimieren, wurde darauffolgend eine neuartige Pruning-Methode entworfen und untersucht. Dabei werden die Zwischenergebnisse von CNNs analysiert und Bereiche identifiziert, die mit den Dimensionen des Beschleunigers übereinstimmen und so besonders effizient durch diesen geprunt werden können. Da diese Regionen selten in CNNs vorkommen, wurde zusätzlich zu der Hardwarearchitektur noch ein Werkzeug entwickelt, welches Schwellenwerte bestimmt ab denen eine Region geprunt wird. Insgesamt konnten so mit dieser Pruning-Methode über 20 % aller Berechnungen und 19 % der Energie eingespart werden. Die entsprechende Hardware-Erweiterung benötigt dabei deutlich weniger Chip-Fläche als vergleichbare Arbeiten.

Im Rahmen der sicheren und effizienten Ausführung von CNNs wurde abschließend Mixed-Precision-Quantisierung untersucht. Diese ermöglicht es, die Präzision von Operanden feingranular einzustellen, um den Kompromiss zwischen Modellgenauigkeit und Energieeinsparung zu optimieren. Da CNN-Beschleuniger, welche Mixed-Precision unterstützen, oft eine hohe Latenz aufweisen, wird zunächst eine neue und platzsparende Hardwarearchitektur entwickelt. Durch die Verwendung von Mixed-Precision kann nicht nur die Energie reduziert, sondern auch die Robustheit von CNNs erhöht werden. Im Rahmen dieser Arbeit wurde gezeigt, dass durch eine dynamische Erhöhung der Berechnungspräzision bei Eingangsbildern gestört durch Wettereinflüsse, die ursprüngliche Modellgenauigkeit moderner CNNs wiederhergestellt werden konnte. Gleichzeitig konnten etwa 45 % der energieaufwendigen Speicherzugriffe eingespart werden.

Danksagung

Der Weg einer Promotion ist mit zahlreichen Höhen und Tiefen verbunden. Ohne die immense Unterstützung vieler Personen hätte ich dieses Projekt nicht erfolgreich abschließen können. Gerade in herausfordernden Zeiten hat sich gezeigt, dass individuelle Leistungen nicht ohne Teamwork möglich sind.

An erster Stelle möchte ich meinem Doktorvater, Prof. Dr.-Ing. Dr. h.c. Jürgen Becker, für seine Anleitung, Ermutigung und die anregenden Diskussionen während meiner Forschung danken. Die Arbeit am Institut für Technik der Informationsverarbeitung (ITIV) war eine bereichernde Erfahrung. Ich habe seine fordernde, aber unterstützende Art immer sehr geschätzt. Das Vertrauen, das er mir entgegenbrachte, ermöglichte es mir, während der gemeinsamen Zeit verschiedene Aufgaben zu übernehmen. Zudem danke ich meinem Co-Referenten, Prof. Dr. Holger Fröning, für seine Unterstützung und sein wertvolles Feedback. Des Weiteren danke ich Prof. Dr. rer. nat. Ulrich Lemmer für die Übernahme des Prüfungsvorsitzes, sowie meinen Beisitzern, Prof. Dr.-Ing. Mike Barth und Prof. Dr.-Ing. Peter Rost.

Ich möchte auch meiner Familie, insbesondere meinen Eltern Dirk und Regina, meinen herzlichen Dank aussprechen. Wann immer es schwierig wurde, hatten sie einen guten Rat und ein offenes Ohr. Obwohl unsere Familie klein ist, ist unser Zusammenhalt umso stärker. Und ein großes Lob an alle meine Freunde, die mir nicht nur wertvolle Tipps gaben, sondern mir auch halfen, durchzuhalten.

Ein Großteil dieser Arbeit wäre ohne die Zusammenarbeit mit meinen Kollegen und den Studierenden, die ich betreuen durfte, nicht möglich gewesen. Die gemeinsamen ausführlichen Diskussionen und die Kooperation in zahlreichen Projekten ermöglichten es mir, verschiedene Aspekte meiner Forschung zu erkunden.

Meine Zeit am ITIV wurde von einigen der besten Kollegen begleitet, die ich mir vorstellen kann. Projekte, die über die Promotion hinausgingen, wie z.B. die Betreuung des LAMA-Labors waren sehr spannend und haben viel Spaß gemacht. Über die täglichen Aufgaben hinaus blieb die Motivation durch verschiedene gemeinsame Aktivitäten, von Grillabenden und DIY-Elektronikprojekten bis hin zu Kinobesuchen und Skiausflügen, erhalten. Nach fast sechs Jahren in diesem Team kann ich mit gutem Gewissen sagen, dass ich bleibende Freundschaften geschlossen habe, die weit über meine Zeit am ITIV hinausreichen werden. Vielen Dank!

Karlsruhe im Juli 2024 *Tim Hotfilter*

Contents

Acronyms xi

1 Introduction 1

1.1 Motivation and Context . . . 3
1.2 Challenges, Proposed Solutions and Contributions . . . 4
1.3 Outline . . . 7

2 Basic Principles and Fundamentals 9

2.1 Introduction and History . . . 9
2.1.1 Artificial Intelligence and Machine Learning . . . 10
2.1.2 Artificial Neural Networks and its Building Blocks . . . 12
2.1.3 Training of Artificial Neural Networks . . . 16
2.2 Convolutional Neural Networks . . . 19
2.2.1 Convolution Layer . . . 20
2.2.2 Pooling Layer . . . 24
2.2.3 Evaluation Metrics . . . 25
2.3 Benchmark Datasets and Models . . . 27
2.3.1 PyTorch . . . 27
2.3.2 Popular Benchmark Datasets . . . 27
2.3.3 Benchmark Convolutional Neural Networks . . . 30
2.4 Hardware Architecture Development . . . 38
2.4.1 Design Languages, Automation, and Tools . . . 40
2.4.2 CPUs, GPUs and FPGAs . . . 41
2.4.3 Hardware Acceleration and Co-Design . . . 44
2.5 Hardware Accelerators for DNNs . . . 46
2.5.1 Evaluation Metrics and Theoretical Background . . . 46
2.5.2 DNN Computation, Memory Hierarchy and Data Orchestration 48
2.5.3 General Purpose Hardware Platforms . . . 54
2.5.4 Systolic Arrays and Tensor Processing Units . . . 55

3 State of the Art in Deep Neural Network Acceleration and Optimization 59
3.1 Popular DNN Hardware Accelerators . . . 59
3.1.1 Commercial DNN Accelerators . . . 61
3.1.2 Academic and Research DNN Accelerators . . . 62
3.2 Design Methodologies for DNN Accelerators . . . 66
3.2.1 Mapping . . . 66
3.2.2 Data Orchestration . . . 68
3.2.3 Neural Architecture Search and Design Space Exploration . . 69
3.3 Methods to Optimize Neural Network Inference and Training 72
3.3.1 Number Formats . . . 73
3.3.2 Quantization . . . 76
3.3.3 Pruning . . . 81
3.3.4 Combined works . . . 86
3.4 Summary, Challenges and Contributions to the State of the Art . . . 88

4 Algorithm and Hardware Accelerator Co-Design for Efficient Hardware Accelerator Design 93
4.1 Overview, Introduction and Motivation . . . 94
4.2 Related Work . . . 96
4.3 An Accurate and Flexible End-to-End Co-Design Simulation Framework for System on Chips: FLECSim-SoC . . . 104
4.3.1 SystemC Simulation Environment . . . 105
4.3.2 Python Simulation Framework . . . 112
4.3.3 Design Space Exploration . . . 114
4.4 Analytical Modelling of Systolic Arrays for Rapid DNN Accelerator Assessment . . . 115
4.4.1 Analytical Accelerator Model . . . 117
4.4.2 Design Space Exploration . . . 122
4.4.3 Implementation using the Systolic Array Generator Gemmini 123
4.5 Evaluation and Discussion . . . 127
4.5.1 Design Space and Setup . . . 127
4.5.2 Cycle-accurate Simulation using FLECSim . . . 128
4.5.3 Analytical Evaluation . . . 132
4.5.4 Case Study: Design of an eFPGA tile . . . 138
4.5.5 Discussion . . . 143
4.6 Conclusion and Outlook . . . 145

5 Leveraging and Increasing Regular Sparse Activations in CNNs for Energy Efficiency **147**
5.1 Overview, Introduction, and Motivation 148
5.2 Principle of Sparse-Blox and Spex 151
5.3 Related Work 154
5.4 Spex Exploration Framework: A Tool to Systematically Increase Regular Sparsity in CNNs 159
5.4.1 Design Space Exploration 160
5.4.2 Implementation details of Spex 163
5.4.3 Energy estimation using Timeloop and Accelergy 167
5.5 Sparse-Blox: A Hardware Extension to Prune Regular Activation Sparsity 169
5.6 Evaluation and Discussion 172
5.6.1 Selected Workloads 172
5.6.2 Evaluation Setup 173
5.6.3 Number of Found Sparse Blocks and Savings in during Inference 177
5.6.4 Comparison to state-of-the-art Sparse DNN Accelerators . . . 181
5.6.5 Discussion 183
5.7 Conclusion and Outlook 185

6 Exploiting Mixed-Precision CNN Inference to Increase Robustness and Energy Efficiency **187**
6.1 Overview, Introduction and Motivation 188
6.2 Our Mixed-Precision Concept for Increased Robustness and Energy Efficiency 191
6.3 Related Work 193
6.3.1 Hardware Architectures for Mixed-Precision Inference 193
6.3.2 Robustness-aware Quantization of CNNs 197
6.4 Toolchain to find the Best Precision for Robust Mixed-Precision CNN Inference 198
6.4.1 Design Space Exploration 199
6.4.2 Implementation details 201
6.5 Hardware Support for Mixed-Precision Inference 204
6.6 Evaluation and Discussion 207
6.6.1 Selected Workloads and Test Setup 208

6.6.2 Evaluation of Mixed-Precision to Enhance Energy Efficiency and Robustness . . . 210
6.6.3 Hardware Considerations for Mixed-Precision Inference . . . 216
6.6.4 Discussion . . . 218
6.7 Conclusion and Outlook . . . 219

7 Summary, Conclusion and Outlook 221
7.1 Concluding Remarks . . . 221
7.2 Outlook and Future Work . . . 224

Bibliography 225

Publications 251

List of Figures 255

List of Tables 259

Listings and Algorithms 261

Acronyms

AGU Address Generation Unit.
AI Artificial Intelligence.
ALU Arithmetic / Logic Unit.
ANN Artificial Neural Network.
AP Average Precision.
ASIC Application-Specific Integrated Circuit.

BLAS Basic Linear Algebra Subprograms.
BNN Binarized Neural Network.

CE Cross-Entropy.
CMOS Complementary metal-oxide-semiconductor.
CNN Convolutional Neural Network.
CONV Convolutional.
COTS Commercial-Off-the-Shelf.
CPU Central Processing Unit.
CSC Compressed Sparse Column.
CSR Compressed Sparse Row.
CSV Comma Separated Values.
CV Computer Vision.

DMA Direct Memory Access.
DNN Deep Neural Network.
DRAM Dynamic Random Access Memory.
DSE Design Space Exploration.
DSP Digital Signal Processor.

EDDO Explicitly Decoupled Data Orchestration.
eFPGA embedded Field Programmable Gate Array.

EPI European Processor Initiative.

FC Fully Connected.
FLOP Floating-Point Operation.
FPGA Field-Programmable Gate Array.
FPN Feature Pyramid Network.
FPU Floating Point Unit.

GA Genetic Algorithm.
GEMM General Matrix Multiply.
GPGPU General-purpose computing on graphics processing units.
GPP General-Purpose Processor.
GPU Graphics Processing Unit.

HDL Hardware Design Language.
HLS High Level Synthesis.
HPC High-Performance Computing.

ILSVRC ImageNet Large Scale Visual Recognition Challenge.
IoU Intersection over Union.
IP Intellectual Property.
IS Input Stationary.
ISS Instruction Set Simulator.

LB Logic Block.
LSTM Long-Short Term Memory.
LUT Look Up Table.

MAC Multiply-Accumulate.
MAE Mean Average Error.
ML Machine Learning.
MLP Multilayer Perceptron.
MSE Mean Squared Error.

NAS Neural Architecture Search.
NLP Natural Language Processing.
NoC Network-on-Chip.
NPU Neural Processing Unit.
NSGA Non-dominated Sorting Genetic Algorithm.

ONNX Open Neural Network Exchange.
OS Output Stationary.

PCA Principal Component Analysis.
PE Processing Element.
PPA Performance, Power and Area.
PTQ Post-Training Quantization.

QAT Quantization Aware Training.

ReLU Rectifier Linear Unit.
RL Reinforcement Learning.
RNN Recurrent Neural Network.
RTL Register Transfer Level.

SIMD Single Instruction Multiple Data.
SNN Spiking Neural Network.
SoC System-on-Chip.
SOTIF Safety Of The Intended Functionality.
SRAM Static Random Access Memory.

TLM Transaction Level Modeling.
TPU Tensor Processing Unit.

VHDL VHSIC Hardware Description Language.

WS Weight Stationary.

Chapter 1

Introduction

The last 70 years were defined by exponentially rising advances in computing technology, which opened increasing technological possibilities. One important technology that is currently moving into humanity's reach are self-learning and Artificial Intelligence (AI) systems. Implemented using highly sophisticated algorithms they start to show tremendous performance in an increasing number of areas, from autonomously driving vehicles, over personalized and targeted medical treatment to improved and more comfortable user interfaces. Although AI research already started in the 1950s with the advent of computing technology, it took another 60 years and many technological leaps to make AI algorithms applicable to real-world problems. Nowadays, most AI systems are carried out as Deep Neural Networks (DNNs), which resemble the structure of biological brains. Millions of neurons, with billions of connections enable DNNs to solve increasingly challenging tasks. In most, they even outperform human capabilities by orders of magnitude.

This trend is notably highlighted by the advances in Computer Vision (CV). A common benchmark is image classification, using the dataset provided by the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It features 1000 classes of images with over one million training samples. In 2015, for the first time a DNN model achieved human-level performance in distinguishing those image classes [1]. Since then, the classification accuracy is rising steadily with the development of more sophisticated DNNs. DNN classification performance is measured as Top-1 accuracy, which gives the percentage of unambiguously correct classification results. The recent development of DNNs from 2014 until today is shown in Figure 1.1. It also reveals by what means the growing accuracy is achieved. The bubbles depict the amount of Floating-Point Operations (FLOPs) it takes to compute a single classification result. Vision-Transformers [2] reach just over 90 % Top-1 accuracy, but one prediction

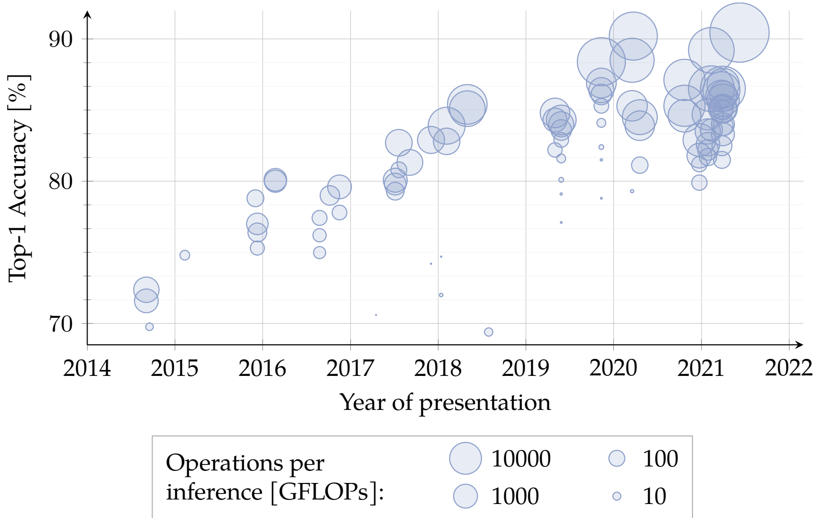


Figure 1.1: Development of the parameter count and FLOPs required for a single inference for different CV neural networks, which were presented from 2014 until 2022; visualization adopted from [4].

requires 5270 million FLOPs. In comparison, the network that first outperformed humans in image classification, GoogLeNet [3], achieves about 70 % accuracy, with 3 million FLOPs, 1750× fewer operations. However, applications like robotics or autonomous vehicles require very accurate and reliable results to operate as expected and to eventually gain acceptance by customers.

Modelling and executing DNN models with hundreds of million connections and neurons, requires a vast number of computations and huge amounts of memory. The current development pace builds on a computational performance doubling in every six-month period [5]. At the beginning of DNN model development, shrinking rates of transistors and the associated performance and energy efficiency gains could keep pace with the growing demands of the algorithms. However, Dennard Scaling poses a major challenge to modern silicon technology. Compressing an immense number of transistors onto a single chip, yielded diminishing returns over the years. The tight integration of all the switching transistors, opens two major issues for the next generation computing systems. All newly added transistors are hard to utilize efficiently, and the miniaturization leads to massive heat accumulation, which must be dissipated.

This creates a so-called utilization wall. To compensate for the diminishing returns of silicon integration, a paradigm shift happens for DNN systems. Around 2010, the computation of DNN models moved from General-Purpose Processors (GPPs) to Graphics Processing Units (GPUs), since frequency increases of Central Processing Units (CPUs) slowed down and GPUs offered great support for parallelization of the highly independent operations in DNNs. A couple of years after, DNN computation moved to dedicated hardware platforms with accelerators that utilize the available transistors and power budget even more efficiently.

Today, a wide range of different compute platforms for DNNs are available, from modern smartphone System-on-Chip (SoC) with neural processing engines for better user experience [6] to high-performance datacenter units. These achieve a throughput far beyond 10 million million FLOPs per second [7], depending on the complexity of the DNN model. However, complex engineering challenges, like autonomous vehicles or humanoid robots, still require more efficient hardware platforms. While the algorithms for those tasks are not ready yet, if we have a look at Figure 1.1, it is foreseeable that the requirements for energy efficiency and throughput will rise further.

1.1 Motivation and Context

The recent development progress of DNN models shows a clear trend towards larger and more comprehensive, but also more performant models. Execution of those complex workloads is heavily supported by dedicated DNN accelerators, of which multiple hundreds were presented in the recent decade. All approaches try to overcome the utilization wall by adding smart optimization techniques. These start with novel scheduling methods for DNN workloads, which are referred to as tiling and mapping, and end with quantization and pruning. The latter two leverage one particularly interesting characteristic of DNNs, namely their probabilistic nature. For example, an image classifier tells us the likelihood that an input image falls into a specific category. Now, if due to approximate computations, this likelihood slightly differs, but the class of the prediction remains the same, the algorithm continues to work as intended. Quantization and pruning introduce this kind of computational imprecision that make many operations simpler with only little effect on the prediction accuracy.

Due to Dennard Scaling, simply adding more processing units to DNN accelerators became infeasible to increase throughput. Consequently, the interplay of DNN model optimization techniques, hardware architectures and DNN model accuracy, is currently vigorously researched. Huge leaps in overcoming the utilization and power walls could be made with efficient DNN accelerators that feature, e.g., dynamic online pruning, mixed-precision quantization or reconfigurable computational units. Obviously, a high throughput DNN computation is highly relevant for large-scale datacenter applications. However, we have left embedded systems like platforms for autonomous vehicles aside for now. Besides a reasonable throughput, they pose many other requirements on DNN accelerators. First and foremost, energy consumption and latency, as they are often battery powered, and have to, e.g., perceive the environment within a constant and short amount of time. Especially this environment perception is often done using Convolutional Neural Networks (CNNs), a subtype of DNNs for CV tasks, which have a vast memory footprint for their intermediate feature maps. This characteristic makes the design of an efficient accelerator even more challenging. In addition to this, other requirements like a reliable and safe inference are eminently important for, e.g., autonomous robots or vehicles.

Of course, many research works already tried to find answers to the big question on how does an efficient CNN model and a corresponding accelerator for highly constrained embedded systems looks like. Although, great progress was made and is currently ongoing, some questions are still yet unanswered.

1.2 Challenges, Proposed Solutions and Contributions

In this thesis, we will discuss some key challenges for DNN accelerators in constrained embedded systems. Thereby, we will address them by optimizing the DNN inference. Although, a good deal of research in the area has already been carried out, there are still a lot of open questions. For example, how to design an efficient DNN accelerator in terms of throughput, energy efficiency, latency, and chip area. This accelerator also has to keep pace with the rising parameter count, which is still growing to unforeseeable sizes. However, hardware development is much more time-intensive than model design and training. Moreover, it is still unsolved how we can make DNN predictions more reliable both from an algorithmic and hardware perspective. Finally, we have to figure out how to lay out a DNN model well-suited for a given accelerator that delivers high accuracy and reduces at the same time the parameter count.

All those questions open ways for many solutions, approaches and starting points. Consequently, the current research community is following different ways to achieve the same goal. With the work that was carried out and is described in this thesis, we make three general contributions to the current state of the art. All our contributions focus on embedded systems and respect the associated constraints like area limitations or strict energy budgets. As a result, each of our approaches aims to optimize the trade-off between energy efficiency to throughput. In particular, we looked at how we can reduce the numerous energy demanding off-chip memory accesses. To see this in action, we focused our work especially on CNN workloads, as they have large memory footprints and offloading to external memories happens often. Therefore, we chiefly investigated hardware methodologies and stuck to commonly used benchmark CNN workloads, which also allows for a better comparison to state-of-the-art works. Hence, altered model topologies or approaches like Neural Architecture Search (NAS) are only touched briefly.

Algorithm and accelerator co-design In our first contribution, we tried to answer the question on how we can design a DNN accelerator tailored for a given model and set of requirements. Therefore, we introduced two simulation platforms to assess DNN accelerators in terms of Performance, Power and Area (PPA), and to perform a Design Space Exploration (DSE) to identify accelerator configuration well-fitted for the target workload. First, we presented a cycle-accurate simulation framework called FLECSim. It allows modelling entire SoC designs using SystemC or Register Transfer Level (RTL) modules to enable a quick time to market and an early verification of possible designs before actual chip production. Second, we describe our analytical model approach, which generates PPA estimates much quicker, since it uses the Roofline model instead of time-consuming cycle-accurate simulation. This is feasible because DNN workloads have a highly homogenous dataflow. Both simulation tools are demonstrated with common benchmark workloads. Here, we also proved that our models reflect real-world performance numbers. For the analytical model, the deviation is less than 1 % off, but with orders of magnitude faster execution time. With our two approaches, we finally conducted two case studies, which showed their flexibility and revealed useful insights about how various DNN accelerator characteristics affect the PPA figures. For example, FLECSim was used to lay out an embedded Field Programmable Gate Array (eFPGA) design and our analytical model for camera-based environment perception.

More in Chapter 4. Relevant publications: Hot+20; Hot+21; Hot+22b; Hot+23c.

Exploitation of regular sparsity in activation feature maps of CNNs In this and the next contribution we investigated ways to make the actual DNN inference more efficient in terms of energy and throughput. One promising method is pruning of activations that only contribute little to the model's result. Therefore, we designed a modern regular activation pruning method. Our approach targets sparsity in the input feature map in the form of blocks that have the same size as the underlying hardware accelerator. Thus, we can prune with only little extra hardware, which makes the pruning highly efficient compared to index-based irregular pruning schemes. Since sparsity in regular blocks is only rarely present in modern CNNs, we introduced a technique and tool to increase it by using threshold-based magnitude pruning. Our tool Spex explores the search space of accuracy and number of sparse blocks automatically and yields a favorable magnitude pruning threshold for each layer. With this we can prune all activations that contribute nothing or only little to the result with our hardware extension Sparse-Blox, which aims to have only a small hardware overhead. In comparison to state-of-the-art irregular pruning approaches, Sparse-Blox hereby requires up to 5× less area. Pruning those activations allows us to skip unnecessary Multiply-Accumulate (MAC) operations and memory transfers, which result in large energy savings. Considering a ResNet-50, we can, e.g., prune about 20 % of all activations. This saves about 800 million MAC operations and over 19 % of the inference energy.

More in: Chapter 5. Relevant publications: Hot+22a; Hot+23a.

Increasing robustness and energy efficiency using mixed-precision inference In the third contribution, we analyzed how quantization can be utilized for increased efficiency and robustness. Especially mixed-precision inference was shown in multiple studies to settle at a sweet-spot between model accuracy and energy efficiency. Parts of the model that are sensitive to quantization may, for example, be executed with higher precision. In turn, others are run with less precision to save energy. To determine the sweet-spot, we extended Spex to search also for per-layer precision. Since mixed-precision is typically hard to implement in hardware, we also presented a new method for arbitrary precision quantization without the need to alter the entire DNN accelerator. Our approach uses a reconfigurable fabric, present in most modern SoCs, to host a quantization and dequantization unit. These units compress data moving from or to the off-chip memory to save energy during costly memory accesses. The introduced mixed-precision is evaluated with state-of-the-art image segmentation CNNs. We found that it can save up to 6 % energy with negligible resource consumption on the reconfigurable unit. Besides that,

we investigated how adjusting the precision can restore the model accuracy in case of perturbed inputs. This is of particular importance, when considering safety-critical embedded systems like environment perception in autonomous vehicles. Therefore, we evaluated image datasets corrupted with environment conditions like rain or fog and demonstrated that mixed-precision can support the robustness of CNNs in these scenarios. We demonstrated, e.g., that severe rain can be mitigated by increasing the precision slightly. In this case, we can still save up to 45 % energy by compressing off-chip memory transfers, since only the perturbation sensitive part of the model is executed with higher precision.

More in: Chapter 6. Relevant publications: Hot+23b.

To enable subsequent work and transparency on the results, our model exploration and optimization tools are made available as open-source to the public, under the following link: https://github.com/itiv-kit/dnn-model-exploration.

1.3 Outline

To get an overall impression and understand of the topic, the next chapter (Chapter 2) will introduce all required principles. Here we will cover DNNs and CNNs, their function and building blocks, popular datasets and DNN models, as well as basics of hardware architecture design and finally how to utilize dedicated hardware architectures to make DNNs more efficient. When the principles are explained, we can dive deeper into the state of the art, which is covered in Chapter 3. This chapter introduces notable DNN hardware accelerators along with common optimization techniques that are today widely used to increase the inference efficiency. In addition, we will talk about the software ecosystem that is also essential for highly efficient DNN computation.

In Chapter 4 we will tackle one of the most challenging questions in DNN accelerator design, which is what accelerator configuration is the best for a given application. Therefore, we present FLECSim, a cycle-accurate simulation SoC framework and its companion, an analytical approach. In combination, both can support the design process and help to figure out how certain design decisions impact the inference performance. To increase the performance further, Chapter 5 introduces Spex and Sparse-Blox, a tool and hardware extension to increase and prune regular activation sparsity in DNNs. With this method, we can save up to 20 % of all computations

and inference energy in a model. Quantization is also a great way to boost the efficiency. Hence, Chapter 6 shows how mixed-precision cannot only reduce energy consumption of the hardware architecture, but also help to increase the robustness against perturbed inputs. Using this approach, we can restore the accuracy of a CNN with inputs perturbed by severe rain and at the same time reduce energy-expensive off-chip memory accesses by 45 %. Finally, Chapter 7 wraps up our work and discusses potential future research paths.

Chapter 2

Basic Principles and Fundamentals

This chapter introduces basic principles as well as theoretical concepts that are necessary for further understanding of this thesis. First, we give a brief overview of Machine Learning (ML) in general, before diving deeper into neural networks and especially Deep Neural Networks (DNNs) with all their components and building blocks. A particular focus is put on Convolutional Neural Networks (CNNs) and how they leverage spatial information and thus work very well in Computer Vision (CV) tasks like image classification, object detection or semantic segmentation. We then introduce commonly used benchmark network topologies and datasets in the world of DNNs. With an agreed set of benchmarks, results of different architectures can be compared among each other. In the second half, this chapter will cover the foundations of hardware architectures and hardware accelerated neural network computation. Therefore, we first present an overview of common hardware architectures as well as general hardware development methods such as design automation and hardware / software co-design. This chapter closes with hardware accelerators for DNNs and why dedicated DNN accelerators support faster and more energy-efficient computation.

2.1 Introduction and History

Recent advances in computational capacity have enabled execution of highly demanding algorithms such as complex real-time simulations or multidimensional optimization in a reasonable amount of time. This led to the development of novel ML algorithms like Artificial Neural Networks (ANNs). In general, the idea behind self-learning systems is not entirely new. The first approaches towards machines that can adapt to situations based on experience came up around 1960 by pioneers like

Arthur Samuel, Marvin Minsky, or Frank Rosenblatt. They introduced, for example, the first perceptrons [8] or self-learning applications [9]. However, early machine learning research had to face the limits of computational performance at that time. As a result, the effectiveness of machine learning algorithms was questioned. After a period of diminished interest, Artificial Intelligence (AI) development regained attention in the mid-1980s with the discovery of the backpropagation algorithm [10]. It helps to determine the complex estimation of weights in neural networks. However, for larger networks it is highly computationally challenging. Hence, given the computational requirements of backpropagation, AI research fell out of favor once again due to a lack in computational performance at the end of the 1980s [11].

Although, some few researchers continued to follow the AI research path, it took yet another twenty years until further progress on the topic was made. Around 2010 neural networks and the backpropagation algorithm became popular again, benefitting from the rise of widely available and powerful Graphics Processing Units (GPUs). GPUs also gave a head-start to deeper and more complex neural networks. This started and coined the term *deep learning* around that time [12]. Since the last decade it allows solving a growing amount of increasingly difficult tasks. Today's exemplary applications range from healthcare over robotics to autonomous vehicles. CV algorithms for medical imaging based on DNNs outperform human doctors in detecting skin cancer [13]. Face recognition algorithms based on CNNs enable robots to interact with the elderly in a personalized and human-friendly way [Wal+21]. Even data acquisition systems can benefit from DNNs. For example, in the large scale physics experiments Belle-II, a DNN detects and compresses data from particle collisions [14]. These mere three applications already show the wide range of applications that can be covered with DNNs. In general, it is conceivable to deploy DNNs in almost any application, especially when modelling the problem with traditional algorithms is too complex and enough data for training and validation is available.

2.1.1 Artificial Intelligence and Machine Learning

ML is one of the most important subfields of the more general field Artificial Intelligence. The general goal of ML is to build an algorithm that can understand a given dataset [16]. An algorithm might predict results of unknown data or extract valuable information from given input data. We can divide machine learning algorithms into three categories based on the nature of the data and on the way the algorithm is

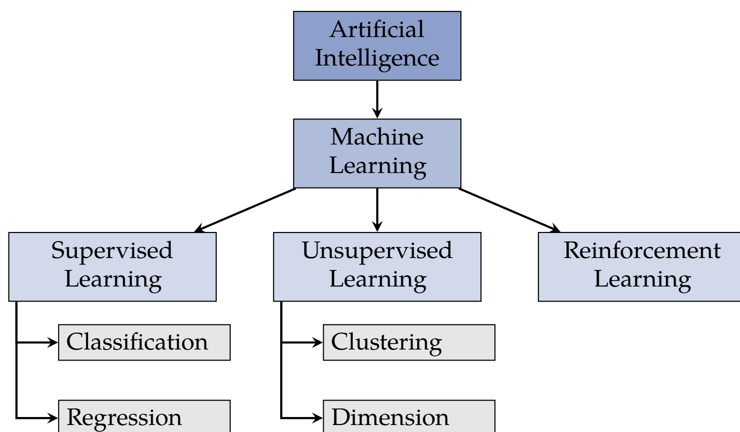


Figure 2.1: Overview of the distinct types of machine learning and the typical workloads that can be solved using the given type, according to [15].

trained: Supervised, unsupervised and reinforcement learning. An overview of the different types and their classification is given in Figure 2.1.

Supervised learning algorithms require input data that has target labels, which represent an expected prediction. The aim of supervised learning is to determine a function that maps inputs to the associated target labels. Once an algorithm is trained on a large set of labeled data, it can be applied to unknown data to predict an output that matches the trained labels. Based on the output either classification or regression can be performed [16]. The first aims to group inputs into distinct classes, whereas the latter tries to find the relation between inputs and outputs, and determines regression coefficients, which describe a linear or even more complex regression function. Hence, supervised learning is an extremely popular way to tackle challenges like image classification, natural language processing or object detection.

Unsupervised learning, in contrast, operates on unlabeled data. Its goal is to extract information or find patterns in large unstructured datasets. One common type of unsupervised learning algorithms are clustering methods. Examples are centroid based methods like k-Means [17] or density-based methods like DBSCAN [18], just to name two. Even different algorithms put emphasis on various aspects of the data, all aim to group unstructured data into clusters based, e.g., on their Euclidean distance. Besides clustering, dimension reduction methods like Principal Component Analysis

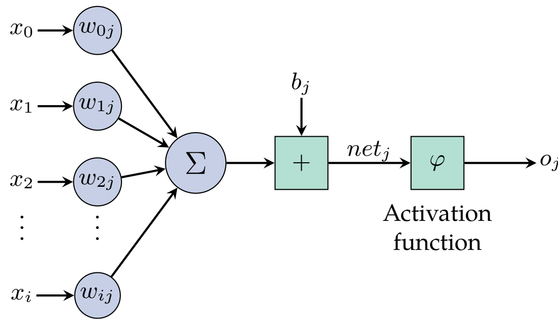


Figure 2.2: Overview of a single artificial neuron, featuring a set of inputs and weights, one bias from which an output is computed; visualization from [21].

(PCA) [19] help to extract information from unstructured data by mapping data with high a number of dimensions to fewer based on their variance. This class of algorithm can eliminate features that contain none or only little relevant information.

Reinforcement Learning (RL) [20] works slightly different compared to the other algorithms. The main principle of RL is that an intelligent agent interacts inside an environment and thereby learns beneficial actions that it can take in this environment. A reward function guides the entire process towards a beneficial outcome. The agent tries to maximize the reward, and therefore tests different actions in a trial-and-error fashion. RL does not require labeled data, instead it learns a general set of rules based on the defined environment. Common applications of RL are robot control or playing games since the environment for those applications can be well described.

2.1.2 Artificial Neural Networks and its Building Blocks

ANNs are nowadays the most popular type of machine learning algorithm due to their versatility. Supervised, unsupervised and reinforcement learning tasks can all be tackled with ANNs. Already in the late 1940s first approaches to mimic the organization of biological brains and neuron structures were presented [22]. Even though biological brains are not fully understood up to the present day, some basic principles are known and were brought into computer science to form an artificial counterpart. As stated before, the idea of ANNs is not entirely new, however, only modern computer hardware is capable of handling multiple million neurons. A single artificial neuron that resembles a biological neuron to some extent is shown in

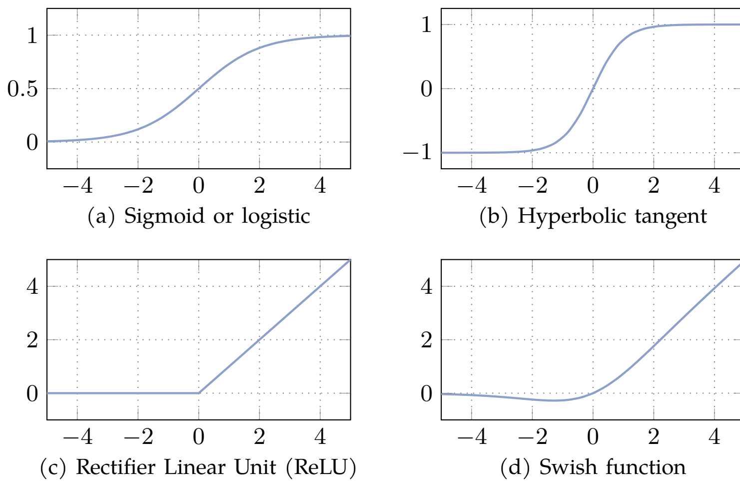


Figure 2.3: Four common and modern examples of activation functions for ANNs, according to [23].

Figure 2.2. In principle, a neuron with an index j computes an output o_j. Therefore, a set of inputs $x_0, \ldots, x_i$ is multiplied with the corresponding weights $w_{0j}, \ldots, w_{ij}$. The product of inputs and weights is then summed up. The result represents a linear combination of all inputs based on their weight. Hence, it allows to construct a curve in two-dimensional space, which is parametrized by the weights. To shift this curve representation on the y-axis, a bias b_j is added, yielding the neuron value net_j.

However, ANNs should represent complex non-linear functions. Hence, at the end a non-linear activation function is applied to net_j. For various applications, different activation functions are available. A selection of commonly used functions in shown in Figure 2.3. One of the first successfully deployed activation functions is the Logistic or Sigmoid activation [24] (see Figure 2.3a). It is frequently used in statistics. The main benefit is a constrained range $(0, 1)$, which limits the overall result of an individual neuron. In addition, Sigmoid smoothens input values around zero in contrast to a simple step function. Similar to Sigmoid, a hyperbolic tangent can be used to create similar behavior, but allowing for negative values (see Figure 2.3b). Many state-of-the-art neural networks nowadays use the Rectifier Linear Unit (ReLU) activation [25; 26] (see Figure 2.3c). Although, it neither limits the range nor smoothen the output value, its benefits become apparent when millions of neurons have to be processed in large

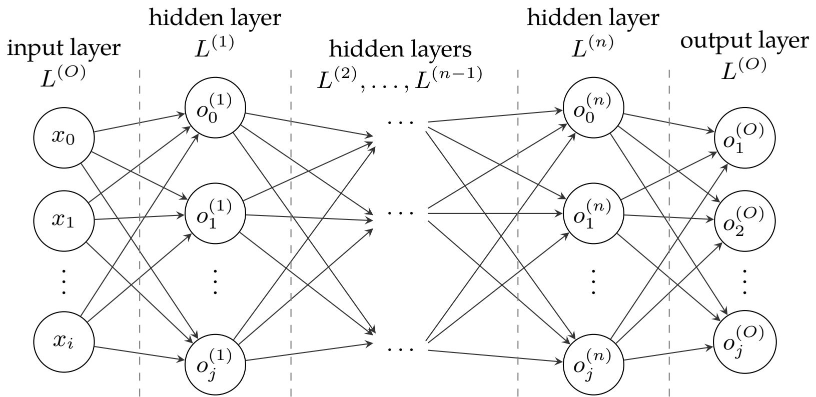


Figure 2.4: Overview of a Multilayer Perceptron (MLP) with i inputs, j outputs and n layers, according to [28]. Each connection represents a weighted vertex and each node a summation, with bias and activation function.

networks, as their computation is very straightforward using a simple comparison [27]. The traditional ReLU activation can, however, create difficulties in very large and deep networks, since negative intermediate results are neglected during the training process, which may lead to a non-converging network training. Hence, very recent networks use modifications of ReLU, like the Swish function [26] (see Figure 2.3d). It is still simple to compute, but also allows for negative activations.

Putting all components of an artificial neuron together, we can compute its output o_j as follows using Equation 2.1:

$$o_j = \varphi(net_j) = \varphi\left(\sum_{i=0}^{N-1} (x_i * w_i) + b_j\right) \tag{2.1}$$

To form a complex neural network, we can now simply put multiple of these simple neurons together. In ANNs neurons are grouped together into layers, based on which neurons provide the inputs for the next. Layers can be stacked upon each other, in which the output of a layer L^n represents the input of the next layer L^{n+1}. The connections in between layers, can be varied: Fully Connected networks connect each output to all neurons of the subsequent layer and sparse networks only connect selected neurons. Connections do not necessarily need to follow a straight acyclic directed graph. As a result, besides feed forward networks, Recurrent Neural Net-

works (RNNs) exist, which also feature cyclic connections back to preceding layers. Networks with multiple layers are often called either MLPs or DNNs. Layers of the network itself, are considered hidden layers. Besides them input and output layers represent external data input and the final network output, respectively. An example of a simple fully-connected MLP with an input and output layer is given in Figure 2.4. Looking at the computation of a single Fully Connected (FC) layer, we can represent the weights, inputs, biases, and outputs as matrices. Equation 2.2 shows the general equation for a single layer with i inputs and j outputs. In this example, the network is fully-connected, however, sparse connections can be represented by zeros in the weight matrix.

$$\begin{bmatrix} net_0 & net_1 & \cdots & net_j \end{bmatrix} = \begin{bmatrix} i_0 & i_1 & \cdots & i_i \end{bmatrix} \times \begin{bmatrix} w_{00} & w_{01} & \cdots & w_{0j} \\ w_{10} & w_{11} & \cdots & w_{1j} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n0} & w_{i1} & \cdots & w_{nj} \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \cdots & b_i \end{bmatrix} \tag{2.2}$$

Each neuron in the output layer of a regression ANN contains a parameter of the associated regression function. However, for classification some further considerations have to be considered. It does not work with continuous output values, as ambiguous result can occur. To solve this, classification networks each output neuron represents a specific class. This is referred to as one-hot-encoding. Here, an optimal prediction would contain maximum activation at a neuron that represents the target class, while all others show no activation. To achieve this, the SoftMax activation function is used for the output layer. It limits the continuous output of each neuron to a range of $[0, 1]$ with a sum of 1 over all output neurons. As given in Equation 2.3: With K classes, SoftMax weights the output net_j of neuron j according to all output activations. As a result, the output of each neuron can be interpreted as probability that a prediction falls into a specific category.

$$\varphi(net_i) = \frac{e^{net_i}}{\sum_{j=1}^{K} e^{net_j}} \tag{2.3}$$

With multiple layers ANNs started to solve more complex problems. For example, in 1989 a network with three hidden layers was able to recognize handwritten ZIP

codes with less than 1 % error rate [29]. Over time and with more sophisticated hardware support for neural networks, the number of feasible layers in deep learning grew massively [30], enabling more challenging tasks like image classification or speech recognition. Therefore, in the 2010s many other forms of neural network layers, connection pattern and neuron behaviors were introduced. For instance, CNNs which apply convolutions on the inputs and consider spatial information, Long-Short Term Memories (LSTMs), a sub-category of RNNs, which show a very good performance in prediction of time-series data, due to their build-in memory or Transformer models, which are the backbone for large language models that show tremendous performance on Natural Language Processing (NLP).

2.1.3 Training of Artificial Neural Networks

ANNs have to be trained beforehand to predict expected results. The goal is to optimize all weights and biases of the ANN so that they match the expected result as close as possible [23]. As discussed before, individual neurons in a neural network represent linear functions, which in combination can form complex mappings from inputs to outputs. Hence, training of ANNs can also be formulated as a multivariable function optimization problem, well-known in the world of stochastic optimization. In general, the search space of modern DNNs is quite complex. Mathematicians already found solutions to keep this design space in bounds, using effective algorithms like gradient decent. Usually, neural networks are initialized with random parameters. These weights are now iteratively optimized in a training loop that tunes the network towards its target function.

The current status of the training process and closeness to the optimization goal is formulated as cost or loss function C, which in simple terms gives a measure of the network's error. Loss functions are chosen based on the task. For regression tasks Mean Average Error (MAE) and Mean Squared Error (MSE) are used (Equation 2.4). Both compare the predicted outputs of a neuron o_j to the expected target outputs t_j. The first function weights the resulting error on average over all values and the latter penalizes greater errors stronger by squaring the difference.

$$MAE = \frac{\sum_{n}^{j=1} |o_j - t_j|}{n} \qquad MSE = \frac{1}{n} \sum_{n}^{j=1} (o_j - t_j)^2 \tag{2.4}$$

Classification problems are normally trained with Cross-Entropy (CE) loss (see Equation 2.5 left). It penalizes wrong classification results with a logarithmic magnitude, leading to a higher error for predictions further apart from the target and smaller error for closer predictions. For one-hot encoding t_j only has one non-zero value at the location where the target is. Considering this and replacing o_j with the SoftMax activation function, we can simplify Equation 2.5 to compute the loss value for a neuron at a given position p (see right part).

$$CE = -\sum_{j=1}^{K} t_j log(o_j) \qquad CE = -log\left(\frac{e^{net_p}}{\sum_{j=1}^{C} e^{net_j}}\right) \tag{2.5}$$

Now that we can quantify the model's error on a given input, we can proceed with the next step, optimizing the network parameters towards a good prediction performance, i.e., minimizing the loss. As stated before, this kind of optimization problem can be solved by gradient-based optimization algorithms. A quite simple example is gradient decent. Besides that many other optimization algorithms appeared over the last years, some especially for ANN training [31; 32]. The loss function of every neural network spans an n-dimensional plane that has at least one optimal solution, which represents an optimal set of parameters. Now, the objective is to find this solution and not getting stuck in local minima. Calculus can tell us in which "direction" along the n-dimensional loss plane we have to move to get closer to the desired optimal solution. Therefore, we have to compute the derivate of the loss function at a given position and subtract the magnitude from the parameters to obtain a new set of parameters. However, determining the derivate of the loss function regarding each weight becomes a series of nested terms for large and deep ANNs.

To overcome this complexity while computing the derivate, an algorithm called *backpropagation* [10] is employed. It computes the gradient and derivate of the loss function for each weight in an iterative manner, denoted as $\partial E/\partial w_{ij}^L$. The advantage is that we can break the derivate into individual steps with partial derivates, as given in Equation 2.6. All parts can be computed separately and be reused for other weights. This works analogous to the forward pass as shown in Figure 2.2 but the other way round: E.g., the first term is the derivate of the loss function with respect to the network output, the next is the derivate of the weighted sum, and so on. Backpropagation now reuses terms, which were already computed for each weight instead of computing all partial deviates individually, making the entire process much more efficient.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial b_j} \cdot \frac{\partial b_j}{\partial w_{ij}} \tag{2.6}$$

If we extend the example of a single layer to an entire MLP, we can apply the chain rule to append all layers. To avoid duplicate computation of the layer outputs the backpropagation algorithm reuses their outputs as well and propagates only an error vector for each layer. Therefore, intermediate outputs computed during the forward pass have to be cached. Hence, for large models a lot of memory has to be provided for training.

Finally, updating the weights to minimize the error is done by subtracting the partial error with respect to a given weight. However, this will lead to a non-converging optimization as too steep weight changes are applied. Hence, a learning rate η scales the magnitude of the weight update. Typical learning rates are chosen somewhere between 0.01 to 0.0001; usually the learning rate is lowered during the training process to support fine-tuning at the end [33]. Combining all those parameters results in the following equation, showing how an individual weight is updated:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{2.7}$$

Learning rate is considered as hyperparameter. Hyperparameters define the structure and properties of a DNN. The most notable hyperparameters are the number and kind of layers, and neurons per layer. Sometimes the term is even defined more broadly and includes aspects like the selected optimization algorithm or the type of activation function.

Besides learning rate, other methods to support fast convergence and proper fitting to the training dataset are applied: First to mention are regularization strategies that prevent the network from overfitting, e.g., early stopping [34], which tracks the training progress and stops when the network performance starts to worsen. Second, batch normalization to limit the magnitude of gradients and errors that result in large networks [35]. Among others, those two methods are routinely used for fast and goal-directed DNN training.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of ANNs that chiefly consist of *Convolutional (CONV) layers* instead of FC layers. They work especially well on input data that features spatial information, like images or videos. The first basic structure of a CNN the *neocognitron* was presented already in 1980 [36] and was inspired by the visual cortex of mammals. However, it took another ten years for LeNet-5 [37] to appear, which outperformed traditional MLPs in handwritten bank check number recognition. Although, CNNs demonstrated their performance, computational power was simply not sufficient at that time to solve large image recognition tasks. As a result, research in CNNs almost came to a stop, until in 2006 the first GPU supported CNN implementation was presented by Chellapilla et al. [38]. This fast implementation, gave rise to more sophisticated models like AlexNet [39] in 2012. Training this model with eight layers was only made possible through the advances in GPU utilization. AlexNet, as a result, won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), an image classification challenge hosted by Microsoft, with about 15 % error rate. Over the last decade, even deeper models with today over 150 layers [40] paved the way to an error rate of only a few percent, which outperforms humans.

The applications of CNNs include nearly all conceivable tasks, which involve temporal or spatial information. In computer vision, CNNs are used for image classification [40; 41; 42], object detection [43; 44] as well as instance [45; 46; 47] and semantic segmentation [48; 49; 50]. CNNs, in general, can work with data of any dimension. Using one dimensional signals they detect anomalies or extract data, e.g., in Electro Cardio Grams (ECG), where they help to identify arrhythmic heart beats [51]. Two-dimensional inputs are usually images. Application examples range from healthcare, where CNNs help to identify disease from chest X-Ray images [52] or in skin cancer detection, in which the algorithms already outperform human doctors [53]. Other important domains are autonomous driving or autonomous robots. CNNs as object detection networks and segmentation networks play a significant role in environment perception [50; 54]. For example, a CNN can detect pedestrians for autonomous vehicles [55] or evaluate and fuse data from multiple sensors like cameras and LiDARs [56]. A third popular domain for two-dimensional CNNs is face recognition. For example, FaceNet [57], which extracts essential information from any given face and hence can be generalized to new faces quickly, or DeepFace [58], which outperforms humans in distinguishing persons. These three application examples only give a glimpse into the domains that can be addressed by two-dimensional

CNNs. Three-dimensional CNNs are found to process video data, e.g., for human pose estimation [59]. Input data for pose estimation usually has both temporal and spatial information, which is extracted by the three CNN dimensions.

Modern approaches of neural networks in CV nowadays typically combine several types of ANNs. For example, a combination of LSTMs and CNNs layers is beneficial to take both spatial and temporal information into account, with less overhead compared to three-dimensional CNNs. FC layers are added for classification tasks, to combine information that at the end of the CNN into classification results.

2.2.1 Convolution Layer

The main component of CNNs are CONVs layers. In contrast to either fully or sparsely connected layers, which have individual weights for each connection, CONV layers are typical sparsely connected and share weights with different connections. Weights are grouped in convolution kernels, which consist of stacked filters. Weight sharing happens though the convolution operation, in which a kernel is moved along the spatial dimensions of the input. In each movement step, a so-called receptive field of the input is pointwise multiplied with the filter, creating an output value. On this a bias and an activation function is applied, analogous to other types of ANNs. The resulting output value is then placed at the corresponding position in an output feature map. Filters are reused for the whole input, which leads to fewer parameters, compared to MLPs. A simplified example of a two-dimensional convolution operation as it is performed in CONV layers is given in Figure 2.5. Here, a two-dimensional 3×3 kernel moves along the width and then the height dimension of the input feature map. The first receptive field of the input data and the corresponding output is shown in blue, the second in green. The size of convolution kernels is typically much smaller than the size of the input. Common sizes are 3×3, 1×1 or any odd number. Odd numbers cover an uneven portion of the input feature map with a clear center point for the output feature map, which is favorable for CNNs.

With this basic principle, CONV layers can extract information from inputs that are spatially close to each other. Due to the fewer weights, they can be trained efficiently. FC layers, in contrast, combine information from all inputs at once and have many more weights. Training MLPs for image classification thus takes a lot of time, and it tends to overfit because many small weights are created. CNNs regularize the training by reducing the overall number of connections.

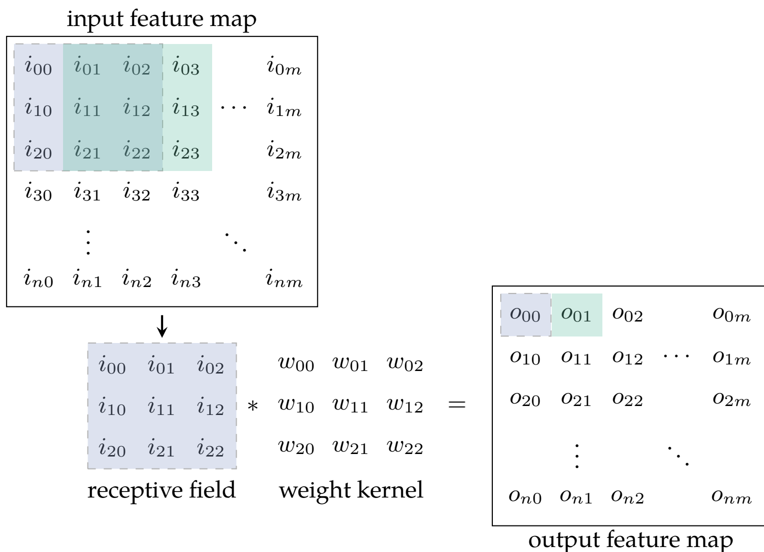


Figure 2.5: Principle of a two-dimensional convolution with input i, weights w and output o, according to [21].

Modern CNNs for image processing feature multiple CONV layers. Why multi-layer CNNs work especially well, can be easily understood: After successful training, the first couple CONV layers usually extract simple vertices and edges from the input image. Hereby, they also discard noise or brightness differences. In doing so, pretrained kernels filter input images and are hence also referred to as convolutional filters. To extract diverse types of vertices and edges, a CNN has multiple filters that are applied to the same input, but with different weights. Going deeper into the CNN, CONV layers condense information from intermediate results. Different simple vertices are put together to form more complex shapes. The higher an activation after a CONV operation, the higher the probability for the presence of a learned feature. Since, the filter is moved across the input, spatial information is preserved. In instance segmentation or object detection tasks, this information is used to determine the position of an object. Image classificators, however, drop this information, but combine the likelihood of different complex shapes to determine the image class. This last step is usually done by one or two fully connected layers, which combine information from all inputs into a single output.

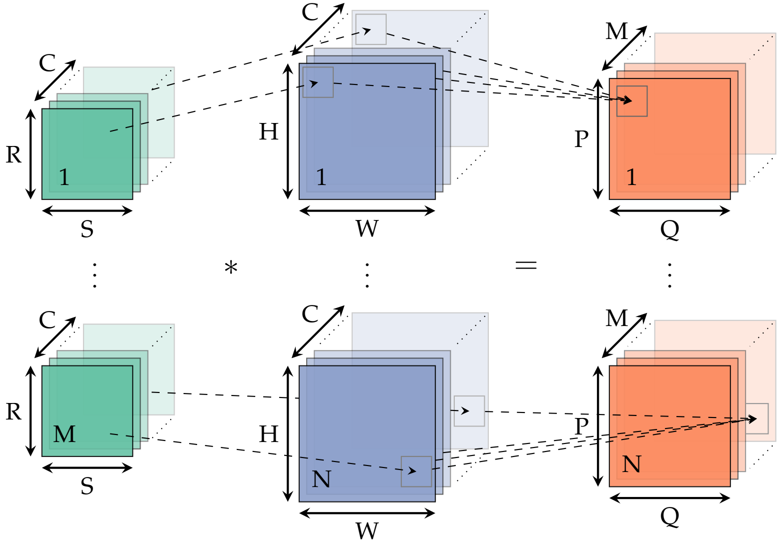


Figure 2.6: Convolution operation and nomenclature in typical CNNs using batched inputs and multiple filters, following the nomenclature propsed in [60].

From a computational perspective, CONV layers require more operations than FC layers. However, they have fewer weights and thus have a smaller memory footprint for parameters [61]. For real-world applications, we have to extend the simple CONV layer example, as depicted in Figure 2.6. In this thesis, we will use the parameter naming introduced by Sze et al. [62], which are listed in Table 2.1. As stated before, a CONV layer typically has multiple filters (M), to represent distinctive features. The spatial dimensions of a filter are assigned the shape parameters R and S. In addition, inputs have multiple channels (C), for example three channels for each color of an RGB image. Filters that are applied to an input feature map have to match the channel count. Multiple images might be processed at the same time in batches (N). Putting this together, we will use the following CONV layer notation throughout this thesis:

$$\underbrace{I^L_{fm}}_{\substack{\text{Input}\\\text{feature map}}} * \underbrace{W^L}_{\text{Weights}} + \underbrace{B^L}_{\text{Bias}} \Rightarrow \underbrace{O^{L+1}_{fm}}_{\substack{\text{Output}\\\text{feature map}}} \quad \text{with} \quad \begin{aligned} I_{fm} &\in \mathbb{R}^{N\times C\times H\times W} \\ W &\in \mathbb{R}^{M\times C\times R\times S} \\ B &\in \mathbb{R}^{M} \\ O_{fm} &\in \mathbb{R}^{N\times M\times P\times Q} \end{aligned} \tag{2.8}$$

Index	Meaning
N	Batch size of the input; Stays the same through the whole network.
M	Number of filters, determines the number of channels in the output feature map.
C	Channels, depth of the input. Filter depth has to match the channels in the input feature map.
R / S	Width and height of a filter. Also, referred to as kernel size.
P / Q	Output width and height
W / H	Input width and height
U	Stride
W_{pad} / H_{pad}	Padding for width and height
W_{dil} / H_{dil}	Dilation for width and height

Table 2.1: Parameter nomenclature for CONV layers used in this thesis.

Besides the dimensions, a CONV layer has modifiers like stride, dilation, and padding. The stride (U) of a convolution filter determines the step size of movement along the input feature map. A filter with a stride of two, for example, skips every other receptive field while sliding over the input. Hence, a stride can be used to down-sample the input or to compress information. Some networks apply dilation (H_{dil} and W_{dil}, often $H_{dil} = W_{dil}$) in a convolution operation to widen the receptive field of a kernel. For example, a dilation factor of two only takes every other input pixel into account. By increasing the receptive field, more information can be considered. However, by simply increasing the kernel size, more weights are added to the network. This can be circumvented with dilation. Padding (H_{pad} and W_{pad}, often $H_{pad} = W_{pad}$) can be applied to put the same attention to the edges of an input as to the center parts. When the filter is moving along an input, pixels at the border of an image are only passed once by A kernel, while pixels in the middle are seen more frequently by it. Padding compensates for this effect by artificially enlarging the input along all four corners. The added pixels can be filled with zeros (zero-padding) or with the same value as neighboring pixels (same-padding).

Equation 2.9 gives the computation for an output of a CONV layer. An algorithmic form can be found in Listing 2.2. The size (P and Q) of the output feature map is influenced by the size of the input, the kernel size, and the stride. The computation

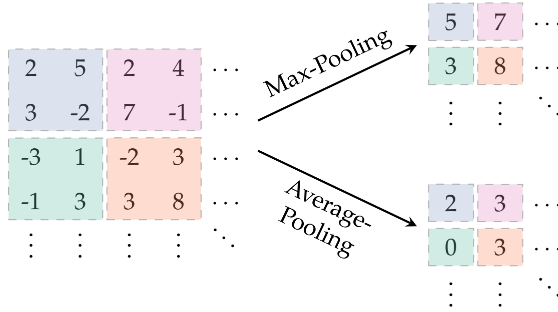


Figure 2.7: Principle of Max- and Average-Pooling. Shown with a two-dimensional input and a 2×2 pooling filter; visualization adopted from [28].

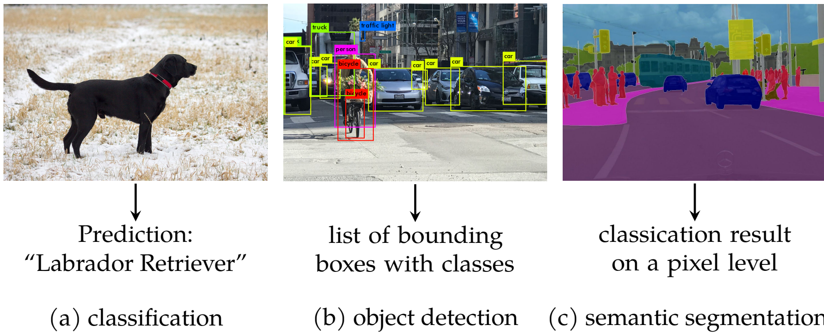
involves seven nested for-loops that iterate over all dimensions of the input feature map and the filters. As we will see later, in a hardware realization these loops can be executed in parallel and shuffled, since there is almost no data dependency. This increases data reuse and thus boosts inference performance.

$$o[n][m][p][q] = \sigma\left(\sum_{c=0}^{C-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S-1} i[n][c][Up+r][Uq+s] \times w[m][c][r][s]\right) + b[m]$$

$$P = \frac{H - R + 2 \cdot H_{pad} + U}{U} \qquad Q = \frac{W - S + 2 \cdot W_{pad} + U}{U} \tag{2.9}$$

2.2.2 Pooling Layer

Pooling layers are a key component of CNNs. They help to compress spatial information, which is especially desirable in image processing. Although CONV layers do this as well, they require a vast number of Multiply-Accumulate (MAC) operations, which is growing with the dimensions of the input image. Consequently, it is beneficial for the computational load and the training process to drop input pixels that contain only little information. Pooling works like a convolution operation using a sliding receptive field, but instead of pointwise multiplication, it simply pools all values in the field. Different kinds of pooling are available, of which the two most



(a) classification (b) object detection (c) semantic segmentation

Figure 2.8: Typical tasks of CNNs.

common are shown in Figure 2.7 using a two-dimensional example. Max-Pooling looks for the maximum activation in the receptive field and drops all others, while Average-Pooling computes the average of all activations in the receptive field. The first looks for activations that contain the most information, and the latter can be applied to smoothen the CONV output. The size of the receptive field and a stride of a pooling operation can be parametrized. Usually, stride is set to the same size as the receptive field to create non-overlapping pools. For data reduction pooling is preferred over large convolution filters with strides, since it is simple to compute and does not add any weights. Hence, pooling is found in most CNNs after a set of CONV layers, typically one to three.

2.2.3 Evaluation Metrics

Typically, three types CV tasks are addressed with CNNs: Classification, object detection and segmentation. Examples are given in Figure 2.8. The latter can be subdivided into semantic and instance segmentation. We will now look at each task and how the performance can be measured, which is essential to check whether the model training converges.

Classification (example in Figure 2.8a) is one of the most common tasks for a CNN. The output of a given network should tell which category an input belongs to. This is realized with a SoftMax activation in the last layer and One-Hot encoded labels. Classification problems are usually evaluated with a Top-1 or Top-5 accuracy score. The first metric tells how many predictions made by the CNN equal the label exactly.

Similarly, a Top-5 accuracy, gives a percentage of how many of the actual labels are among the highest five prediction results for a single classification. Top-5 accuracy can be generalized to a Top-K accuracy, to see if a the highest K predictions are amoung the label.

Object detection (Figure 2.8b) has the goal to identify the position of an object in an image. Therefore, CNNs are trained to output the coordinates along with the size and class of all objects present in an image. As an evaluation metric for object detection tasks, we can combine the classification accuracy for the predicted labels and the Intersection over Union (IoU) for the predicted position. In general, IoU gives the ratio of the intersecting area of target and prediction regarding the total area of label and prediction. E.g., if prediction and label do not overlap at all, the IoU returns 0, if they exactly match, the IoU computes to 1. With IoU, another metric called *mean AP* (*mAP*) can be computed. Only detected objects with an IoU above a defined threshold, are considered valid detections, others are false predictions. Based on the valid or true positive and true negative predictions, one can calculate Precision and Recall of the task. The area under the precision recall curve then yields the Average Precision (AP). Taking the average of each class Average Precision (AP), gives finally the mAP. This metric has the advantage that it considers each object's class, and the threshold prevents ambiguous results from showing up.

Segmentation can be divided into instance and semantic segmentation. Figure 2.8c shows an example of semantic segmentation, in which each pixel is classified individually. Instance segmentation, in contrast, aims to identify contiguous regions which belong to the same instance of a class. Hereby, a single class can be present multiple times, e.g., other cars in a driving scenario.

A very straightforward metric to evaluate segmentation results is pixelwise accuracy. For this metric, each individual pixel is compared to the label and then the ratio of correct predictions in relation to total pixels yields the pixel accuracy. However, this metric might be misleading, when one class, e.g., background is overrepresented in an image. Therefore, more often IoU or mean IoU (mIoU) is used. In contrast, to IoU in object detection, here it is computed on pixel level. Hence, correctly classified pixels for each class are summed up and compared to all falsely predicted pixels. In doing so, we get the individual IoU for a given class. By averaging all classes, we can compute the mIoU. The main advantage of this metric is that it represents the accuracy of each class evenly. Similar to objection detection, mAP is also frequently found in segmentation tasks.

2.3 Benchmark Datasets and Models

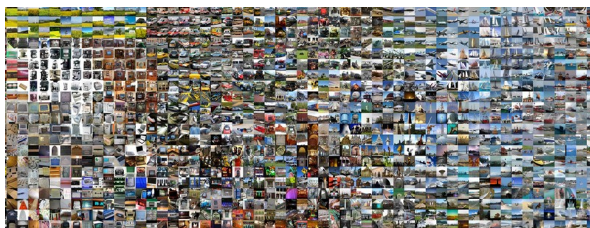
We have seen in the previous chapter that computation and training of any kind of ANN can have ample complexity. To cover some of this complexity under a hood of reusable functions, many software tools and libraries were presented. Besides the training itself, comparability to other works became more important. Hence, today a wide range of established datasets and network topologies for a large variety of applications and domains exists, from environment perception in autonomous driving over medical imaging data to networks and datasets aimed for audio recognition. This section will give an overview of popular networks and datasets, and introduces the widely used ANN framework *PyTorch*, which is used throughout this thesis.

2.3.1 PyTorch

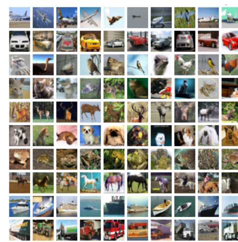
PyTorch [63] is one of the largest available library and framework for any task dealing with ML. Despite its relatively recent initial release in 2016, it quickly grew to become a widely used framework alongside another large framework, TensorFlow. The open-source library is used and supported by large companies and research facilities like Uber and Tesla. PyTorch provides a simple Python and C++ interface for handling complex workloads in a simple manner, with dedicated parts enabling straightforward acceleration using GPUs. It also features a wide range of tools to load and manipulate data, as well as a large selection of pre-trained models. Currently, in its 2.0 release, PyTorch offers various modules for training, optimization, and setup of ANNs.

2.3.2 Popular Benchmark Datasets

With ANNs becoming more popular, the need for benchmark datasets to achieve comparability of different ANNs grew. As of today, a variety of benchmark datasets are available to train and evaluate all kinds of network topologies. Modern datasets feature many million samples and are labeled with outrageous effort by human dataset creators. Here we focus on CV datasets, of which the most popular are listed in Table 2.2. It has to be noted that the table only shows the tip of the iceberg and many others are available. For example, for more narrow tasks many more datasets are available, e.g., X-Rays of patients with lung diseases [64], classification of different dog breeds [65], etc.



(a) Examples from the ImageNet dataset with over 1 000 000 samples arranged into 1000 classes.



(b) Examples from the CIFAR-10 dataset.

Figure 2.9: Example images of ImageNet and CIFAR-10, which are two of the most popular image classification datasets [1; 67].

Image classification is one of the most widespread challenges for ANNs. Hence, many datasets were published in the last few years. Most notably are MNIST [37] from 1998, ImageNet [66] from 2009 and CIRAR-10 [67] also from 2009. Sample images of the two latter datasets are shown in Figure 2.9. MNIST and CIFAR-10 are elementary datasets, with each 60 000 annotated images categorized into ten classes. ImageNet, in contrast, has more than 10 million samples grouped into 20 000 classes. About 1 400 000 samples (1 281 167 training, 50 000 validation and 100 000 test samples [1]) with 1000 classes are publicly available as *ImageNet-1K*. The dataset features some extremely hard to distinguish image classes like, e.g., 90 different dog breeds. Hence, in 2012 ImageNet-1K was released to host a challenge referred to as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Over time, the challenge boosted network performance tremendously. Starting from over 15 % error rate in 2012, it ended with an accuracy score over 97 % in 2017, outperforming the empirically determined human classification performance, which is has an error rate of about 5 % [1].

For object classification, the most popular datasets are COCO [69] by Microsoft, which stands for “common objects in context” and Pascal Visual Object Classes (VOC) [70]. It has to be noted that all object detection datasets mentioned in the table, in addition, contain labels for instance and semantic segmentation. Some extra data is available as well, e.g., COCO features data for human pose estimation and Pascal VOC data for action classification. Similar to ImageNet, the creators of both datasets also hosted a competition to push new network architectures beyond the former boundaries. The Pascal VOC challenge ended in 2012 with improvements in detection AP, starting

Dataset	Year	Samples	Description / Classes
Image classification			
MNIST [37]	1998	60 000	28x28 grayscale images of handwritten digits
CIFAR-10 [67]	2009	60 000	32x32 color images showing 10 different image classes
ImageNet [1]	2009	1 431 167	RGB images with 1000 classes
ObjectNet [68]	2019	50 000	313 classes
Object detection			
COCO [69]	2014	330 000	80 classes of bounding boxes
Pascal VOC 2011 [70]	2011	11 530	20 classes of bounding boxes
DOTA [71]	2018	11 268	18 classes of bounding boxes
Semantic segmentation			
CityScapes [72]	2016	25 000	Segmentations of driving situations from 50 cities with 30 classes
BDD-100k [73]	2020	$\sim$10m	50 000 driving scenes with 40s at 30 fps
KITTI [74]	2012	400	
DAVID [75]	2020	10 767	driving scenes generated by the car simulator CARLA

Table 2.2: Popular machine learning datasets tasks with their size and domain.

from below 0.3 in 2008 and ending at over 0.4 in 2012. COCO's challenge started in 2015 with submissions achieving 0.41 detection AP. It is still ongoing and current submissions yield 0.59[1].

While all object detection datasets also contain data for semantic segmentation, some datasets have established themselves especially for segmentation tasks. All the above-mentioned datasets are intended for autonomous driving situations. The first three datasets from the table contain recorded video sequences from actual test vehicles, while the DAVID dataset consists of simulation-generated images from CARLA [75]. CityScapes [72] is one of the most popular datasets for autonomous driving, recorded in 50 different cities spread over Germany. It features 5000 fine annotated images

[1] https://cocodataset.org/#detection-leaderboard accessed 2023-04-18

together with 20 000 coarsely labeled samples. In addition, metadata like vehicle odometry data, position information or stereo camera images are provided. The dataset has 19 semantic classes and high-resolution images with $2,048 \times 1,024$ pixels. Similarly, BDD-100k [73] from Berkeley is considered the largest available segmentation dataset with over 10 million samples, of which every tenth is annotated, recorded in multiple cities in the US. Features of BDD-100k contain annotations for lane marking and drivable area, as well as full frame segmentation. It also comes with additional metadata and features samples under different weather conditions. The creators even hosted a challenge on notable conferences. The KITTI [74] dataset was released in 2012 and received a major update in 2015 [76]. It features driving sequences recorded in the mid-sized city of Karlsruhe in Germany. Besides the raw segmented image data, it contains much other data relevant for autonomous driving like LIDAR, RADAR, stereo camera, and metadata information. Hence, it is until today one of the most influential datasets for the autonomous driving community.

2.3.3 Benchmark Convolutional Neural Networks

Together with datasets, many neural network topologies, especially CNN models, have been presented by numerous research groups. Most of the network architectures were propelled by the competitive environment created through the challenges called by dataset creators. An overview of very impactful works that showed breaking results is given in Table 2.3. Needless to say, that this selection represents just a tiny fraction of networks that were proposed in the last ten to twenty years. However, each listed network set a new milestone through achieving higher accuracy, reducing the number of parameters or by lowering computational complexity.

LeNet-5 [29] presented in 1989 is considered the first sophisticated CNN. Featuring seven layers, of which two are CONVs layers, it is deployed for recognition of handwritten digits based on the MNIST dataset. Due to is small footprint size and thus fast training and inference speed, LeNet-5 is commonly used for experimentation by beginners. However, LeNet-5 shows only limited benchmarking capability, as more complex image classification tasks demand for larger model topologies.

In the early 2010s, rising GPU performance and novel software tools, kicked off the development of more complex CNNs. AlexNet [39] introduced by Krizhevsky et al. in 2012, was the first network to show great prediction results on ImageNet and won the ILSVRC in the same year with a Top-5 test error rate of 15.3 %. Five CONV layers

Model	**Year**	**Parameters** millions	**Computations** GFLOP
Image classification			
LeNet† [29]	1989	0.06	0.42
AlexNet* [39]	2012	61.10	0.71
VGG-16* [77]	2014	138.36	15.48
GoogLeNet* [3]	2015	13.00	1.50
ResNet-50* [40]	2016	25.56	4.09
SqueezeNet* [41]	2016	1.25	0.82
DenseNet-169* [78]	2018	14.15	3.36
MobileNet-v2* [79]	2017	3.50	0.30
ShuffleNet-v2-x?* [80]	2018	3.50	0.30
EfficientNet-v2-m* [42]	2018	54.14	5.36
Object detection			
SSD300-VGG16§ [81]	2016	35.64	34.88
FasterRCNN-ResNet50§ [82]	2016	41.76	177.64
RetinaNet-ResNet50§ [83]	2018	34.01	206.07
YOLO‡ [43]	2015	271.72	20.30
YOLOv5§ [84]	2020	21.17	18.36
YOLOv7§ [85]	2022	36.91	39.26
Image segmentation			
YOLACT§ [45]	2019	50.16	81.57
YOLOP§ [86]	2022	7.64	11.56
U-Net§¶ [47]	2015	7.76	64.15
DeepLabV3-ResNet50§ [48]	2017	39.64	191.69

†input size $1 \times 32 \times 32$, batch size 1; *input size $3 \times 224 \times 224$, batch size 1; ‡input size $3 \times 448 \times 448$, batch size 1; §input size $3 \times 640 \times 480$, batch size 1; ¶U-Net is available with a wide range of different backbones. As reference, we used the implementation from [87].

Table 2.3: Overview of popular CNN models with the year of presentation, the number of parameters and their computational complexity. The numbers are derived using the `torchinfo` package and the corresponding PyTorch implementation.

and grouped convolutions allow AlexNet to be mapped on multiple GPUs for faster training. However, they concluded in their paper that for better model accuracy, they have to wait for faster GPUs.

Shortly afterward, in 2015, VGGNet [77] by Simonyan and Zisserman was presented. The model has more layers (11-19) and proved that deeper CNNs have better prediction accuracy. VGG models are available in different configurations, with VGG-16 being the most popular. This heavyweight network has about 132 million parameters and took several weeks of training using at that time state-of-the-art GPUs. Their result show also that adding more CONV layers increases network complexity quickly, leading to exploding training durations, but also to unacceptable inference times. In addition, deeper networks tend to show overfitting.

The creators of GoogLeNet [3] tried to mitigate this issue by adding *Inception* blocks into their model. They feature convolution operations with different kernel sizes running in parallel, which widens the network instead of solely deepening it. Each kernel size learns unique features of the input feature map, allowing for a higher prediction accuracy. In total, GoogLeNet has a depth of 22 CONV layers. To limit feature map sizes and parameter count, the authors add a 1×1 CONV and a pooling layer before each 3×3 convolution operation, which in turn reduces the computational complexity. GoogLeNet won the ILSVRC in 2014 with a Top-5 Error of 6.67 %. However, the depth of the network made another challenge apparent during training: vanishing gradient, i.e., minimal errors during backpropagation lead to much slower training convergence.

In 2016, the authors of ResNet [40] addressed vanishing gradings using skip connections in between a group of CONV layers. Those groups with a skip connection are referred to as *residual block*. Depending on the configuration of CONV layers, they are either specified as Bottleneck or Basic blocks. Multiple of these blocks are coupled to achieve a network that is both deep and converges fast during training. A complete setup of ResNet is depicted in Figure 2.10. It consists of a head or stem (a single 7×7 CONV layer with MaxPooling) and tail (Dense layer with SoftMax activation for classification), as well as four ResNet stages. Each features a configuration dependent number of residual blocks. In each stage, the first residual block has a downsampling feature to compress the input feature map, which is realized by inserting a 1×1 convolution into the skip connection branch, as shown in the figure. The research paper introducing ResNet, proposes five different configurations that determine the number and types of residual blocks in each stage. ResNet-50, for instance, has 3, 4, 6

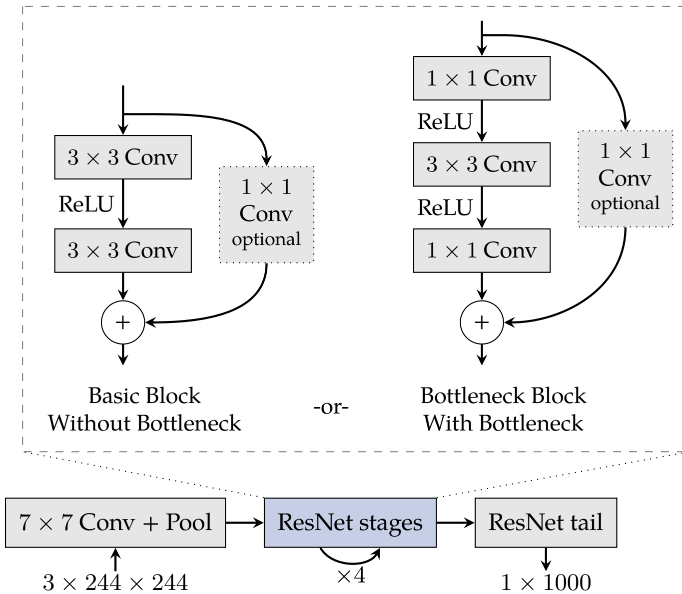


Figure 2.10: Overview of ResNet's general structure along with detailed cutouts for the two types of residual blocks, according to [40].

and 4 residual blocks in each stage: In total, 50 CONV layers in all stages. In total, ResNet-50 has 52 CONV and one dense layer counting head and tail as well. According to the results presented by the authors, ResNet-34, ResNet-50 and ResNet-101 archive Top-1 classification error rates in ImageNet of 21.53 %, 20.74 % and 19.87 %, respectively. As a result, it won the ILSVRC in 2015. Today, many variants of ResNets exist, e.g., FCN-ResNet [88], in which the tail is replaced by CONV layers to obtain segmentation results.

Inspired by the design of GoogLeNet and ResNet, SqueezeNet [41] was presented in 2016. The authors had the goal to create a model that has much fewer parameters compared to the comparability large models that were used until then. SqueezeNet achieves the claim of fewer parameters through Fire modules, as shown in Figure 2.11. Similar to Inception modules, they offer two paths for data to pass; results are then concatenated. This scheme is called expand and squeeze. SqueezeNet achieves a Top-5 classification accuracy of 80.3 %, which about 500 times fewer parameters compared to AlexNet. With compression techniques, its size can be reduced even further.

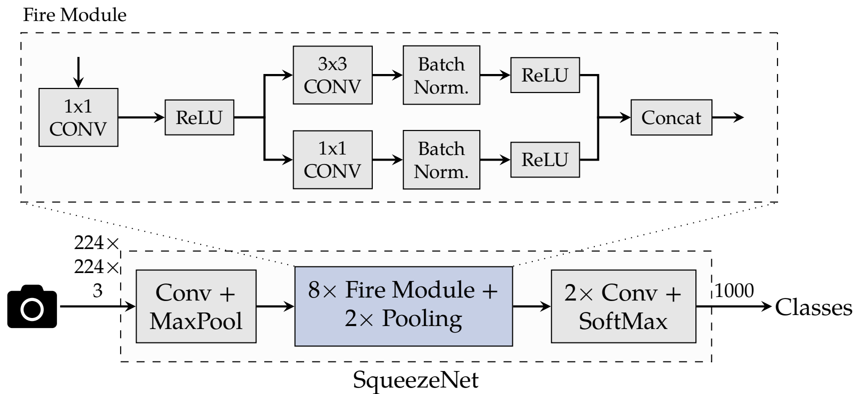


Figure 2.11: Overview of the SqueezeNet [41] architecture, with a detailed zoom into the fire modules, which perform the *expand and squeeze* operations of the network.

A summary of the above-mentioned classification CNNs and their computational requirements is given in Table 2.4. One notable aspect, if we look at the development over the last few years, is a clear shift from MLP layer to CONV layers. This aspect is visible in both the number of weights and required MAC operations. For example, VGG-16 requires 8.4× more computations to process its three MLP layers, compared to all CONV layers together. In contrast, ResNet-50 reduced the number of MLP layers to one, and now spends less computational budget on MLP layers. In absolute numbers, ResNet-50 achieves a better prediction accuracy, although it requires fewer operations in total compared to VGG-16.

Besides the more extensively explained topologies, many other classification architectures exist. Some are still worth mentioning here: DenseNet [78] from 2018 achieves prediction accuracy similar to ResNet, but with fewer Floating-Point Operations (FLOPs). MobileNet [79] is a network especially designed for highly energy constrained mobile applications. Therefore, the network uses so-called Depthwise Separable Convolution. While common CONV layers, as introduced in Subsection 2.2.1, combine the results along the input dimensions in a single step, those convolutions split the filtering and combination of dimensions into two steps, resulting in fewer computations. MobileNet achieves a higher accuracy than GoogLeNet, but with about 2.5× less computation time. ShuffleNet [89] introduced in 2018, has an optimized network architecture aiming for faster inference rather than bare reduction in number

of operations. It factors the actual computation on a computation device, since it considers, e.g., memory accesses or data movement. The successor, ShuffleNet-v2 [80] achieves a 22.8 % error rate on ImageNet-1K with as few as 2.3 G FLOPs. It features 164 layers and achieves 60 FPS on an ARM Central Processing Unit (CPU).

	AlexNet	**VGG-16**	**GoogLeNet**	**ResNet-50**
Source	[39]	[77]	[3]	[40]
Top-5 error [%]	16.4	7.4	6.7	5.3
Input size	227 × 227	224 × 224	224 × 224	224 × 224
Year	2012	2014	2015	2016
# CONV layers	5	13	57	53
Filter sizes	3,5,11	3	1,3,5,7	1,3,7
Number of channels	3–256	3–512	3–832	3–2,048
Number of filter	96–384	64–512	16–384	64–2,048
CONV weights	2.3 M	14.7 M	6.0M	23.5 M
CONV MACs	666 M	15.3 G	1.43 G	3.86 G
# MLP layers	3	3	1	1
MLP weights	58.6 M	124 M	1 M	2 M
MLP MACs	58.6 M	124 M	1 M	2 M
CONV:MLP MACs	1:25.5	1:8.4	6:1	11.7:1
CONV:MLP weights	11.4:1	123.4:1	1430:1	1930:1

Table 2.4: Selection of popular CNNs for image classification. The trend clearly shows that CONV layers became more popular over time.

With the increased availability of more powerful computation platforms, more complex CV tasks like object localization and image segmentation became conceivable. Until today, a myriad of new architectures was added to the model zoo. First steps in the direction of CNNs for object detection, were made with region proposal networks like Region-CNN (R-CNN) [90]. This network type first extracts regions of interest, which can differ in size and dimension. Then a second step tries to discard falsely identified regions and classifies the remaining. Although methods like Fast-R-CNN or Faster-R-CNN [82] were presented shortly afterward, today new and even faster methods are available. Feature Pyramid Networks (FPNs) [91] are today a commonly used method for object detection. Inside the network, features are extracted

at different scales at the same time, similar to a pyramid. This allows an FPN to recognize objects at various sizes much faster than networks that propose different bounding boxes. Hence, they achieve about 20 % improvement in AP and a 3 times speedup compared to region proposal networks. To improve the performance further, RetinaNet [83] uses focal loss, obtaining another 10 % AP increase. Similarly, single shot detectors, like SSD [81], outpaced the performance of region proposal networks quickly. Those networks conceptually follow a similar procedure as region proposal networks. However, the main distinction is that an input image is first split into smaller parts, which are classified. In the meanwhile, bounding boxes are generated. The corresponding class label for a given bounding box is finally picked based on the IoU of the bounding box and the classified input image portions. This also allows discarding falsely found bounding boxes.

One of the most popular single shot detection networks is YOLO (You only look once) [43]. Although Faster-RCNN outperformed the first version of YOLO with about 10 points more mAP on the Pascal VOC 2007 dataset, YOLO is one of the first real-time detectors. It runs at 45 FPS on a GPU, 3× faster compared to Faster-RCNN. Over the years, YOLO went through multiple evolution steps, increasing the performance and accuracy [92; 93]. In its seventh version from 2022, YOLOv7 [85] achieves, e.g., a 120 % speedup compared to YOLOv5 and 2 points more AP on the COCO dataset.

The high performance of YOLO was in the meanwhile transferred to segmentation tasks. For example, YOLACT [45] replaces the YOLO head with a multi-head that returns bounding boxes, classes, and segmentations at the same time. As backbone YOLACT deploys a ResNet-50 as FPN. Even though YOLACT yields three metrics simultaneously, it outpaces YOLOv3 in bounding box prediction accuracy and speed. Another very recent variant of YOLO for autonomous driving tasks is YOLOP [86] from 2022. It also uses a multi-head approach to detect lane markings, road objects and drivable area in a single forward pass.

Another early candidate for image segmentation is U-Net [47] from 2015. It is built as lightweight segmentation CNN for medical application, however, it can also be modified to work in other domains, like road recognition [94]. Figure 2.12 gives the basic architecture of U-Net. The network consists of a contracting path, which follows the common structure of CNNs with CONV and MaxPooling layers, and an expanding path, which uses up-convolutions to enrich and expand input feature maps. In contrast to fully convolutional networks like FCN-ResNet [88], U-Net merges data

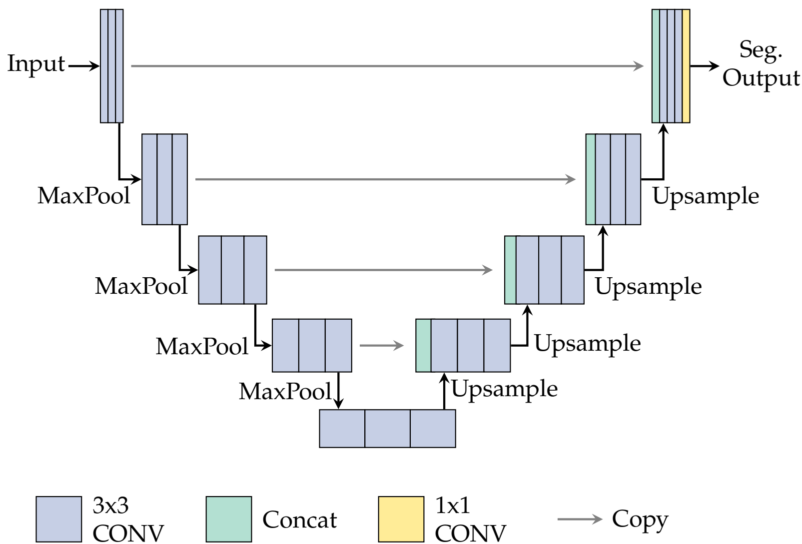


Figure 2.12: Structure of U-Nets [47]. Input feature maps are compressed through MaxPooling layers and then expanded with upsampling layers. High performance of U-Net is achieved by concatenating upsampled and compressed inputs.

from the contraction path into the up-convolution operations in the expansion path. This allows U-Net to segment features at all complexity levels and saves memory for large feature maps during up-convolution.

In the area of semantic image segmentation, the DeepLab series also made great contributions. With their first version presented in 2017 [48], they introduced atrous convolution operations that enable an enlargement of the field-of-view for a single up-convolution operation. Using a ResNet-101 backbone, the authors achieve 70.4 mIoU and 79.7 mIoU on CityScapes and Pascal VOC 2012, respectively. In the following years, improvements to the network were presented with DeepLab-v2 [95] and DeepLab-v3 [96]. The latter achieves 81.3 mIoU on CityScapes.

2.4 Hardware Architecture Development

The presentation of the first silicon transistors around 1950 marked the beginning of a journey towards integrated digital circuits, which gradually replaced vacuum tubes [97]. Researchers and engineers quickly invented circuits entirely made of transistors and discrete electronic components that solve mathematical problems. One key technology that enabled this progress is the Complementary metal-oxide-semiconductor (CMOS) fabrication process, which allows for low-power consumption and has a high immunity to noise. Subsequently, simple circuits evolved into more sophisticated architectures of silicon circuits, capable of performing arbitrary mathematical tasks many times faster than any human can. The integration of transistors into chips continued at an unprecedented pace, with millions of transistors fitting into a single integrated circuit today. This trend is shown in Figure 2.13. The depicted progress follows *Moore's Law*, postulated by the founder of Intel in 1965, which states that each year the number of transistors that fit into the same chip area doubles. Despite some contradictory opinions, it holds true for most cases until today. This tremendous increase in transistor density was made possible through immense efforts in chip manufacturing and handling, which enabled continuous shrinkage of a transistor's size. From a 10 µm technology node in one of the first integrated circuits around 1970, the size of a single transistor gradually decreased to way below 10 nm. As of 2023, modern state-of-the-art System-on-Chips (SoCs) like Apple's M3 Max chip accommodate over 92 billion transistors [98], each with a mere size of 3 nm, 10 000-times smaller than a human hair.

Besides the bare number of transistors that fit in a single silicon chip, smaller integration of transistors also allowed for higher clock speeds. This directly results in higher operational throughput of processors, as single instructions simply take less time to compute. But there is a flip side to the ever-smaller integration: with higher clock speeds also the dissipated power rises that is emitted while transistors switch. We can see the trend of rising power (shown in orange) and clock frequency (blue) in Figure 2.13. However, if one looks at the power dissipation of a single transistor, we observe that along with transistor shrinkage, the power goes down. This is due to lower voltages and currents that can be used for smaller silicon circuits. Hence, for a constant area the power density stays the same, which is referred to as *Dennard Scaling* [99]. However, dissipated power from ever faster switching transistors eventually became a problem, which led to the breakdown of Dennard Scaling around 2005, which is also clearly visible in the figure above. As one can see, the amount of

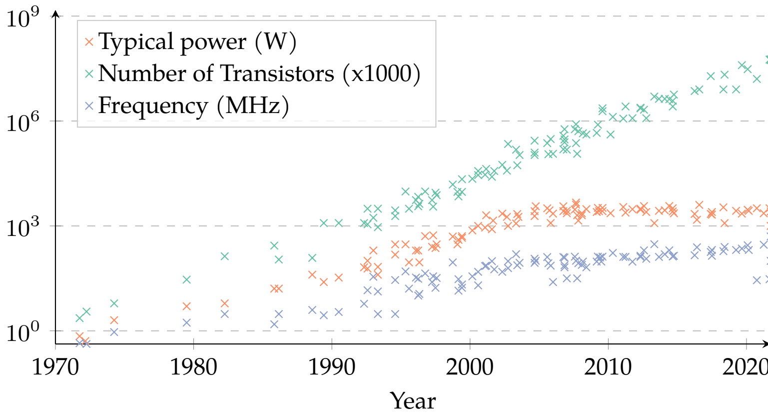


Figure 2.13: Evolution of the number of transistors, the maximum frequency, and the typical power consumption in microprocessors from 1970 until now. Data source: [100].

power hit a boundary at the same time as the clock speed gains did. At that time, dissipated power from the circuit overshoots the capabilities of chip cooling, causing a thermal runaway and possibly damage to the chip.

Since then, an interplay of various techniques allowed us to move forward towards today's highly sophisticated chips. First—at least for now—unrestrained shrinkage of transistors allows hosting an increasing number of transistors on the same chip area. Hence, parallelization of tasks became much more favorable than waiting for faster silicon. In addition, modern Hardware/Software Co-Design approaches allow developing hardware circuits together with the application they later compute, leading to Application-Specific Integrated Circuits (ASICs) tailored to solve a particular task very efficiently. Moreover, dedicated hardware accelerators for a specific task are added to integrated circuits, rendering complex but efficient heterogeneous systems that often integrate many CPUs, Field-Programmable Gate Arrays (FPGAs), interface controllers, and so forth into a single SoC. Over time, the process of designing and creating well-functioning circuits is increasingly aided by modern Hardware Design Languages (HDLs) to make complex designs more manageable. In addition, subsequent processes in chip design were automated to accelerate deployment even further. In the following, we will give a brief overview of the tools, methods, and technologies used in state-of-the-art chip design.

2.4.1 Design Languages, Automation, and Tools

Highly sophisticated HDLs and design automation tools, as well as tools in general, are essential for the design of modern integrated circuits. Tools not just support the architecture design itself, but also the later integration and verification process. Both are essential to ensure correct and reliable behavior for the chip.

In the first days of chip design, circuits were created by placing individual transistors manually. While this is feasible for simple processor designs, rising demands and hence growing design complexity rendered manual design soon inefficient and uneconomical. As a consequence, in the 1970, the development of HDLs stated. HDLs work similar to programming languages and describe the logic and behavior of a circuit in a formal way. Their development gained pace when new tools for automated circuit synthesis were presented. These tools transform the behavioral description of a circuit into a netlist consisting of a list of gates and their connection among each other. From the many introduced languages, today the largest two could establish themselves: VHSIC Hardware Description Language (VHDL) [101] and Verilog [102]. Both were introduced in the 1980s, are standardized by the IEEE and went through multiple versions until today. Because both languages are relatively old, also new concepts were proposed, like Chisel [103], a Scala dialect introduced in 2012.

Formal description of digital circuits with HDL, also enabled tools to parse this input. Today, a wide range of them is available for hardware designers. Starting by simulation tools that support the behavior verification of circuits before they are hardened in silicon. Circuit simulators use testbenches, which define input stimuli and expected outputs for the circuit under test, allowing for automated testing. Synthesis tools are another crucial tool. They transform HDLs input into netlists. To do so, today's synthesis tools employ highly sophisticated algorithms to optimize the user input towards an efficient netlist. Modern synthesis tools also take aspects like fan-out, i.e., the number of endpoints a gate output has, into account to achieve better power efficiency. With implementation tools, netlists can be mapped onto silicon devices. One straightforward method is to make use of standard cells. For a given technology node, typically a library of standard cell is provided by the fab, which contains premade designs for common gates. Those pre-made designs are like building blocks that can be placed onto the chip area. Power supply inputs in standard cells are located at the same end of a cell to ensure less complex wiring. Regardless of whether standard cells are used, implementation tools nowadays take care of complex tasks like placing the cells or transistors and routing the connections among them. Finally,

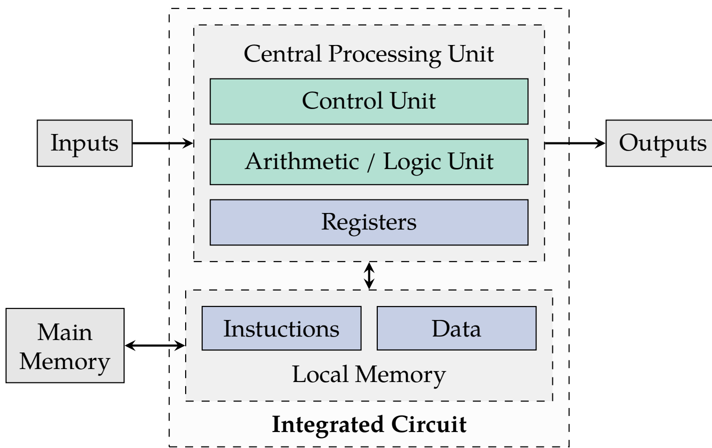


Figure 2.14: Simplified building blocks of a CPU, visualization taken from [104].

tools for verification of an implemented design or tools to simulate power dissipation are crucial to test and evaluate circuit designs before the costly chip manufacturing process is stated. Summarizing, today's increasingly complex designs are only made possible through serious progress in tools supporting the designers.

2.4.2 CPUs, GPUs and FPGAs

Some of the most notable achievements enabled through the described advances in silicon technology and design methodologies are highly sophisticated processors and Intellectual Property (IP) macros. Today, processors have inevitably become indispensable for many applications and are hence considered the *brain of a computer*. That's also why they are referred to as CPUs. General-Purpose Processors (GPPs), a kind of CPU, can solve almost any workload that can be described as a series of commands or instructions. This series is then transformed into a program using a compiler. The encoding of individual instructions is hardwired into the CPU's logic and is referred to as an instruction set. Figure 2.14 shows the basic components of a simple CPU. A local memory stores a sequence of instructions and data the processor is currently working on. Dedicated memory controllers with Address Generation Units (AGUs) and Direct Memory Accesses (DMAs) allow accessing large off-chip memories. Inside the CPU, arithmetic computation happens inside an Arithmetic

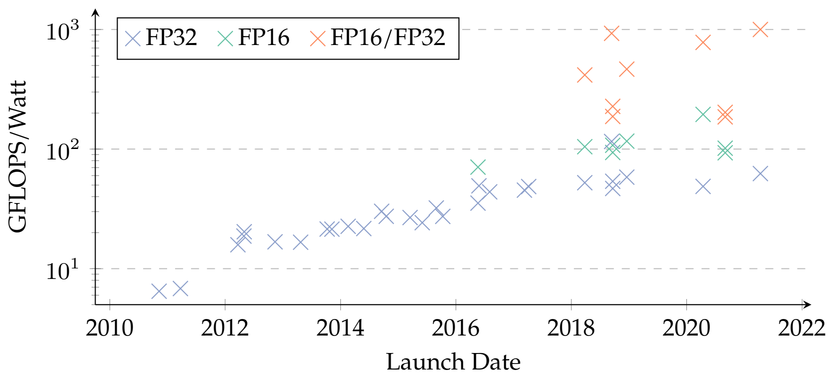


Figure 2.15: Performance of recent GPUs from 2010 to 2022. Data source: [4].

/ Logic Unit (ALU), which executes simple logical and mathematical operations, like additions or bitwise and. A set of registers stores intermediate results of the ALU for fast access. Moreover, dedicated registers keep track of the program status, e.g., a program counter. The ALU itself is driven by a control unit. It moreover handles data movement, i.e., loading the right data and instruction from the main memory. Another key characteristic of a CPU is that it can respond to conditions. Those conditional program paths, are also taken care of by the control unit. If, for example, a conditional expression evaluates positive, it triggers a jump to a specified location inside the program by updating the program counter register.

While the description elaborated above paints only a basic picture, it holds true for today's much more intricate and advanced CPUs. In novel CPUs memories got replaced by multilevel caches that employ smart methods to keep data that is likely to be used soon already close to the ALU. Modern control units, use very sophisticated techniques to predict conditional code execution and to preload the corresponding sequence of instructions beforehand. Techniques like Pipelineing and Superscalarity [105] allows ALUs to perform individual steps of a single instruction in an interleaved manner to increase clock speed and throughput. Moreover, ALUs started to implement vector instructions to operate on multiple data using the same instruction and leverage parallelism in applications [106]. This is referred as Single Instruction Multiple Data (SIMD) according to Flynn's taxonomy that clusters computer architectures [107]. Finally, Floating Point Units (FPUs) in CPUs enable hardware accelerated processing of floating-point arithmetic.

With the rise of personal computers and more dedicated graphic interfaces, Graphics Processing Units (GPUs) were introduced. They feature large memories to buffer output images and units to process specialized instructions for graphics in parallel. GPUs are coprocessors to CPUs, as they receive their instructions from the CPU. Modern GPUs have multiple units realized in hardware to support polygon computation, texture mapping and video rendering. Besides that, GPUs were opened also for general purpose workloads that can benefit from massively parallel computation. This General-purpose computing on graphics processing units (GPGPU) approach, e.g., boosts the performance of tasks like image processing [108] or databases [109]. The openness to general purpose workloads makes GPUs also a key component in efficient DNN computation. Since then, better accessibility to GPUs was addressed by the manufacturers with libraries like CUDA or cuDNN by NVIDIA or ROCm[2] by AMD. Today, GPUs are readily programmable and deliver tremendous performance with great efficiency, as shown by the recent advances in performance (Figure 2.15).

Another notable piece of hardware are Field-Programmable Gate Arrays (FPGAs) [110]. FPGAs are themselves fixed hardware blocks, however, they allow hosting other generic hardware designs on top of them. In contrast to ASICs, they are not hardened once, but the hardware circuit can be changed even during runtime. This is made possible through their reconfigurable and flexible structure, of which a simplified version is depicted in Figure 2.16. An FPGA leverages a characteristic of arbitrary digital circuits, which is that it can always be represented by a set of Look Up Tables (LUTs), flip-flops, and multiplexers. All these basic building blocks are found in FPGAs. They are pooled together as Logic Block (LB), which are connected with very flexible routing infrastructure. This allows FPGAs to map arbitrary hardware circuits on them. To actually map a given digital circuit onto an FPGA, sophisticated tools emerged that translate a given netlist into a LUT and flip-flop representation. Besides that, modern FPGAs adopted to the requirements of faster and more versatile possibilities for circuit design by adding dedicated Digital Signal Processors (DSPs) and local memories. All of those additional building blocks are attached to the routing infrastructure as well. Today, FPGAs are used in many domains, from behavioral circuit testing to accelerators for highly parallel workloads. Modern SoCs even feature embedded Field Programmable Gate Arrays (eFPGAs) enabling extendability and flexible acceleration.

[2] https://github.com/RadeonOpenCompute/ROCm

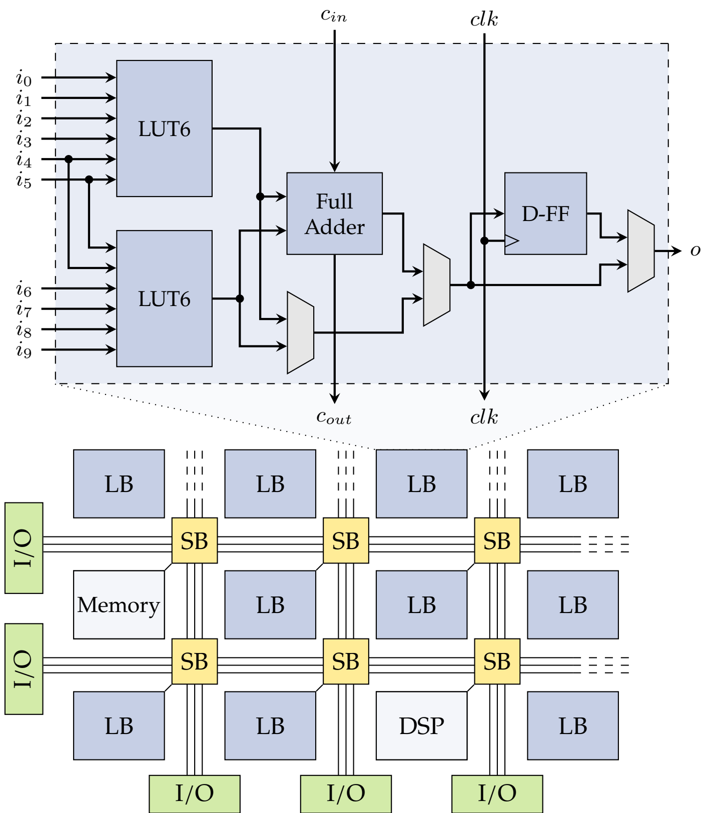


Figure 2.16: Overview of an FPGA with a detailed cutout of a LB, which allows an FPGA to represent any arbitrary digital circuit, according to [104].

2.4.3 Hardware Acceleration and Co-Design

Over decades, programmers and application engineers could rely on mere advances in silicon technology to yield more performance and better energy efficiency of integrated circuits. But gradually this trend came to a halt, as demand for more computational capabilities drifted apart from the progress silicon made (refer also to Figure 2.13). One technique to circumvent this is integration of dedicated accelerators into SoCs or systems in general [111]. A notable example are the aforementioned GPUs, which are added to offload graphics computation.

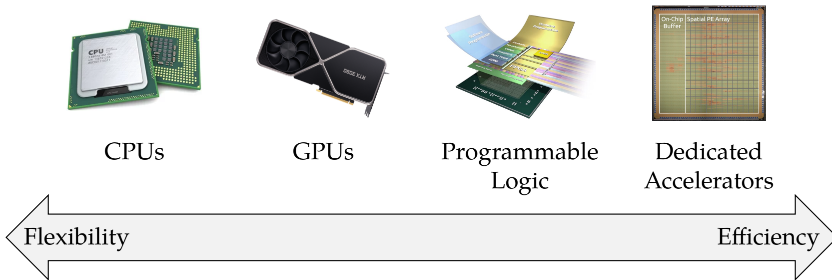


Figure 2.17: Overview of different computation devices, according to [112]. From CPUs that offer high flexibility for generic and versatile workloads to ASICs, which are tailored for a particular workload and hence deliver an impressive performance per invested silicon area and energy.

Figure 2.17 gives an overview of various processing units, which can be categorized based on their flexibility and efficiency. While dedicated accelerators deliver a superior performance per silicon area and power, they forfeit flexibility for changing workloads and applications. In addition, it can be challenging to design them efficiently. On the other end, General-Purpose Processors (GPPs), like CPUs, severe a wide range of applications, but suffer performance in some applications.

As a consequence, modern integrated circuits feature a selection of different processing units. However, laying out a well-balanced overall system design with the right accelerators is not an easy task. Just adding accelerators for every task will likely overrun energy and area budgets. Hence, depending on the application, it has to be understood whether it is worth spending the development effort and chip resources to add an accelerator. Therefore, it is important to thoroughly understand the kind of computation that is later executed on the platform. This is typically done by in-depth tracing of the data and control flow. As a result, not all parts of a computation may benefit from a dedicated hardware accelerator. For example, many branches in the program bloat the hardware accelerator dimensions, or inefficient memory access pattern may lead to underutilization of the accelerator. Moreover, applications with much data movement might not necessarily benefit, as it consumes too much time and cancels the acceleration gains. In contrast, applications that can benefit the most, have homogenous data movement, only little branching and offer high potential for parallel execution. Hence, DNN workloads are a perfect candidate for large-scale hardware acceleration, which will be covered in-depth in the following section.

2.5 Hardware Accelerators for DNNs

As we have seen so far, DNNs became extremely popular for a wide range of applications, since they show great performance in accomplishing a myriad of tasks. However, the computational complexity of DNNs, CNNs and especially large language models is huge [113]. To put this into perspective, a single inference of a $3 \times 224 \times 224$ image using ResNet-50 [40] requires already 4.09 G FLOPS. In addition to the computational complexity, memory requirements for current DNNs are vast, e.g., a ResNet-50 inference requires 177.83 MB memory to store all intermediate results of a single image considering 32-bit floating-point precision. While one can argue that modern computation platforms are well capable of handling those workloads, the major challenge lies in the requirements some applications impose on inference speed, energy efficiency and reliability. These aspects highlight what computation systems face when executing inference of DNNs, not to mention to immense computational complexity of the DNN training process. In this section, we will introduce first how DNNs are computed in general, before diving deeper into dedicated platforms, which can process DNNs efficiently.

2.5.1 Evaluation Metrics and Theoretical Background

Looking at the sheer number of computations required for a ResNet-50 inference, it seems hard to design efficient hardware for such workloads. But, before looking at actual hardware platforms that can tackle DNN computation, we have to untangle what exactly efficient refers to. To make a discussion on efficiency objective, we can look at the following five different metrics, which are usually used to evaluate DNNs.

Throughput describes the raw number of operations that can be processed within a given time span. Typically, throughput is denoted as operations per second or in the case of floating-point operations as FLOPS per second.

Energy consumption is the energy required, e.g., for an individual operation or inference. It is frequently composed of static and dynamic energy, which depends on the architecture and the workload, respectively. However, it is often reported as energy demand per inference. In correlation with the throughput, one can compute the power, which makes it easy to compare different hardware platforms by looking at the Operations per Watt.

```
for x in range(H - R):
  for y in range(W - S):
    # get receptive field for
        all channels
    patch = I[:, x:x+R, y:y+S]
    # then transform into a row
    out[x*(W-S)+y,:] = \
        flatten(patch)
```

(a) im2col

```
for m in range(M):
  # Transform single row back
      into 3-D tensor
  out[m, :, :] = \
      reshape(inp[:,m], (C,R,S))
```

(b) col2im

Listing 2.1: Python-like pseudocode implementation of im2col and col2im.

Latency describes the time it takes until the computation of an operation or inference has finished. It becomes especially important in control tasks like autonomous driving when a computation shall have real-time behavior.

Scalability, Flexibility and Reconfigurability are essential metrics when a great variation of DNN task will be computed. For example, training poses different requirements on the hardware than a pure inference application.

Accuracy is not really a hardware metric itself, but rather refers to how good the algorithm performs on a given task. However, we will later discuss how the model's accuracy can play an essential role when it comes to more efficient hardware accelerators, as some methods to increase throughput or energy efficiency have an accuracy impact.

In general, there is no clear answer to how important a given metric is, but it depends on the application where the DNN accelerator will be deployed. For example, datacenter applications may prioritize raw throughput over energy consumption, while mobile environments like smartphones require much more focus on energy efficiency.

From a computational standpoint, we have already discussed in Section 2.1 that the basic operations of an ANN are multiplication and addition operations with corresponding weights and inputs as operands. For MLPs, they can be written as vector-matrix multiplication, of which the vector represents the inputs $(1 \times n_{inputs})$ and the weights are denoted as matrix $(n_{inputs} \times n_{outputs})$. If we consider multiple inputs computed at the same time, this batch of inputs can also be written as matrix $(N \times n_{inputs})$, which turns the computation of one layer into a matrix-matrix multiplication. Multiple layers then create a chain of matrix multiplications.

While the connection scheme between neurons in MLPs makes it easy to use simple matrix multiplications to compute them, the connection scheme of CNNs looks much different. CNNs, in contrast, have fewer weights connecting inputs to outputs in a sparse manner and individual weight filters are slid along the three-dimensional input feature map. Processing them happens in a seven-fold loop. However, we can apply a smart transformation, called *im2col* ("image to columns"), to break down the seven-fold loop in a CONV layer [38]. A pseudocode implementation is shown in Listing 2.1a. Therefore, four-dimensional input feature maps and parameters of CONV layers are reshaped into two-dimensional matrices, which can be computed as straightforward matrix-matrix multiplication. This requires the same number of operations like computing the seven-fold loop. After matrix multiplication, *col2im* ("columns to image") transforms an output feature map back into the four-dimensional form (see Listing 2.1b). However, it can also stay two-dimensional for the next convolution operation. A simple example of how operands and outputs look like after they underwent im2col transformation is given in Figure 2.18. Colors and parameter identifiers follow Figure 2.6. Four-dimensional filters and kernels are broken down into rows, each representing an entire convolution filter. Similarly, input feature maps are transformed into columns. It has to be noted that the figure only shows a three-dimensional input, however as indicated by the identifiers, we can compute entire input batches (N) by stacking them together.

In summary, all typical layers of DNNs can be computed using matrix multiply operations. This is very advantageous, since matrix multiplications are very straightforward to compute and a wide range of implementations for various platforms are already available.

2.5.2 DNN Computation, Memory Hierarchy and Data Orchestration

Now that all DNN layers are present in matrix form, we can discuss how they are computed and where the challenges are. If we look at how a single result of a matrix-matrix multiplication is computed ($o_{ij} = a_{i0} \times a_{0j} + \ldots + a_{in} \times b_{nj}$), we can see that at center are only multiplication and addition operations, which can be fused into MAC operations. Each MAC operation thereby takes an input, a weight, and the previous partial sum as inputs. Then it computes the next partial sum, as shown in Figure 2.19.

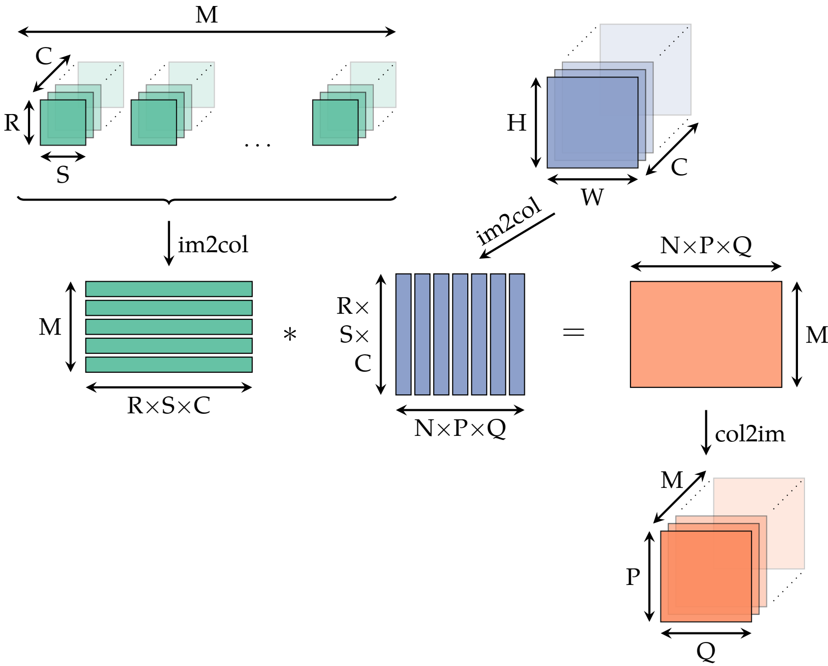


Figure 2.18: Principle of im2col and col2im operations to transform four dimensional inputs for CONV layers into two dimensions in order to compute them using generic matrix multiplication; visualization adopted from [38].

As we only need two kinds of operations, the complexity in computing DNN workloads does not originate from the individual operations, but rather in the immense number of computations that have to the processed. Looking at the trend of modern CNNs (see Table 2.4), the network architecture is moving towards more CONV layers and less FC layers. Image segmentation networks even omit all FC layers. At the same time, CONV layers require by orders of magnitude more computations than FC layers. However, they also have fewer weights. For example, the three FC layers of AlexNet account for 58.6 million weights and MAC operations. In contrast, AlexNet's CONV layers only require 2.3 million weights, but 666 million MAC operations. With deeper CNNs this trend intensifies, ResNet-50 requires 3.86 billion MAC operations to process its 52 CONV layers, but only 2 million for the FC layers, a ratio of 1930:1.

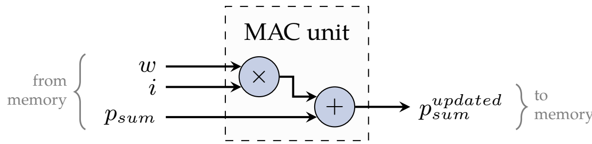


Figure 2.19: Inputs and output of a MAC unit when computing partial results for DNN inference, according to [21].

The enormous number of computations in CONV layers originates from multidimensional filters that are slid over an input feature map. In general, given two matrices with dimensions $L \times M$ and $M \times N$ the computational complexity results to $\mathcal{O}(L \cdot M \cdot N)$. Consequently, considering matrices that grow in each dimension cause a cubical increase in MAC operations. Looking again at ResNet-50 for reference and example, we can pick one of the computationally most complex layers: the second convolution in the first bottleneck block. It has the following shapes: a $1 \times 64 \times 56 \times 56$ input feature map, considering batch size 1, $64 \times 64 \times 3 \times 3$ weights and with padding of 1 consequently a $1 \times 64 \times 56 \times 56$ output feature map. If we apply im2col to transform this problem shape, we end up with a 576×3126 input matrix and a 64×576 weight matrix. Multiplying those matrices requires 115 605 504 MAC operations.

Those billions of MAC operations for just a single inference give a first impression on how computationally demanding CNNs can become. However, there is one major advantage that we can exploit when computing matrix multiplications. DNN computation is considered an *embarrassingly parallel* workload. That means in basic terms that it is simple to identify many independent computation steps, which can all be executed in parallel. In particular, in matrix multiplications each cell in the result matrix is entirely independent of all others, hence the example from above allows for $3126 \cdot 64 = 200\,064$ independent computations. To leverage this massively parallel nature, we can, e.g., deploy multiple processing units that can compute MAC operations in parallel. However, solely adding more MAC units to a given hardware architecture shows slowly diminishing returns. As shown in Figure 2.19 each MAC operation requires one weight, one input, the previous partial sum (p_{sum}) and returns an updated partial sum; four load / store operations per MAC. Considering the billions of MACs in recent CNNs, we face a new challenge: computation of ever-increasing CNNs becomes a so-called *memory bound* problem, as we struggle to move the massive amounts of data into and from the accelerator chip.

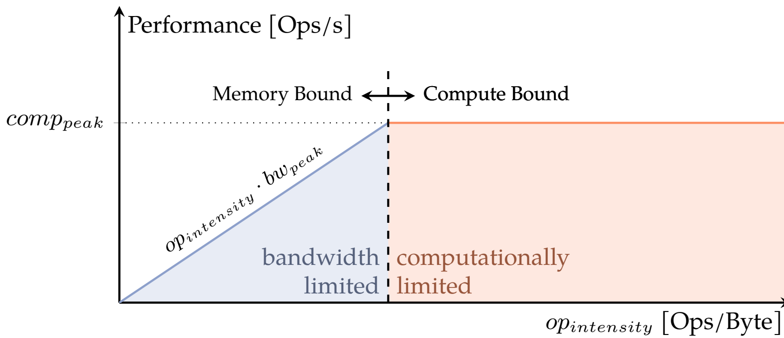


Figure 2.20: Overview of the Roofline model that helps to assess whether applications are memory or computaionlly bound [114].

To get an impression of memory vs. compute bound problems, we can take a look at the *Roofline model* [114], which is shown in Figure 2.20. It offers a generalized way of evaluating computational problems. A given problem is considered compute bound (orange area) whenever computational capabilities ($comp_{peak}$) of a system are lower than the amount of data that can be provided from memories and communication networks. In contrast, memory bound problems are constrained by data supply from memories (blue area). As a result, computation resources will suffer from underutilization. An optimal workload settles at the transition point. Here, memory capabilities match the computation capabilities, resulting in theoretically full utilization of available resources.

In general, a DNN inference is very hungry for data, since matrix multiplications barely need any control flow and are marked out mostly by dataflow [116]. If we look at DNN accelerators, computational capabilities are merely limited by the number of available MAC units. Limits set through data supply are, however, more complex to determine. A very simple accelerator, similar to what is shown in Figure 2.19, may load every single operand and partial sum from a memory and store the resulting partial sum back to this memory. While this is not only very inefficient (4 load / store operations per MAC), associated memories have to be huge and fast for a high-throughput inference. As a reference, ResNet-50 [40] then requires 15.45 billion load and store operations, which equals to 61.8 GB data movement considering 32-bit floating-point numbers, and the network requires at least 3.1 MB memory to store the largest intermediate feature map. To put this into perspective, with modern silicon

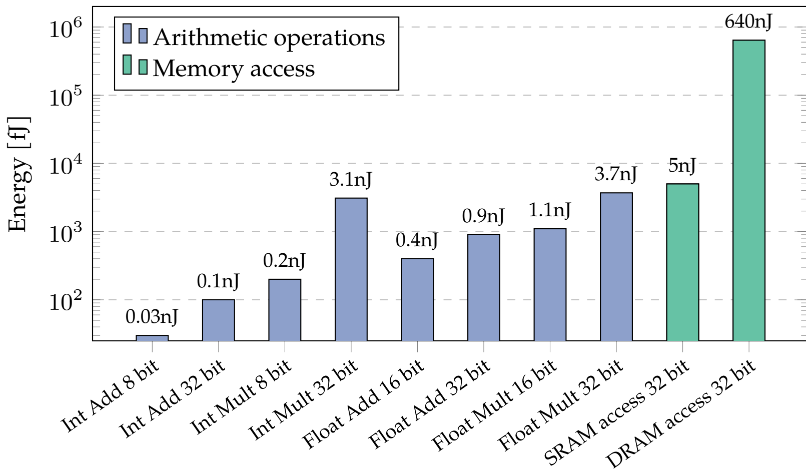


Figure 2.21: Energy breakdown of common instructions that appear in DNN inference or training. Evaluated using 45 nm CMOS with 0.9 V supply voltage. Data originates from Horowitz et al. [115].

technology we can have hundreds of MAC units running at fast speeds over 500 MHz on a single chip[3]. However, local on-chip memories, which are capable of holding all required data for this network, have to be massive in terms of area. While off-chip memories have larger capacities, they are often not as quick and energy-efficient as required. Looking at Figure 2.21 we can see that a 32-bit load from an off-chip memory realized as Dynamic Random Access Memory (DRAM) requires about 100× more energy than a 32-bit load from a local on-chip Static Random Access Memory (SRAM).

To address this, we can employ smarter techniques. Matrix multiplications offer besides independent operations the possibility to reuse data. For example, one row of the first operand matrix can be reused with all columns of the second operand to compute the first row of outputs. Based on this characteristic, we can start to think about hierarchical memories and thus combine the advantages of large but slow off-chip and smaller but faster on-chip memories. This concept is already successfully in use for hierarchical CPU caches. A typical SoC memory hierarchy with a DNN accelerator is given in Figure 2.22. Usually, the accelerator itself is made of some

[3] For example, a Xilinx Zynq 7020 device already has 220 [117] DSPs on a relatively small die.

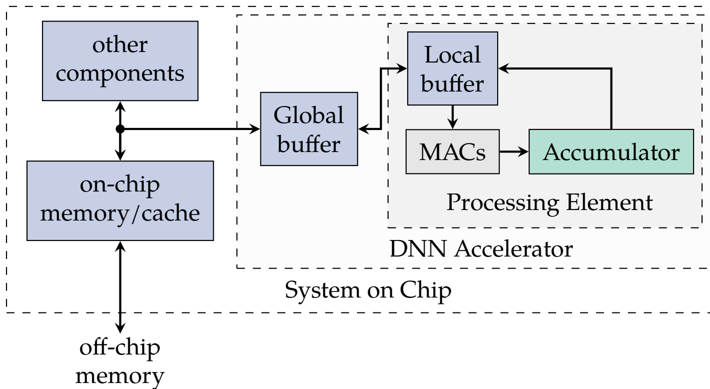


Figure 2.22: Typical memory hierarchy and components of an exemplary output stationary DNN accelerator embedded into a SoC, following [21].

hundreds MAC units. Each has a small buffer, oftentimes a simple register, e.g., for accumulation of intermediate results or to reuse a weight or input. A MAC unit together with those small bits of storage form a so-called *Processing Element* (*PE*). An array of PEs typically builds a DNN accelerator that has another local memory, which can be used to preload weights or inputs for the next batch of computations, enabling fast update of the PE's buffers. The DNN accelerator with its PEs and local memory may now be integrated into an SoC that features a large local memory shared with other components. From here, a high-speed connection to an off-chip memory for offloading of intermediate results and fetching weights of layers other to the one that is computed right now, is available. Memory sizes usually become larger and chip area per memory unit becomes smaller the further away they are from an actual MAC unit, but they also get gradually slower. Therefore, it is crucial for an efficient DNN accelerator to carefully lay out a well-optimized memory hierarchy, as larger local buffers boost the processing speed but at the same time draw from the energy and area budget. Moreover, it is important to identify an efficient mapping strategy. The mapping of a DNN workload basically determines, which data is reused, and which is offloaded. We will talk in detail about mapping tools and strategies in Subsection 3.2.1. Here, we have to keep in mind, that defining the memory hierarchy has to happen in accordance with characteristics of the DNN workload and requirements given by the use case.

2.5.3 General Purpose Hardware Platforms

Now that we discussed how DNNs are theoretically computed, we can now look into actual platforms able to process workloads presented by DNNs. Readily available libraries support processing MAC operations in a fast manner on almost any hardware. Hence, training and inference of DNNs is very accessible for generic computation devices like CPUs. A very common software implementation for linear algebra like matrix multiplications occurring in DNN workloads is provided by Basic Linear Algebra Subprograms (BLAS) libraries [118]. They have been ported to a manifold of different compute platforms and were developed to become highly efficient [119].

On CPUs for instance, BLAS leverages SIMD instructions for efficient DNN computation and dispatches the workload to multiple available CPU cores in parallel. Due to their simple programmability and availability, first DNNs around the 2000s were computed exclusively on CPUs. Until then, DNN workloads still had a manageable size and complexity, for which CPUs offer sufficient performance. However, with rising complexity and the advent of CNNs the performance was unable to keep pace and innovative technologies were demanded.

As a result, around 2010 the processing of DNNs moved from CPUs to GPUs [38]. In contrast to vector units in CPUs, which usually work on 8 or 16 data in parallel, GPUs achieve much higher degrees of parallelization. Originally meant for graphics processing, GPUs feature many more processing units that compute MAC operations. However, those compute units are hard to use efficiently for other workloads than graphics tasks. The breakthrough of GPUs for DNN computation was then only made possible through GPGPU (see Subsection 2.4.2). In fact, the increased use of GPUs for general purpose tasks, gave spark to the most recent advances in DNNs, which until then were hindered by the limited computational resources. DNN benefit from the many thousand SIMD streaming processors in GPUs, which can compute matrix multiplication in a massively parallel way. The aforementioned transformations and GPGPU programming, enabled CNNs like AlexNet [39] to distinguish 1000 different image classes in the first place. Today, GPUs are widely supported by ML frameworks.

Until today, GPUs play a significant role when it comes to DNNs, especially for training of large models, as modern GPUs offer large local and off-chip memories. However, slowly but steadily requirements for DNN inference and training drifted apart from the capabilities of even modern GPUs. Especially in embedded systems, GPUs are sometimes not the right choice, since they require too much energy [120].

To address the rising demand, also for embedded applications, GPU manufacturers introduced new architectures, faster interconnects, and larger memories. For example, NVIDIA started to add *Tensor Cores* in their Volta architecture [121] in 2017, which enable fast and more efficient computation of DNNs. Similarly, embedded GPUs like NVIDIA's Jetson entered the marked for more energy constrained environments. However, after all, GPUs kept the ability for computing graphics tasks like ray tracing or rendering screen output, which are all realized in hardware. Those hardware units are not in use when training or inference of a DNN workload happens. Hence, to build even more efficient hardware platforms for DNNs they can be omitted.

2.5.4 Systolic Arrays and Tensor Processing Units

Today, the most powerful hardware platforms for both DNN inference and training are systolic arrays and Tensor Processing Units (TPUs) [122; 123]. Those architectures focus only on computing matrix multiplications and got rid of the ability to process general purpose workloads. To compute matrix multiplications efficiently, they are solely composed of MAC units. Registers or small buffers are typically added to each MAC unit to enable data reuse and eliminate unnecessary memory loads and stores. With modern silicon technology, multiple hundred computational units are placed onto a single DNN accelerator. Thus, they deliver high computational performance and consume only little energy compared to more generic computation devices.

A typical example and structure of a systolic array is given in Figure 2.23. As shown in the cut-out, a set of registers and a MAC unit are pooled as Processing Elements (PEs). Here, the PEs are arranged and connected in a two-dimensional mesh. Other topologies are also conceivable, but for DNN computation, two-dimensional is the most common. The figure highlights how data in a systolic array is reused: Since all PEs are connected in a two-dimensional mesh, an operand is shared with a neighboring PE after it got used inside the PE, which is referred to as spatial reuse of data. In other words, after one computation cycle, one operand is passed from left to right and the other operand from top to bottom. The example shows in particular an Output Stationary (OS) systolic array. This type of dataflow refers to the characteristic that intermediate results or outputs stay in each PE until the computation has finished. Hence, the PE has an accumulator register that holds the partial sum, which is then reused in the same PE. This is an example for temporal reuse.

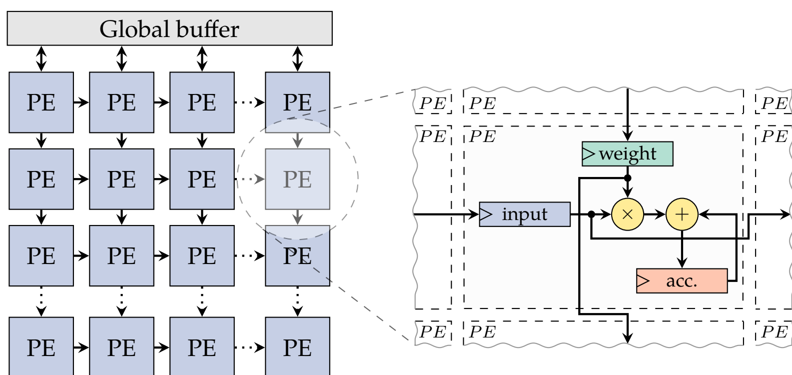


Figure 2.23: Hardware structure and dataflow inside a systolic array, according to [28]. A cutout shows the insides of an output-stationary processing element, with registers for weights, inputs, and accumulation as well as the components of the MAC.

If one thinks about the nature of matrix multiplication, we see that it can be represented as a three-fold nested loop, in which each loop is independent of each other. Hence, besides OS, we can also implement Input Stationary (IS) and Weight Stationary (WS). While an OS arrays keep the output, an IS or WS reuses a weight or input temporally [124]. Likewise, an IS or OS array passes intermediate results from one PE to the next and adds its intermediate result to the partial sum. Figure 2.24 visualizes how individual PEs are supplied with data from a global buffer, and which operand is held stationary in a local register. Operands which are stationary have the least load and store operations from a buffer. As a consequence, this operand also has the smallest impact on the energy budget. Hence, it is hard to tell whether one or another data loading pattern is more efficient, but it depends on the workload. As specific example Listing 2.2 gives a pseudocode implementation for CONV layer computation with all three taxonomies. CONV layers are known for their large input and output feature maps but have fewer weights, however, one cannot immediately exclude WS. In fact, multiple filters may be the evaluated in parallel, achieving a high utilization of the array, which is also done in many well-established accelerators [7; 125].

Besides these three straightforward taxonomies, over the years different other data loading patterns and kinds of systolic arrays were presented. Typically, they employ a larger PE buffer to store not only a single operand for the next operation, but multiple. One notable example for such dataflow is the Eyeriss series [60; 126] by Chen et al.

```
# Hold m, p, q
    stationary as
    long as possible
for m in range(M):
for p in range(P):
for q in range(Q):
  for c in range(C):
  for r in range(R):
  for s in range(S):
    o[m,p,q] +=
      i[c,p+r,q+s] *
      w[m,c,r,s]
```

(a) Output Stationary

```
# Hold m and c
    stationary as
    long as possible
for m in range(M):
for c in range(C):
  for r in range(R):
  for s in range(S):
  for p in range(P):
  for q in range(Q):
    o[m,p,q] +=
      i[c,p+r,q+s] *
      w[m,c,r,s]
```

(b) Weight Stationary

```
# Hold c, p, q, r, s
    stationary as
    long as possible
for c in range(C):
for p in range(P):
for q in range(Q):
for r in range(R):
for s in range(S):
  for m in range(M):
    o[m,p,q] +=
      i[c,p+r,q+s] *
      w[m,c,r,s]
```

(c) Input Stationary

Listing 2.2: Python-like pseudocode implementation of different dataflows to compute CONV layers. The computation itself stays the same, only the loop order is changed. Here N is set to 1, for $N > 1$ another loop has to be added.

They introduced a row stationary dataflow. Here, they preload a row of, e.g., a 3×3 filter into the local buffer of a PE to reuse an entire row of a convolution filter. In the next chapter, we will discuss in detail various state-of-the-art accelerators, their dataflow and why they in particular perform well for a given workload.

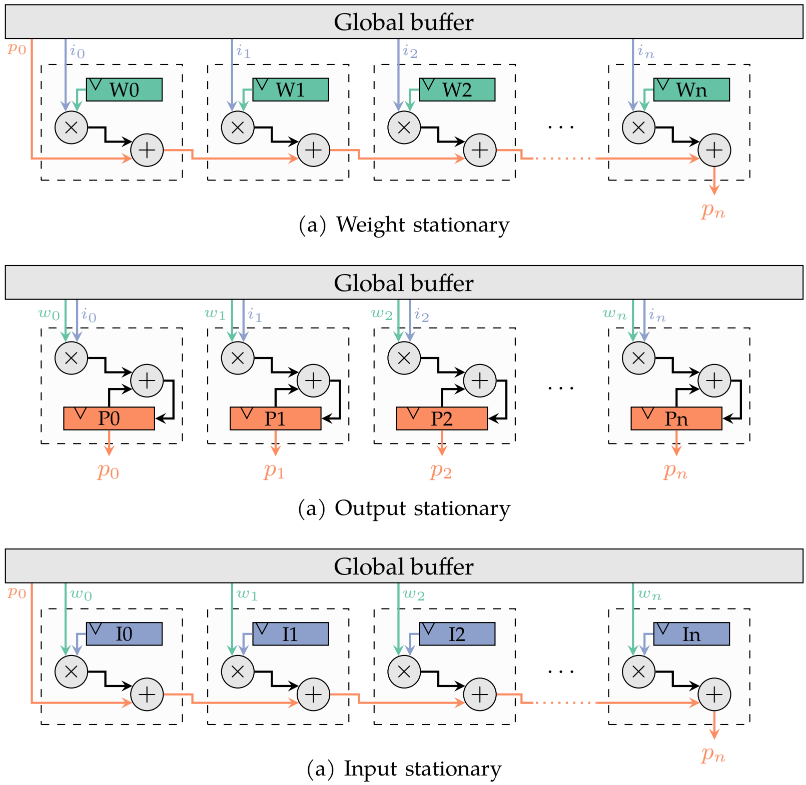


Figure 2.24: Different types of dataflows, which are commonly seen in DNN accelerators, according to [28].

Chapter 3

State of the Art in Deep Neural Network Acceleration and Optimization

The following chapter gives an overview of the current state of the art in the field of hardware accelerated deep learning. First, we will review some of the most notable accelerators that were presented by industry and academia. However, for ubiquitous Deep Neural Network (DNN) deployment many challenges like fast inference, low-power consumption and high accuracy remain [127]. Therefore, the next part of this chapter will introduce design methods and tools such as Neural Architecture Search (NAS), advanced techniques for mapping DNN workloads onto accelerators, and approaches for algorithm/accelerator co-design. All aim to realize efficient DNN topologies and fast inference on the corresponding accelerators. Finally, we will look deeper into DNNs and how we can exploit their over-parametrized nature to make the inference faster, more resource efficient and less energy consuming. Therefore, we will give an overview of quantization and pruning and how they can be implemented efficiently in hardware accelerators. This chapter closes with remaining open questions for efficient DNN accelerator design, of which some are in-depth studied and addressed in this thesis.

3.1 Popular DNN Hardware Accelerators

As the entire field of DNNs gained massive attention in 2010, many novel hardware architectures to address the huge computational workloads were proposed. In the background section, we already covered the basic principle of hardware-aided DNN

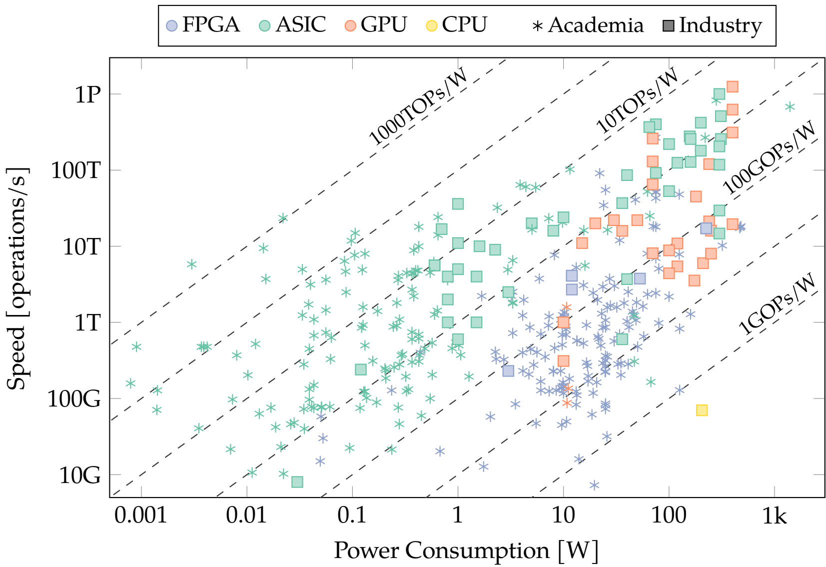


Figure 3.1: Overview of recent DNN accelerators grouped by platform and whether they are a commercial product or research accelerator. Data from [129].

inference and training. Moreover, we discussed that, up until today, many tasks are served by Graphics Processing Units (GPUs). As GPUs start to fail to deliver sufficient performance, dedicated accelerators become increasingly important, especially when one considers highly constrained systems like embedded or battery-powered devices. To address this, the frequently updated survey works by Reuther et al. [128] and Guo et al. [129] give a great overview of currently available accelerators. In this section, we will introduce some of the most impactful works. Starting with commercial accelerators that can already be purchased and continuing with works from research. The academic section will of course only cover some few of the hundreds presented approaches. Accelerators that focus on optimization strategies like pruning or quantization will be covered in the next section when these terms are introduced.

To get a first impression of the huge range of accelerators, Figure 3.1 shows an overview of recent architectures [129]. For each accelerator, the operational speed, and the associated power consumption, as well as whether they originate from industry or academia, is shown. As reference to compare individual accelerators, the efficiency in terms of operations per watt is shown with diagonal lines. From the plot, it can be

seen that only a few works have surpassed the 100 T OPS/W mark yet. Moreover, we can observe that works from academia are rather on the energy-efficient end, while commercial architectures focus rather on higher throughput. This is because most research focuses on efficient systems and most commercial products cover the large datacenter market for training infrastructure.

3.1.1 Commercial DNN Accelerators

Commercial architectures provide the large backbone infrastructure, which is necessary to train large-scale networks. Hence, this section will start with those large architectures for datacenters and then moves right down to embedded accelerators. Most of the training and large-scale inference workload is addressed by datacenter-grade GPUs. To fulfill the rising need for faster computation of ever-growing DNN topologies, NVIDIA, one of the leading companies for GPUs, started to integrate dedicated TensorCores into their cards. Those cores, which are essentially systolic arrays, enable the most recent GPU generation to compute DNN tasks fast and efficient [121; 130]. Similarly, Field-Programmable Gate Array (FPGA) manufacturers stated to integrate more sophisticated Digital Signal Processors (DSPs) and even dedicated DNN accelerators into their chips. For example, Xilinx's most recent Versal generation features dedicated AI cores for efficient computation of large matrix operations, as they occur in DNNs [131].

To address the market demand for even faster model training, Google released their Tensor Processing Unit (TPU), a dedicated datacenter accelerator [7]. Already their first version from 2017 surpassed GPUs with an inference speed-up of 15× and 70× increase in efficiency, respectively. The TPU is essentially a systolic array with a Weight Stationary (WS) dataflow. Google evolved their datacenter accelerator over four generations and even presented an embedded version, the Edge TPU. Today, version four has twice as much memory, more memory bandwidth and can be linked together with more units to significantly increase the performance compared to the first generation. Unfortunately, Google has not released detailed information on the internal architecture of their latest TPU.

Another example, of a datacenter accelerator is the chip proposed by the European Processor Initiative [132]. Their heterogeneous multicore architecture will cover a wide range of applications from traditional High-Performance Computing (HPC) workloads to energy-efficient DNN computation using, for example, optimizations

such as variable precision. The race for fast and efficient DNN accelerators, also attracted some newly founded start-ups. Notable companies are Graphcore with their Colossus architecture that delivers up to 280 T FLOP [133], and Cerebras Systems who presented a wafer-scale DNN accelerator [134].

Moving more into the embedded domain, Tesla introduced their Full Self-Driving Chip (FSD) [135] in 2019. Their System-on-Chip (SoC) features, besides 12 Central Processing Unit (CPU) cores and a GPU, a two core Neural Processing Unit (NPU) operating at 2 GHz. The NPU is arranged as a matrix of 9216 Multiply-Accumulate (MAC) units coupled with 32 MiB Static Random Access Memory (SRAM) and dedicated units for pooling and activation [136]. The chip with 6 billion transistors is manufactured in 14 nm and delivers 36.68 T OPS per NPU using 8-bit integer precision, while consuming only 36 W. With their FSD, Tesla offers a platform for their future autonomous vehicles. Besides Tesla, NVIDIA also has a series of embedded DNN accelerators. NVIDIA's Jetson [137] series is available in a wide range of different configurations. The large AGX Orin, e.g., delivers over 275 T OPS. While the smallest modules, like the Jetson Nano or Jetson TX2, show great cost and power efficiency.

In the last few years, DNN accelerators moreover found their way into modern smartphones. For example, Apple's A16 processor [6] features a 16-core NPU that delivers about 17 T OPS. Another example is Qualcomm's current Snapdragon 8 Version 2 architecture [138] that yields 4× better DNN performance with 4-bit integer precision, manufactured in a 4 nm process.

3.1.2 Academic and Research DNN Accelerators

With the first presentation of sophisticated DNN models like AlexNet [39] in 2012, the interest for fast computations of those homogenous workloads rose in academia. One of the first accelerators that specifically targets the computation of DNNs is DianNao [139] from 2014. The accelerator uses an Output Stationary (OS) dataflow (see Subsection 2.5.4). It features 256 16-bit multipliers with adder trees and achieves a performance of 452 G OPS at a power consumption of 0.48 W. This performance has been enhanced over three generations: DaDianNao [140], ShiDianNao [141] and PuDianNao [142]. The latter two enabled support for Convolutional Neural Networks (CNNs) and seven other Machine Learning (ML) algorithms. Those four accelerators form the DianNao family, which is considered a pioneering work in DNN acceleration and is often used as baseline for other accelerators [143].

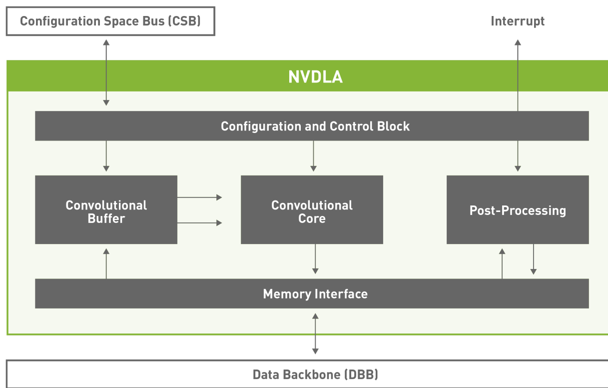


Figure 3.2: Block diagram of NVDLA [146].

Other notable OS architectures are the works by Gupta et al. [144] or Moons et al. [145]. Both accelerators are systolic arrays and reuse activations across columns and weights across rows of MAC units. As a result, they can process multiple Convolutional (CONV) filters and inputs at the same time. In contrast, DianNao processes convolution operations one channel at a time. Gupta et al. implement a 28×28 systolic array on an FPGA, achieving 37 G OPS/W using a comparably small Kintex K325T device. While the work by Moons et al. is an example of an Application-Specific Integrated Circuit (ASIC) design. They integrated 16×16 MAC units, achieving up to 2.6 T OPS/W, which can run AlexNet at 47 FPS.

Besides OS some works from academia followed the approach of a WS dataflow. For example, Origami [147] presents a scalable WS DNN accelerator for Computer Vision (CV) applications. They implemented their design into an ASIC that achieves 369 G OPS/W, which is comparably to workstation GPUs at that time but with much less energy consumption. NVIDIA's Deep Learning Accelerator NVDLA [125] is a very popular work that implements a WS dataflow. A block diagram is given in Figure 3.2. It is available as open-source and comes in two configurations: large and small, which determine the number of modules and the accelerators' complexity and capabilities. Additionally, NVDLA offers many more options for configurability. Everything from the used data format, over the number of MAC units and their arrangement, to the buffer and Direct Memory Access (DMA) sizes might be adjusted. Hence, it is used in many works as a reference or basis. The basic principle of the NVDLA was developed further by Shao et al. to their Simba accelerator [148]. Simba

is a large-scale accelerator that integrates multiple MAC units arranged as vector processors into chiplets, which in turn are combined into a large silicon device. They demonstrate that an accelerator design with 36 chiplets can process ResNet-50 with 1988 FPS and achieves an efficiency of 6.1 TOPS/W.

In 2021, Genc et al. presented Gemmini [149], which is a DNN accelerator generator. With Gemmini systolic arrays with either OS or WS dataflow can be generated. The generated accelerator is integrated into the Chipyard [150] ecosystem. As a consequence, one can build entire SoCs quickly and test them in simulation or even on FPGAs. Since Gemmini is available as open-source software and comes with an entire ecosystem, it serves as a great research platform for accelerator modifications.

In addition to the introduced WS and OS dataflows, an example of an accelerator that implements an Input Stationary (IS) dataflow is SCNN [151]. Here, data from the input feature map is reused across the CONV filters. The Eyeriss series by Chen et al. [60] proposes a row stationary dataflow. This dataflow stores a row of input and filter data locally. A row thereby matches the dimensions of the filter. Hence, each Processing Element (PE) has registers to store and entire row of data. The authors argue that this kind of dataflow maximizes data reuse across inputs, weights, and partial results at the same time, resulting in a high overall energy efficiency. Furthermore, both works support pruning that increases their efficiency. Hence, they will be described more in detail in the next section.

Some recent works even explore the potential of supporting multiple dataflows simultaneously by using flexible interconnects that connect the PEs and memories. The goal is to unify the advantages of all presented dataflows and to achieve an unprecedented utilization. FlexFlow [152] is one example of an accelerator that supports a large range of different tiling strategies and dataflows. They are able to reduce the number of energy expensive Dynamic Random Access Memory (DRAM) accesses per operation significantly. As a result, they demonstrate a 2.5 to 10× power efficiency increase over Eyeriss and DianNao by testing six workloads. MAERI [153] proposes a flexible dataflow accelerator that eliminates some restrictions of FlexFlow like fixed dataflows within layers or the support of only CNNs. Their PEs can be configured at runtime though switches that route the dataflow. The authors evaluate their accelerator against systolic arrays, showing a 6.3× reduction in SRAM reads, and a speed-up of 72.4 % compared to the row-stationary dataflow proposed by Eyeriss. In total, MAERI shows an 8 to 459 % better utilization across multiple workloads and static dataflows.

So-far we have only discussed DNN accelerators that build on traditional, design patterns and Complementary metal-oxide-semiconductor (CMOS) technology. However, this excludes a hole body of research that was carried out besides that. Some recent works explored near-memory or in-memory computing [154], which shifts the computational units into the large memory banks that are required for DNN computation. Near-memory computing hereby aims to build smarter memories that reduce memory transfers on the fly, while in-memory approaches build on analog memories that can for example accumulate data directly inside a memory cell. The goal in both is to reduce the energy consumption for data movement. However, modern dataflows in particular for CNNs leverage the large potential for data reuse in the accelerator, which makes a conventional accelerator without in-memory computing faster. Hence, in-memory computing can be considered for data-hungry workloads like Multilayer Perceptrons (MLPs) [155]. Novel memory technologies like non-volatile memories, which are not cleared when power is removed, allow for new strategies of DNN computation [156]. For example, weights can be stored in such memories, which allows for fast online training and eliminates the need to download weights before inference.

Worth mentioning are moreover neuromorphic architectures that extend in-memory computing approaches with analog computation [157]. The idea here is to mimic biological brains closer than the presented Artificial Neural Networks (ANNs). In neuromorphic systems MAC operations are realized in the analog-world with resistors that add together voltages. This operation is orders of magnitude more energy-efficient compared to a digital MAC unit. However, analog computers still lack efficient and fast analog to digital converters. Moreover, noise from the digitalization accumulates over layers. While specifically designed DNN topologies can compensate for the inaccuracies, this is still a large challenge for analog computers. Other popular examples of neuromorphic hardware are Spiking Neural Networks (SNNs). They encode data not in binary representation, but as pulses with different frequencies depending on the activation, which is similar to the encoding of signals in biological brains. To support this multiple hardware platforms were presented in the last decade, for example SpiNNaker [158] or Neurogrid [159]. However, SNNs are currently lacking sufficient and fast training algorithms like backpropagation used for DNNs. In addition, designing fast inference hardware for SNNs is challenging, as the timing and quick sampling of individual spikes play an important role [160]. Hence, a combination of digital and analog hardware is heavily investigated to make fast SNNs feasible and to benefit from increased energy efficiency.

As we have seen, the last decade was marked by numerous remarkable works to accelerate DNN workloads. However, what kind of dataflow, accelerator structure, interconnect and memory technology results in the best energy-performance trade-off is still an open research question [62]. After all, to build an efficient DNN accelerator one has to systematically analyze the myriad of different design possibilities.

3.2 Design Methodologies for DNN Accelerators

The previous section gave an overview of the vast range of different DNN accelerators. A key characteristic of the presented accelerators is the dataflow. Obviously, there is no single dataflow that works best for all kinds of DNN workloads and performance requirements. In contrast, one has to carefully tune the accelerator to the expected workload. This makes a co-design between both indispensable [161]. Consequently, over the last few years many works have been presented to address this algorithm and accelerator co-design, an efficient workload mapping and Design Space Exploration (DSE) of different design options. This section will broadly cover all aspects starting from mapping and compilation methods to NAS and DSE tools.

3.2.1 Mapping

In addition to an efficient DNN topology, a well-tuned mapping is also crucial to efficient DNN inference and training. A mapping describes which PE of a DNN accelerator executes which kind of instruction at a given time. Sometimes a mapping is also referred to as scheduling and a mapper as a compiler. The analogy is not by chance, a mapper works similar to a compiler that creates machine code for a given Instruction Set Architecture (ISA). An overview of the inputs and exchange formats of a mapper for DNNs is shown in Figure 3.3. As input, a mapper takes the specification of the workload like its dimensions, as well as implementation details and constraints of the target DNN accelerator. These include, e.g., the capabilities of the accelerator or the number of available PEs and their arrangement. Output of the mapper are not only the mapping itself, but also configuration parameters for the DNN accelerator. With a mapping, the accelerator is instructed which input data and weights it should request from external memory, when it is replaced in the internal memory and how often it is reused. Especially, the reuse factor of individual pieces of data has a substantial impact on the energy efficiency, as data movement draws

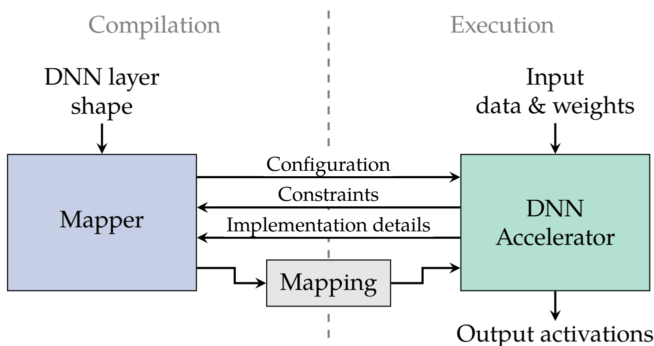


Figure 3.3: Mapping process of a DNN workload onto a given DNN accelerator. A mapper generates a mapping based on the constraints of an accelerator. This mapping is a sequence of instructions that the accelerator processes. Figure adopted from [162].

heavily from the energy budget. Tiling also plays a significant role for the efficiency of the mapping. As described in section Subsection 2.2.1, individual loops, e.g., of a CONV operation are independent of each other and can be swapped or split to fit in the available memory and be unrolled or fused to reduce the loop-depth. This is referred to as tiling.

Based on these degrees of freedom, a mapper usually generates a set of feasible mappings, called the map space. Figure 3.4 shows the internals of a mapper. The map space is constructed by the accelerator and workload specification and constraints. Then found mappings from the map space are iteratively evaluated using an architecture model, which provides figures like energy, latency or throughput. Depending on the complexity of the DNN accelerator and the workload's degrees of freedom, finding an efficient mapping can be an extensive endeavor. For some small workloads, it might be possible to exhaustively explore the entire search space, but for modern workloads the search space has to be constrained to make an exploration feasible. Heuristics can help to drop inefficient mappings and select the most efficient ones.

To address the complex mapping process, today a range of tools are available [164]. Some of the most popular mappers are TVM [165] from 2018, DLVM [166] from 2018, Timeloop [163] from 2019 and MLIR [167] from 2021. To support the myriad of different DNN architectures, topologies and frameworks, all mappers take standardized input formats. Mappers typically extract the DNN topology from an

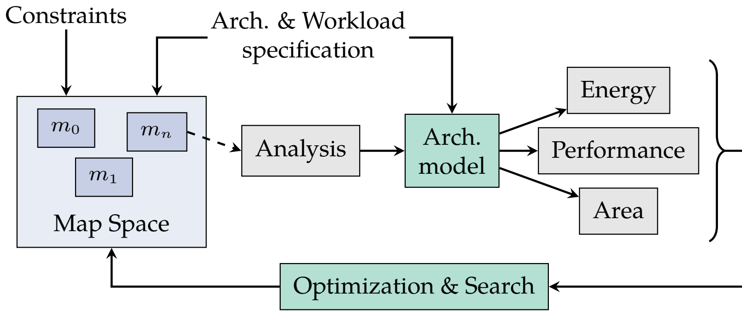


Figure 3.4: Principle of a DNN mapper. Form a set of specifications and constraints,it builds a map space that is iteratively analyzed with an architecture model to find the best fitting mapping. Figure based on [163] and [28].

Open Neural Network Exchange (ONNX) file [168], which can be generated by all commonly used DNN frameworks. An architecture description is given in a template form. Even though there is no established format, the templates look similar across tools and describe all elements of the architecture, like the memory hierarchy and the PE's capabilities. From these input formats, the mapper tools can generate the map space, which is usually encoded using a custom representation [28]. Search heuristics to identify the best mapping in the map space may vary across the tools. For example, Timeloop uses random parallel search and TVM simulated annealing. Energy, performance, and latency estimation works also slightly different among various mappers. To get an estimate, TVM uses a dedicated DNN model to predict the metrics, which is quick and accurate. However, DNN models lack flexibility for new and yet unknown architectures. Hence, another approach is to use analytical models. For example, Timeloop can be coupled with the tool Accelergy [169], which estimates energy and area consumption based on an architectural template and the associated action counts. For DNN accelerators, those action counts, e.g., how many bytes were read from a given memory, can be generated automatically.

3.2.2 Data Orchestration

As stated in Subsection 2.5.2, accessing data from an external memory accounts for the largest share of the accelerator's energy budget. However, we cannot store all required data locally. As a consequence, hierarchical memories established themselves.

Designing them are a crucial step for efficient execution of a DNN workload on a given accelerator. Thus, the way buffers are organized and controlled is besides an appropriate dataflow an essential design characteristic. According to [21] who in-depth investigated the impact of different organization taxonomies, an efficient DNN accelerator design should focus on the following aspects: All data should be transferred just before it is needed, this data should overlap with the data that is currently in use, and finally it should be replaced exactly when it is no longer needed. In addition, this data movement should be achieved in a fast buffer that only adds a minimal synchronization overhead.

While CPUs bank on caches to keep upcoming data and instructions local, integration the very same caches into DNN accelerators may not be advisable. For the highly regular data access pattern of DNN workloads the heuristic replacement policies of caches are not required and add unnecessary overhead. Instead, a more explicit and unmanaged form of data buffering can be applied [170]. Hence, a broad range of DNN accelerators like GPUs rely on scratchpad memories, which are essentially local buffers that are filled and flushed on request. The utilization and what kind of data is loaded into the buffer is, as a result, part of the workload mapping. In addition, decoupling the initiator of a memory request and the local memory itself supports the accelerator's performance [171]. This allows to reduce the latency as each unit can run at its own clock speed. Today, most modern DNN accelerators follow an Explicitly Decoupled Data Orchestration (EDDO) pattern to manage local inputs, parameters, and intermediate results. For example DaDianNao [140], SCNN [151] and the accelerator by Fowers et al. [172] implement EDDO.

3.2.3 Neural Architecture Search and Design Space Exploration

As we have seen so far, there are multiple design option that can heavily influence various DNN accelerator metrics. Neural Architecture Search (NAS) is a method that basically describes an automated way to design a DNN topology [173]. Leaving hardware aspects aside for now, designing a DNN topology, i.e., the number of layers, kernel sizes etc., for a given task is mostly up to the developers' knowledge and experience. However, this can lead to superfluous kernels or layers and thus unnecessary parameters and computations. Hence, pioneer works in NAS like NAS-RL [174] or DARTS [175] help to automatically generate DNN topologies that achieved state-of-the-art classification accuracy in CNNs using Reinforcement Learning (RL) or gradient descent methods, respectively. Fast NAS is recently also enabled by

supernetworks, which is a DNN whose outputs reflect the structure and parameter of a target DNN. Approaches like Auto-DeepLab [176] or SqueezeNAS [177] can significantly reduce the search time for image segmentation topologies that also show better performance and efficiency.

However, hardware aspects play a key role when a DNN workload is deployed in real-world applications. Looking at hardware architectures that have some degrees of freedom like an adjustable memory size, designers typically perform a Design Space Exploration (DSE). The goal of this exploration is to identify architecture configurations that yield the best outcome. Depending on the design objectives, a wanted outcome might be, e.g, to find the smallest memory size that still achieves a given throughput. One DSE method that returns a globally optimal solution is exhaustive search. However, it is usually impossible to explore all options for huge design spaces like the one for DNN accelerators [178]. More goal-directed methods like simulated annealing sample random architecture configurations and keep solutions that improve the objective. In addition, based on a probability parameter, in case of simulated annealing a temperature, worse solutions are also kept for breaking out of local optima. Genetic Algorithms (GAs) are another popular kind of optimization algorithm for DSE. They are inspired by evolution processes as they happen in nature [179]. Here randomly selected design points are assessed and by recombination of the best performing ones, the GA yields optimal solutions after a couple of iterations. A more detailed description will be given in Section 5.4. GAs can, for instance, be used to optimize the scheduling and loop nesting of DNN workloads [180]. Besides the two mentioned, approaches like Bayesian optimization that utilizes Bayesian ML methods or RL are often found for DSE problems.

Considering the importance of the right DNN hardware accelerator, mapping and topology, many works investigated a combination of DSE for hardware architectures and NAS. This is referred to as *Hardware-Aware Neural Architecture Search* (HW-NAS) [181]. In addition to traditional NAS, HW-NAS also considers hardware aspects like available PEs and hardware costs like latency, implementation area or energy consumption. The topic gained attention in the past five years, when algorithms eliminated the need to train and evaluate each proposed topology [182]. Instead, today it is possible directly predict the model accuracy. First works dealing with HW-NAS extended RL approaches with hardware aspects. For example, MONAS [183], a framework that targets GPUs, incorporates a reward function that combines accuracy and energy consumption, resulting in an efficient DNN topology for a given hardware.

HW-NAS is especially popular for constrained embedded systems like smartphones. The authors of MnasNet [184] proposed the first automated method for mobile DNN design by optimizing towards accuracy and latency, derived from actual devices to get proper results. FBNets [185] by Wu et al. improved the approach of MnasNet with differentiable NAS that speeds up the search process. They achieve a slightly worse performance than MnasNet, but with 420× less search cost. This performance was increased by Gupta et al. [186] who extended the search space and found that depthwise convolution operations as they are used in MobileNet can lead to under-utilization in systolic arrays. Their work focuses Edge TPU devices and outperforms traditional ResNet-50 and MobileNet in terms of latency and accuracy. Another domain of resource-constrained systems are microcontrollers that only have limited memory. Therefore, recent works like μNAS [187] optimize DNN workloads for such systems. With their approach, they can create an MNIST model with less than 0.5 kB parameters and still over 99 % accuracy.

Besides those mobile applications, HW-NAS today plays also an important role when DNNs are deployed to datacenters or while designing accelerators for FPGAs. EfficientNet-X [188], for example, focuses on optimization of workloads for TPUs or modern GPUs with TensorCores. They define a set of accelerator-friendly measures and demonstrate a version of EfficientNet that runs twice as fast as off-the-shelf EfficientNet on the TPUv3 or NVIDIA's V100 GPU. For FPGAs designs, Fan et al. [189] apply differentiable NAS to find a suitable CNN topology. Their approach yields a 1.14× accuracy and a 12.4× speed improvement on CIFAR-10 compared to RL methods. Targeting multiple architectures, another notable work for HW-NAS is AutoTVM [190], a framework to automatically determine inference schedules that outperform hand-tuned ones.

Determining the energy, area, and performance of a dedicated hardware accelerator, typically relies on synthesis of each individual architecture, which is very inefficient and time-consuming. As a result, cost models for hardware architectures became popular, especially considering the homogenous dataflows, which occur in DNNs. For example, works like MAESTRO [191] or Timeloop [163] are capable of mapping DNN workloads on a given hardware description to estimate performance and utilization of the accelerator. Tools like Accelergy [169] or the tool by Yang et al. [192] were introduced to estimate energy and area of given hardware designs. While the latter two tools are not entirely limited to DNNs, they are widely used for HW-NAS. Especially, Accelergy is widely used for DNN accelerator architecture search. A high-level overview of the tool is given in Figure 3.5. As input, it takes an architecture

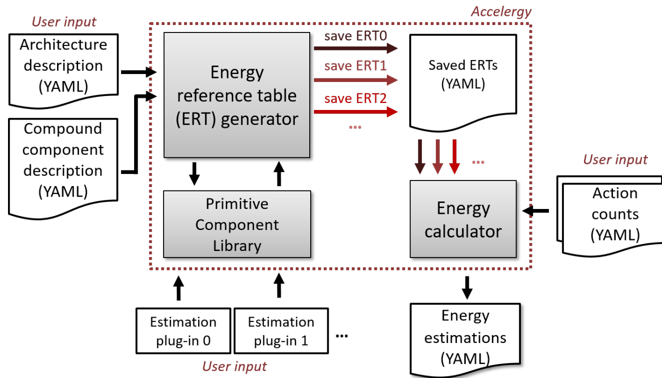


Figure 3.5: Overview of the Accelergy approach presented in [169].

description that lists all components, a library of those components, and an action count file. The latter lists all activity in each component and is thus used to estimate the energy by summing up the energy for active and idle phases in each component. Action counts may be generated during execution of a workload or through analytical computation. The component library is typically generated by synthesis tools and contains information like how much energy, e.g., an 8-bit adder needs per addition and how much area it consumes. But Accelergy also provides some component library to start with, e.g. a 45 nm library generated by Aladdin [193], that is also often used in HW-NAS works. Accelergy allows, moreover, adding plugins like CACTI [194] for energy estimation of memories. CACTI offers a wide range of models for DRAMs and SRAMs with different technology nodes. In combination with the plugins, Accelergy can accurately tell which component consumes how much energy and area.

3.3 Methods to Optimize Neural Network Inference and Training

While many hardware accelerators for DNN workloads and tools to map them efficiently onto those accelerators were introduced, research has found other means to further accelerate DNNs computation. Depending on the DNN input some activations are irrelevant. Instead of straightforward computation of all associated operations and loading all operands, the methods introduced in the following look exactly at those

operations and check whether they have to be computed. In addition, some methods exploit the fact that DNN are not returning exact results, but probability distributions. As a consequence, smaller network imprecision can be tolerated, and the result may change only a little or not at all. First, we will discuss different number formats that allow for a more efficient utilization of the memory associated with weights and inputs. Next, the closely related topic quantization is touched. Quantization allows compressing parameters and intermediate results using less precise representations and allows for less energy-intensive operations. Finally, we will deal with pruning, which identifies operands that are zero and skips the corresponding computations to save energy and computation time. All these methods need support from the tools, the network topology, and the accelerator itself. Hence, a selection of impactful corresponding works will be presented as well.

3.3.1 Number Formats

As stated before, DNNs can allow imprecision in operands, without losing its result significance. One of the most commonly used approaches to leverage this for increased energy efficiency and faster computation is the right choice of number format. Traditionally, DNNs are trained and computed using mathematically precise computations. Since digital computer systems cannot represent an arbitrary number precision and are usually limited to integers, computer architects found a representation that allows to encode a large range of real numbers, which is the floating-point number format [195]. A floating-point number consists of a sign bit (s), a mantissa (m) and an exponent (e). The first tells whether the number is negative or positive. The actual value of the number is then determined by the mantissa and exponent, as follows: $x = s \cdot m \cdot b^e$. Here, b is the number basis, which is usually 10. The number of bits that are invested into the mantissa and exponent define the precision and range of a floating-point number. While all level of precision are possible, some few have established themselves and are specified in the IEEE standard 754 [196]. Popular are single (FP32) and double precision (FP64) formats with 32 and 64 bits for the entire number, respectively.

The distribution of exponent, mantissa and sign bits for common floating-point formats used in DNN computation are shown in Figure 3.6. Single precision numbers (FP32) have 23 mantissa bits and 7 exponent bits, which allow them to represent fractional numbers very accurate. However, in DNN inference, where imprecision can be accepted, a lower precision may be applied. Fewer exponent and mantissa

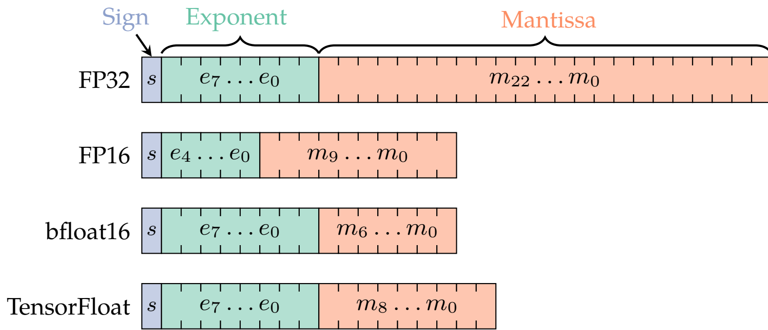


Figure 3.6: Examples of different established and modern number representations.

bits directly result in a lower memory footprint of the network. Hence, the inference energy is lower since fewer data has to be moved around. For example, a combination of FP32 for accumulation and FP16 for weights and activations even supports the model accuracy [197]. While FP16 is also specified in the IEEE standard, bfloat16, which was especially designed for efficient DNN training and inference [198], allows to get rid of FP32 entirely. It features two more exponent bits at the expense of two fewer mantissa bits. This allows bfloat16 to cover a larger number range by sacrificing precision. For reference, FP16 can only represent a range of $\pm$ 65 504, bloat16 extends this to $1.18 \times 10^{-38} \dots 3 \times 10^{38}$. This distribution of exponent and mantissa bits was shown beneficial for fast convergence of DNN training [199]. Thereby, it maintains the 16-bit alignment, which enables compatibility with data buses of common hardware platforms. Putting this compatibility aside, NVIDIA also introduced their TensorFloat format that extends the mantissa by two bits, which allows for increased precision.

However, floating-point arithmetic is computationally challenging and the required energy for a floating-point operation is comparably high (see Figure 2.21). Hence, many low-cost and low-energy General-Purpose Processors (GPPs) do not support floating-point operations at all. For example, during a multiplication, mantissa and exponent have to be computed individually, and finally the mantissa has to be adjusted in accordance to the new exponent. To address this, fixed-point number became popular in DNN computations [200]. In contrast to floating-point numbers, they work with a scale factor that determines how many of the available bits are considered fraction bits. Therefore, to convert a number into decimal representation, a given number is simply divided by two to the power of the scaling bits. For example, 10 with

2 scaling bits, becomes $^{10}/_{2^2} = 2.5$. The advantage of fixed-point numbers becomes apparent in computations. Especially multiplications, which are predominantly present in DNNs, can be done with simple integer arithmetic [105]. The resulting scaling factor is subsequently the product of the operand's scaling factors. Although some few works have proven that DNN training works with fixed-point number as well [144], they are mostly limited to DNN inference. This is because the linear scaling cannot represent the necessary precision for gradients to propagate backwards through the network. In addition, choosing the correct scaling factors is not a simple task.

DNN inference can also be computed using pure integers. Analogous to fixed-point numbers, integer operations are efficient in hardware and all CPUs support it. In this case, the scale factor is neglected, and all operands and results are scaled with the same factor. The result of, e.g., a multiplication is simply bitwise shifted to normalize the result to the common scaling. However, pure integer implementations also suffer precision, even slightly more than fixed-point implementations with arbitrary scale factors. As a consequence, they can mainly be found in inference application [127].

Besides those three very popular options, another notable number format in the field of DNNs is Posit [201]. This number format extends the range by adding a regime next to sign, exponent and mantissa. The regime works similar to the exponent but has a different base that depends on a seed value $useed = 2^{E_{max}}$, which is determined by the maximum exponent bits E_{max}. In comparison to IEEE-754 FP16, which has a range of $6 \cdot 10^{-8} \ldots 7 \cdot 10^{4}$, posits with $E_{max} = 1$ can cover a range of $4 \cdot 10^{-9} \ldots 3 \cdot 10^{8}$. For DNN computation, posits claim to deliver a $2\times$ to $4\times$ training acceleration, due to a simple computation of activation functions [202]. In addition, the mix of high precision and a high dynamic range that posits offer support DNN computation [203]. But, similar to floating-point arithmetic, posits require additional hardware effort. As a result, research is actively investigating various methods for efficient Posit operations, especially for DNN workloads [204].

Besides all the presented number formats, many others with special focus on DNN training, inference or both were presented [205]. However, floating-point remains one of the most popular formats for its wide availability in hardware architectures. Today, the optimized format bfloat16 is often used for training since most GPUs offer great support. Inference, however, tolerates more imprecision than training. As a result, it relies on integer and fixed-point formats, which unify sufficient precision with quick and low-energy arithmetic.

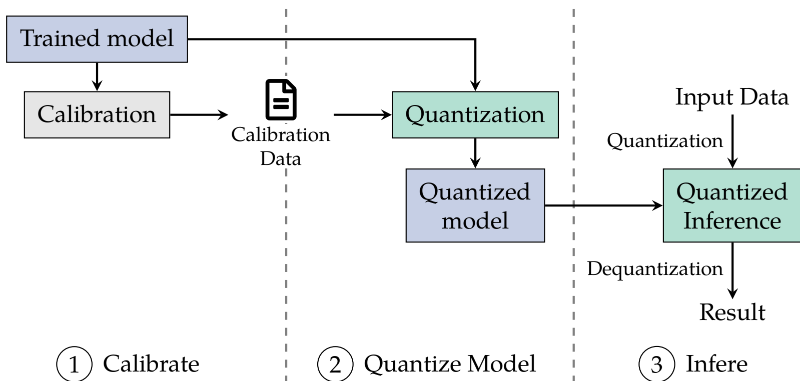


Figure 3.7: Steps required for quantized DNN inference. First, a model has to be calibrated to obtain the operand range. Then the model parameters are quantized. With this model, a quantized inference can be done.

3.3.2 Quantization

Mapping weights, biases and feature maps to less precise number formats is referred to as *quantization*. Similar to quantization used in signal processing, a value continuous input value is projected to a value space with discrete steps. Aggressive quantization works especially well with DNNs because they are usually over-parameterized [206], which was already noticed by the authors of AlexNet [3]. One of the most important aspects of quantization is the relation between original and quantized values, and the number of quantization steps that are used for quantization. These parameters heavily influence the quantization error, which has to be minimized to achieve good results. The quantization error is simply the difference between original and quantized values.

In a more formal way, quantization can be denoted as an original value x that is quantized by a quantization function $Q(\cdot)$, which returns the quantized value $\hat{x}$. Thereby, the number of quantization levels is defined as L and the corresponding set of quantization steps as q_i with $i \in [0, L]$. Putting this together, the quantization function maps an input x to the respective quantization step q_i.

Figure 3.7 shows the steps of DNN quantization [127]. Before the model parameters can be quantized, a calibration has to be performed. In this step, activations and intermediate results of a fraction of the dataset samples are tracked while passing through the network to determine range boundaries for quantization. Excessive

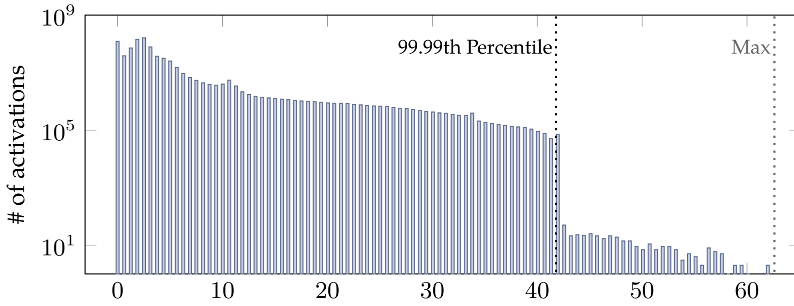


Figure 3.8: Example of clipping during quantization, shown as histogram with all activations of 1000 samples after the first CONV and Rectifier Linear Unit (ReLU) operation in a pretrained ResNet-50 and the 99.99th percentile as well as the absolute maximum.

values are usually dropped to not waste quantization steps for those large values, which is referred to as clipping. Clipping may happen based on a quantile, e.g., the 99.99 % or 99.9 % quantile or based on entropy to preserve as much information as possible [207]. As a reference, Figure 3.8 shows the distribution of activations of the first ResNet-50 CONV layer[4]. It can be seen that there are only a few excessive activations, which are dropped. The boundaries are denoted as α and β and clamp inputs and intermediate results of the network. They may be picked with $-\alpha = \beta$ or $-\alpha \neq \beta$, which is referred to as symmetric or asymmetric quantization, respectively. With this information, parameters of the network can be quantized and finally a quantized inference can be performed. Therefore, the inputs have to be quantized as well and results have to be converted back into original values.

For actual quantization, multiple methods have been proposed over the recent years. A straightforward method is uniform quantization, in which q_i is equally spaced. As a result, the quantization function boils down to $Q(x) = Int(x/S) - Z$, where S is a scale factor, Z is a zero offset [208], and the function Int is a proxy to convert a floating-point number into an integer representation. The scale factor for uniform quantization computes as follows:

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{3.1}$$

[4] Data generated using a pretrained ResNet-50 model provided by PyTorch and 1000 randomly selected samples of the ImageNet-1K dataset

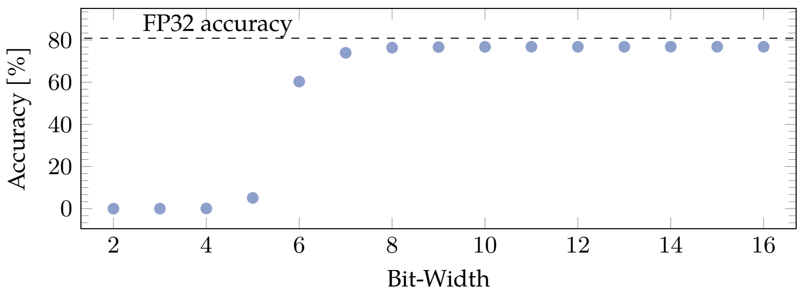


Figure 3.9: Impact of quantization on the accuracy of ResNet-50. Tests performed with same quantization for all parameters and activations. Shown is the Top-1 accuracy determined using the ImageNet-1K validation dataset.

With the corresponding boundaries determined by calibration, and b being the bit-width of the target quantization. It has to be noted that in case of symmetric quantization, we can use $Z = 0$, which makes implementation of a conversion even simpler.

Besides uniform quantization, non-uniform quantization may be used. In this case, the step size of quantization levels is not constant. A non-uniform distribution of quantization levels can depend, e.g., on a logarithmic function. This example was proven by Miyashita et al. [209], who achieved a $7\times$ compression on AlexNet. Their quantization scheme also allows for a straightforward implementation in hardware. For example, Gudovskiy et al. are able to reduce the power consumption on FPGAs by a factor of four compared to an 8-bit fixed-point implementation [210]. Other more recent approaches use search methods to optimize the quantization steps for the given DNN workload [211]. Some works also investigated learnable quantization steps during the DNN training process using either iterative optimization [212] or gradient descent [213; 214]. Learning those parameters supports the minimization of quantization error, and quantization steps are automatically fine-tuned.

Although DNNs are robustness to imprecision as introduced by quantization, rigorous quantization results in a severe model accuracy degradation. Figure 3.9 presents this relation with exemplary means of a ResNet-50 [40] network[5]. The network is quantized with integers only, and the same quantization is applied to all parameters and activations. We can observe that quantization up to eight bits has only mere impact on the accuracy, which was also shown, e.g., by Gysel et al. [215]. From there

[5] The data is collected using a pretrained ResNet-50 model provided by PyTorch and using the entire ImageNet-1K dataset. Quantization is done using NVIDIA's TensorRT library.

on, the accuracy degrades heavily, dropping to about 5 % with six bits and close to 0 % with less than five bits. However, many measures to enable further quantization without such a huge accuracy loss were presented by researchers over the last decade.

A simple countermeasure to the strong accuracy impact is applying quantization at different granularity. From a hardware standpoint, quantizing all parameters and activations at the same level allows for the simplest implementation. However, it is also conceivable to quantize parts of the network with different quantization levels [216]. With this approach, parts of the network, which are sensitive to quantization can be computed with a higher precision and less sensitive parts may be quantized stronger. This unifies model accuracy and energy efficiency. Quantization granularity can be grouped along two dimensions. First, depending on the operand, e.g., quantization of parameters, activations, or both. Second, depending on the operation itself. For example, quantization can happen individually for each layer [217], for each CNN channel [218] or based on groups in the inputs. Works like DRQ even propose an individual quantization levels for regular sub-layer blocks [219]. With DRQ the authors can achieve 92 % performance gains compared to Eyeriss [60] with only 1 % accuracy loss. When various quantization levels are employed at different stages, it is commonly referred to as *mixed-precision* DNN inference [220].

Besides adjusting the quantization granularity, working with dynamic quantization is another common method to boost the model accuracy. The above-described calibration is referred to as static quantization. Dynamic quantization, in contrast, determines the clipping range during DNN training in accordance with the weights. As a result, weights are fine-tuned to yield results that are most of the time inside the clipping bounds. With this approach, works like PACT [221] or LQ-Nets [212] can quantize networks to four bits with almost negligible accuracy degradation.

Obviously, dynamic quantization requires hardware support. However, arithmetic units for variable precision are hard to realize. One method is to clock gate some parts of a larger MAC unit to eliminate the energy consumption for unused parts. This was investigated by Moons et al. [222]. Although they were able to reduce the energy consumption by 30×, extra chip area for large MAC units had to be spent. Another widely explored approach is bit-serial computation. Those accelerators operate on a stream of bits that can be of arbitrary length. Thereby, each bit is multiplied consecutively, and multiplication can simply stop when all bits are transmitted over the line. For example, Stripes [223] from 2016 presents one of the first bit-serial accelerators, with fixed 16-bit weights and arbitrary activations from 1- to 16-bit. Their

accelerator increases the energy efficiency by 57 % and only adds 32 % area to realize arbitrary precision arithmetic. Loom [224] from 2018 extends Stripes by adding support for CONV layers. In addition, their approach increases the performance by 1.87×. UNPU [225] from 2019 is another notable work for bit-serial accelerators. The authors manufactured their design into a 65 nm ASIC. Their accelerator can work with arbitrary weight precision and fixed 16-bit activations. UNPU achieves a performance of up to 50.6 T OPS/W with 1-bit weights, or 3.08 T OPS/W using 16-bit weights. In addition, implementations on FPGAs, which have outstanding performance on binary and flexible operations, were presented. BISMO [226], for example, introduces a vectorized bit-serial multiplier that can be reconfigured. On a comparably small PYNQ-Z1 board, the authors successfully demonstrate a performance of 6.5 T OPS.

Dynamic quantization also touches the training process. More accurate quantized networks can be created when it is directly trained and fine-tuned in quantized form. This is referred to Quantization Aware Training (QAT) [227; 228], while the previously described methods fall into the category of Post-Training Quantization (PTQ). Similar to the idea of determining the quantization steps during network training, QAT fine-tunes the weights and activations during training to fit better into the corresponding quantization steps. Therefore, some training iterations are performed with quantized weights and inputs. During the backward pass, gradient computation still happens using floating-point to have sufficient precision. Then during each weight update, weights are immediately quantized, which targets to minimize the quantization error. While QAT supports the accuracy of a quantized model, it obviously takes more effort compared to PTQ. Usually, multiple hundreds of fine-tuning iterations are required to restore the original accuracy. Hence, PTQ is still relevant for fast network deployment.

Beyond integer quantization, extreme quantization down to ternary or even binary weights and activation was proven beneficial for extreme low-power applications [229]. Binarized Neural Networks (BNNs) use single bit weights and activations that need much less memory. Instead of expensive matrix multiplications, BNN layers may be computed with simple XNOR and Popcount operations [230]. However, such extreme quantization leads to a much higher information loss and subsequently to a lower model accuracy. To overcome this, many techniques have been explored to make BNNs feasible. Notable are, fine-grained scaling factors for quantization [231] or other learning approaches that put more emphasis on the sharp binary quantization steps, like IR-Net [232] or RBNN [233]. Other methods include altering the model structure to work better with binarized weights and activations [234]. To benefit from

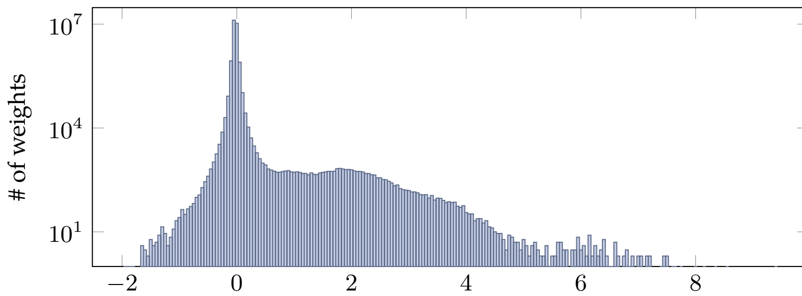


Figure 3.10: Histogram of all weights in a pretrained ResNet-50 [40] in a range of $-2.5 \ldots 10$ with 250 bins on a logarithmic scale. It can be observed that most of the weights are distributed around zero.

the low-energy consumption of BNNs, tools like FINN [235] were presented to allow for a straightforward implementation on FPGAs. Today, BNNs can cover nearly the same scope as traditional DNNs. They are especially interesting for very constrained platforms that can tolerate the associated model accuracy drop.

Quantization presents a great opportunity to reduce the memory footprint and computational complexity of a DNN. As a result, the topic was intensively researched in the recent decade and several approaches were presented to lessen the accuracy degradation, improve the computational performance or to enable fast deployment of quantized DNNs. As we will see in the next section, however, the hardware support for quantization should not be neglected.

3.3.3 Pruning

Pruning is an important technique to optimize DNN inference and training towards a faster and more energy-efficient execution [161]. It removes redundant or insignificant parameters or activations during inference or training [236]. To understand the basic concept of pruning, we can take a look at Figure 3.10, which shows the distribution of weights in a pretrained[6] ResNet-50 [40]. We can observe that a large share of weights is actually zero, or at least close to zero. This is again due to the over-parametrization of DNNs. Since the predominant operations in DNN inference are multiplications,

[6] https://pytorch.org/vision/main/models/generated/torchvision.models.resnet50.html weights V2 are used

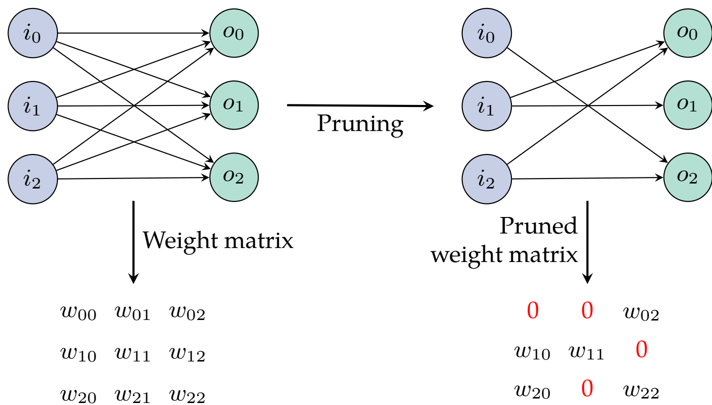


Figure 3.11: Weight pruning shown as graph and weight matrix representation.

we can eliminate those with weights that are zero. This pruning method referred to as weight or connection pruning, i.e., pruning removes unnecessary connections between neurons and makes the weight matrix sparse, as shown in Figure 3.11.

Similar to quantization, pruning approaches can be classified according to the operand they are targeting and the granularity. The granularity of sparsity that can be achieved by pruning is an important design parameter, as it highly impacts the efficiency and design of the network topology and accelerator architecture. This again underpins the aforementioned statement, that co-designing algorithm and accelerator is crucial. The simple example in Figure 3.11 shows unstructured weight pruning. The resulting unstructured sparsity allows for a maximal exploitation of zero weights and subsequently a high reduction in computations [237].

However, efficient storage of irregular sparse matrices requires extra information that represent the location of non-sparse elements. Common methods store non-sparse elements using Compressed Sparse Column (CSC), Compressed Sparse Row (CSR) or run-length encoding formats [238]. Adding this information can lead to a doubling in storage requirements for networks like AlexNet [239]. To circumvent this extra memory usage, numerous research was carried out on structured pruning. Therefore, pruning may be applied to individual convolution filters [240] or to entire feature maps [241], reducing, e.g., computational cost by 38 % on ResNet-110 or by more than 70 % on a network for CIFAR-10, respectively.

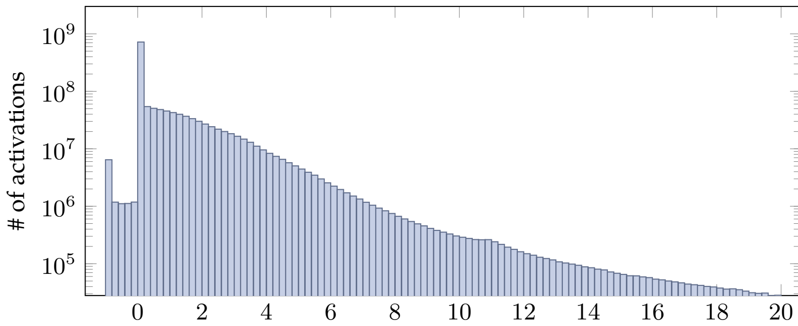


Figure 3.12: Activations of ResNet-50 [40] using the ImageNet-1K validation dataset. Shown as histogram in a range of $-1 \ldots 20$ with 108 bins on a logarithmic scale. Due to the ReLU activation function, most intermediate results are zero or small.

Sparse weights usually occur in irregular patterns across layers, filters, and kernels. Pruning, i.e, skipping computations with those sparse weights, results in no different network output and is consequently the most straightforward approach [242]. In contrast, magnitude pruning [243] assigns each weight an importance to the model output, on which we can decide with a threshold if it is pruned or not. This process is also called sparsification and leverages again the robustness of DNNs to imprecision. For example, ThiNet [244] aims to find superfluous convolution filters and prunes them, achieving a 16× compression of VGG-16 with only 0.52 % Top-5 accuracy loss. To increase sparsity during training other metrics were proposed as well. Lin et al. [245] use Markov chains to identify sparse activations, achieving a 10× inference speed-up. RL methods were investigated by Chen et al. [246]. They optimize for efficient usage of the available memory using pruning and demonstrate a performance improvement over static pruning. Metrics like Markov chains [245] To increase the amount of sparse weights in a DNN, sparse training was proposed [247]. In contrast to the previously described methods, this approach adds sparsity as another metric to the training loop. For example, Sun et al. [248] propose and train a CNN for face recognition that achieves a 33 % better accuracy and has 88 % fewer parameters than state-of-the-art models.

So far, we have covered pruning and sparsification methods for weights in DNNs. However, modern approaches like NAS combined with iterative pruning already yield very compact and dense network topologies [249]. While networks from a decade

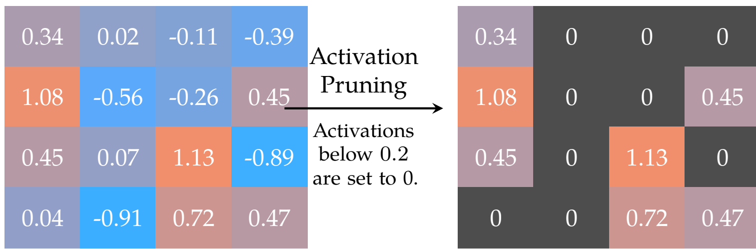


Figure 3.13: Basic principle of magnitude activation pruning.

ago could easily be pruned to 90 % sparsity [250], modern networks would suffer from huge information loss. But pruning is still a highly relevant topic since it can also be applied to activations. Most network topologies until recently mostly relied on the ReLU activation function. Besides its simple computation, ReLU has another useful characteristic, which is that all negative activations become zero, i.e., sparse values in the output feature map [27]. As reference, Figure 3.12 shows a magnitude distribution of input feature maps in ResNet-50[7]. Note the peak in the bin that covers the range from 0 to 0.2. Counting only the exact zero activations during inference, we can observe that in this example 48.05 % of all activations are zero. Activation pruning leverages them in the next layer, because again one operand of a multiplication is zero. Hence, the computation can be skipped analog to weight pruning. Since, especially in CNNs feature maps are huge, activation pruning can significantly improve the energy efficiency [251] or reduce the memory footprint [252].

Yet, two challenges reside in activation pruning. First, the location of sparse values is not fixed, but strongly depends on the network input, second activation sparsity is not present in any regular form. Pruning irregular forms of sparsity during inference can draw disproportionately from the energy budget, since the hardware architecture has to decide on the fly, which operation can be skipped [253]. As a result, for example, a sparse AlexNet is 25 % slower on an off-the-shelf GPU [254]. To increase the amount of sparse activations and their regularity, similar to weight pruning, a threshold can be applied as shown in Figure 3.13. For example, Reagen et al. [255] first added a hand-picked threshold to increase the activation sparsity of a three layer MLP and can reduce the energy consumption by 50 %.

[7] The data is retrieved using a pretrained ResNet-50 network provided by PyTorch and the entire ImageNet-1K validation dataset

Besides all the mentioned challenges, activation pruning combined with NAS offers enormous potential to make DNN inference more efficient, in terms of energy, latency and throughput. For example, Eyeriss from 2017 [60] is one of the first architectures to support activation pruning, by power-gating individual PEs, saving 45 % of the PE power. While just skipping MAC operations is sometimes not worth the extra logic to identify sparse activations, compressing pruned activations for optimized memory usage pays off much faster, especially for energy efficiency. Looking, for example, at the energy required for an 32-bit floating-point MAC operation in comparison to loading 32 bit from external memory (see Figure 2.21) we see that memory movement is 173× more energy intensive. Considering the vast intermediate feature maps of CNNs, selective activation pruning can cut the energy consumption drastically. For example, Alwani et al. [256] can reduce the required memory for intermediate result by 95 % in a VGGNet.

As stated before, hardware support for different pruning mechanisms is key to a benefit from the favored energy savings. Hence, over the last few years, many accelerators, which implement various approaches to pruning, were presented. SIMGA [257], Extensor [258] and Gamma [259] are examples for accelerators that support sparse General Matrix Multiply (GEMM) operations. SIGMA, for instance, can save up to 3× energy compared to a TPU [7]. Gamma improves the performance further and outperforms other sparse GEMM accelerators by an average of 2.1× and reduces memory expensive transfers by up to 13×.

Looking at DNN workloads, Zena [260] applies skipping of operations to both weights and activations, showing up to 9× speed-up compared to Eyeriss [60]. Cnvlutin [261], introduces a new co-designed data encoding scheme to benefit even more from compression and allow for parallel pruning. With their sparse row compression method, they show a 1.37× performance increase by removing operations with zero compared to by then state-of-the-art accelerators. Moreover, they report 44 % sparsity in a wide set of DNN algorithms. Compression through arranging dense values in input matrices using a stride has been explored by Anwar et al. [241]. With their method, they can prune 70 % of a network for the CIFAR-10 dataset with only slight accuracy degradation. The Cambricon series from 2016 and 2018 is another notable example of sparse DNN accelerators. Cambricon-X [262] aims to prune irregular weight sparsity, and the successor Cambricon-S [263] also includes a co-design approach that groups sparse activations into regular patterns, resulting in overhead reduction for encoding mechanisms. The latter work can achieve a 6.43× better energy efficiency compared to the accelerator DianNao [139]. Also, the authors

of Eyeriss presented in 2019 a successor of their accelerator. Eyeriss v2 [126] adds a hierarchical mesh with PEs that can operate directly on compressed data format to reduce the logic overhead for decoding data. They improve their initial work by 2.5× and 12.6× in terms of energy efficiency and speed, respectively.

Another notable and quite recent sparse DNN accelerator is Flexagon [264]. Their reconfigurable accelerator can adjust to the sparsity and workload that is current present to switch to the dataflow that has the best efficiency. With their dataflow, the authors report an 18 % better performance to area efficiency compared to Gamma [259] over eight evaluated DNN workloads. In addition to ASICs, some works focus on accelerating sparse workloads on FPGAs. For example, Niu et al. [265] are able to reduce data offloading through compression in VGG-16 by 42 %, resulting in an efficient utilization of the available DSPs. Besides accelerators for DNN inference, some works investigated sparse DNN training. For example, Procrustes [266] can shrink the energy for training ResNet-18 and MobileNet-v2 by 3.26× or 2.39×, respectively. Also, NVIDIA's Ampere micro-architecture supports pruning of blocks consisting of four values with up to 50 % sparsity [121].

As we have seen in this section, pruning is a very well-implemented method to make DNN workload more energy-efficient. With the proper hardware support, especially activation pruning can drastically reduce the memory footprint of a CNN and the associated energy-expensive memory movements. As a result, modern embedded platforms for efficient DNN inference require not only sophisticated DNN architectures, but also support for online activation pruning.

3.3.4 Combined works

Most works, even some presented before, combine pruning, quantization and often compression algorithms to maximize the energy efficiency. These methods may be combined with sophisticated HW-NAS methods, leading to an end-to-end design process, as shown in Figure 3.14. First, a DNN workload is optimized using quantization and pruning. Once the parameters are sufficiently reduced, we can start the accelerator tuning. Here hardware parameters are adjusted to optimize Performance, Power and Area (PPA) figures towards the design objectives.

One of the first works that introduced a careful combination of pruning, quantization and compression is the very influential work by Han et al. [239]. Using unstructured magnitude-based pruning, arbitrary precision quantization and Huffman coding to

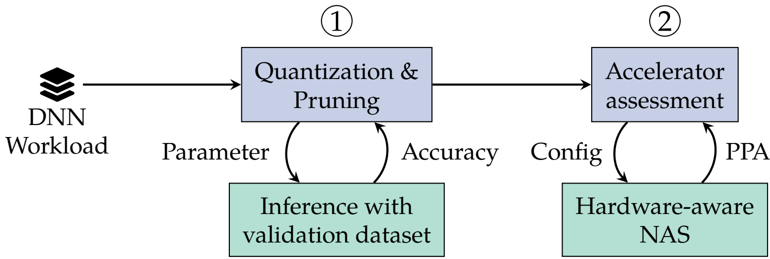


Figure 3.14: Principle and steps of Design Space Exploration for DNN hardware accelerators. First the workload itself is optimized using pruning and quantization. Then the hardware configuration is iteratively optimized.

compress the data, the authors can reduce the memory requirements significantly. On an AlexNet and a VGG-16 workload using ImageNet the memory footprint shrinks by 35× and 49×, respectively, without accuracy drop. This allows a model to fit into an on-chip memory, which drastically reduces energy consumption, and allows for inference on mobile devices. Based on their approach, the author presented their accelerator EIE [238] a year after. By combining pruning, CSC compression, sparse matrix-vector multiplications, and weight sharing, they demonstrate a 189× and 13× faster inference on CPU and GPU, respectively, compared to the same uncompressed DNN. Chen et al. [267] present a work that compresses parameters of a DNN using Tunstall encoding and argue that it can be implemented in a straightforward manner. To prove this, they present hardware implementations for fast decoders on an FPGA and demonstrate a 6.23× speed-up compared to Huffman coding on ResNet-50, while consuming significantly fewer resources.

Besides accelerator design, many works investigated how to apply quantization and pruning at the same time. CLIP-Q [268] proposes such an approach. The authors show an improvement of state-of-the-art quantization or pruning works on a wide range of CNNs. Their approach can also, compress already efficiently designed networks like MobileNet by 7.6×. The work by Yang et al. [269] combines all above-mentioned techniques in an automated framework that removes the need for setting compression factors for each layer manually. With their tool flow, they can compress AlexNet trained on ImageNet by 205× without accuracy loss. HFPQ [270] moreover includes hardware aspects into the compression pipeline. They first prune the network, retrain it, and finally quantize it. By considering the number of Floating-Point Operations (FLOPs) while compression, they can, for example, compress ResNet-50 by 15×,

reduce the number of FLOPs by 72.6 % and only drop the accuracy by about 3 %, which outperforms other state-of-the-art works. More recently, Krieger et al. [271] looked at combined quantization and pruning for constrained embedded systems. Their approach involves a RL agent, which tries to minimize the inference latency measured on an actual device. Using a ResNet-18 to classify CIFAR-10, they can reduce the inference latency by $5\times$, without notable accuracy loss.

3.4 Summary, Challenges and Contributions to the State of the Art

In this chapter we have found that a large body of research exists around the topic of DNNs. Since modern DNN workloads are growing quickly in computational complexity, they pose a huge challenge on the target computing hardware. Small grayscale images showing only single characters eventually became high-resolution camera images from an autonomous vehicle and recognition of ten different characters became recognition of pedestrians along the street, which has to be done very accurate, quick and reliable. As of the shelf hardware cannot fulfill the computational demands, many people stated to search for an answer to the growing computational demand. Thus, over the past two decades, a vast range of different accelerators, tools and optimization techniques have been proposed.

Today, modern DNNs are highly optimized for a given application using methods like NAS, knowledge distillation or parameter sharing. To execute those workloads efficiently on underlying hardware structures, it is crucial to have sophisticated mappers at hand. For strongly constrained environments like embedded systems, it is also recommended to use HW-NAS or co-design the algorithm with the accelerator. This allows to create tailored accelerators and DNNs that can work seamlessly together. Finally, one can consider pruning, quantization and compression to leverage two important aspects of DNNs: their over-parametrized nature and the characteristic that they tolerate imprecision during inference. With all these methods in place, applications like environment perception on autonomous vehicles, real-time face-detection on smartphones or large language models for chatbots became feasible.

However, there are still some unanswered questions and challenges ahead. The design of a DNN, its deployment, its optimization, and the underlying hardware accelerator, affect the performance, latency, energy consumption and model accuracy. As a result,

the search space of different parameter combinations is infinitely large, and no single configuration fits it all. Besides the algorithmic challenges themselves, which are not covered in this thesis, there is still a myriad of problems to solve considering hardware platforms, architectures, tools, and memories. Regarding the current state-of-the-art, we have identified some major open challenges for efficient deployment of DNNs to constrained embedded systems:

1. How does a holistic simulation platform look like that can estimate the energy, latency, and performance impact of the various design parameters? And can we design such a platform to automatically explore all design opportunities to get the best DNN accelerator design for a given workload?
2. Once we have an efficient DNN accelerator, can we optimize it further using regular pruning? And what forms of regular pruning can be implemented with only small hardware overhead and still yield a high utilization of sparsity?
3. How sensitive is a DNN which benefits from online pruning and quantization to weather perturbations like rain, fog, or snow? And how can we mitigate the impact of such weather conditions at run-time to have an efficient and at the same time safe DNN inference?

Of course several aspects of these questions to varying degrees have been addressed by researchers. To answer the first question, simulation tools and HW-NAS frameworks were introduced. For instance, MAGNet [272] helps to explore the impact of different systolic array design configurations on various performance figures. This concept was further extended by works like SMAUG [273], STONNE [274] and the subsequent Flexagon [264]. These works feature end-to-end simulation for more versatile DNN accelerator structures. They even consider various scheduling options and come with their own mapping and tiling tools. As a result, their approaches reveal extensive information to guide the design process. In addition, significant research has been carried out in the field of FPGAs. The works by Motamedi et al. [275] and by Abdelfattah et al. [276] are popular examples. They introduced NAS and co-design techniques to design DNN accelerators tailored for FPGAs. This is achieved by assessing precise area and energy estimates obtained from High Level Synthesis (HLS).

All the studies mentioned share a common feature: they perform cycle-accurate evaluations. Although this method is time-consuming, it provides the highest level of accuracy. However, some works looked into approximate and analytical models to

speed the simulation up. For example, SCALE-Sim [277] which estimates bandwidth and inference cycles for systolic arrays. Timeloop [163] takes this a step further by extending the estimations to a wider range of DNN accelerators, and includes tools for estimating energy and area. In addition, works like MAESTRO [191] and ZigZag [278] made large progress in the richness of mapping problems and simulation speed.

Beyond the works mentioned in this brief overview, we will investigate the first question and the associated state of the art further in Chapter 4. Therefore, we introduce FLECSim, a platform for end-to-end cycle-accurate SoC simulation that helps to evaluate and test a large range of configurable accelerator designs in terms of their energy and area efficiency. The chapter will also present an analytical approach based on the systolic array generator Gemmini. Our analytical approach considers all layers in a DNN, yielding faster and more exact estimates. In addition, a real-world use case is demonstrated. Here, an embedded FPGA design is explored to allow for an energy-efficient inference of a face recognition algorithm.

Regarding the second research question, some works have thoroughly looked into leveraging activation sparsity. Important state-of-the-art examples comprise, for example, SCNN [151]. The authors use an efficient and novel index-based compression method to save energy and reduce the number of computations. In their hardware implementation they can report an up to 2.3× better energy efficiency. Cambricon-S [263] aims to increase the performance further using coarse-grained pruning in the shape of contiguous blocks. Since feature maps have often different degrees of sparsity, STICKER [279] proposed to use nine different compression modes for efficient inference, which balances high exploitation of sparsity and model accuracy. Their work achieves a 1.8× energy efficiency improvement over the state of the art. To make compression and decompression faster, the authors of SNAP [280] introduced a novel parallel associative search to identify non-zero operands in unstructured input data. This helps to reduce memory accesses by 22× and yields an efficiency of 3.61 TOPS/W.

To extend this body of research further, we will look in Chapter 5 into various design options for activation sparsity. This contribution discusses various methods of pruning in detail and why regular sparse blocks that match the dimensions of the hardware accelerator can be pruned efficiently. To bring this into effect, we introduce Spex and Sparse-Blox, a tool to increase this kind of regular sparsity and a hardware extension to prune sparsity during inference.

Considering the third research question, some research already explored hardware accelerators that support mixed-precision instead of fixed bit-width quantization. HAQ [281] explores quantization using RL and demonstrated that mixed-precision can significantly improve the energy efficiency. A more accurate quantization scheme can be achieved using GA, as shown by AnaCoNGA [282], which uses HAQ as baseline. To make support for mixed-precision less area expensive, BitBlade [283] uses bitwise summation and a modern tiling scheme to increase the utilization. The efficiency and model accuracy were further improved by the structured, dynamic and fine-granular quantization approach presented by Huang et al. [284]. This recent work underscores the potential of mixed-precision quantization for better accelerator efficiency.

Moreover, a range of studies have investigated how quantization impacts the robustness of DNNs. For instance, Wijayanto et al. [285] introduced a compression method for classification tasks that also takes into account the model error rate. Khalid et al. [286] examined in particular adversarial attacks on CNNs and used quantization to enhance the robustness against these attacks by inserting quantization layers at the input. By focusing on outliers during training and quantization, Zhao et al. [287] can improve the robustness of various DNN models compared to other robustness-aware quantization schemes. Besides these works, the robustness against different weather conditions was addressed by models that, e.g., remove rain, such as Semi-MoreGAN [288] or the work by Wang et al. [289].

A combination of mixed-precision inference on hardware combined with robustness evaluation, which is currently missing the state of the art, is discussed in Chapter 6. This chapter analyzes the impact of quantization applied to common CNN workloads for image segmentation in the automotive domain regarding energy efficiency and robustness.

To foster further research the work that has been carried out in the last two chapters is made available as open-source to the public[8].

[8] https://github.com/itiv-kit/dnn-model-exploration

Chapter 4

Algorithm and Hardware Accelerator Co-Design for Efficient Hardware Accelerator Design

Already while discussing the background of Deep Neural Network (DNN) acceleration and the current state of the art it immediately becomes clear that there is a myriad of different design parameters that influence various metrics of the accelerator. This wide range of available accelerators, dataflows and memory management strategies highlights that there is no single accelerator design that fits all DNN models and applications. Of course, some design configurations can cover a larger range than others. But sometimes, we also need a very efficient and tailored accelerator for a particular application.

The contribution that is covered in this chapter helps to explore DNN accelerator design options and systematically assess them regarding the models that it runs. Therefore, we introduce FLECSim [Hot+21], a simulation environment that helps to estimate the efficiency of a DNN accelerator accurately. It uses a cycle-accurate SystemC simulation and contains modules for Central Processing Units (CPUs) and memories, allowing for fast integration and evaluation of new DNN accelerators. To underline the flexibility of the simulation tool, we perform a Design Space Exploration (DSE) of a Convolutional Neural Network (CNN) accelerator running on a highly configurable embedded Field Programmable Gate Array (eFPGA) [Hot+22b; Hot+20]. Since entire cycle-accurate simulation can be time-consuming, we furthermore investigate ways of analytical cycle and energy estimation [Hot+23c], since the dataflow of common DNN accelerators is very homogenous and predictable.

4.1 Overview, Introduction and Motivation

The huge advancement in Deep Learning and subsequently DNNs was only made possible by tremendous effort on accelerators to train and execute these workloads. As a consequence, today there is an array of different DNN accelerators available. Most of them distinguish themselves by incorporating various dataflows, memory sizes, data orchestrations, arrangement of computing nodes or interconnectivity, just to name a small set of possible design parameters. Besides that, there are accelerators that bank on optimization strategies that we discussed in Chapter 3 like pruning or quantization. Although pruning shows great potential for making DNN accelerators more energy-efficient, it is currently mostly investigated in research accelerators. Modern commercial accelerators on the other side usually feature quantization to have both energy efficiency and a high degree of flexibility for all kinds of DNN workloads. Besides the bare energy efficiency, considerations like real-time capability or safety requirements add another dimension of complexity to the question of *how does the best DNN accelerator for a given application looks like*.

One generally strives to simultaneously optimize for high throughput, low power consumption, a small area and so on. Obviously, a higher throughput may be realized by adding more Processing Elements (PEs) or a more performant memory interface. However, this affects energy consumption and chip area. Besides that, the DNN accelerator design strongly depends on the DNN topology, which is later runs on this accelerator. An Output Stationary (OS) dataflow might work well for one application, while another benefits from Input Stationary (IS) orchestration. As a consequence, different design parameters have substantial impact on the performance metrics.

As we can see, various applications pose different challenges and requirements to the underlying DNN accelerator. For example, an embedded accelerator may emphasize a low power consumption and real-time capable latency, while data-center accelerators require high throughput instead. Hence, it is important to consider the different adjustments one can make to an accelerator during the system design phase to meet all design and performance requirements. To achieve this, a tool is needed to systematically observe the outcome of small changes in the accelerator's configuration. In particular, this tool should be utilized as early in the design process as possible to save cost. Additionally, it should deliver fast and accurate results for an automated exploration of the large design space that opens up when considering the vast number of design parameters.

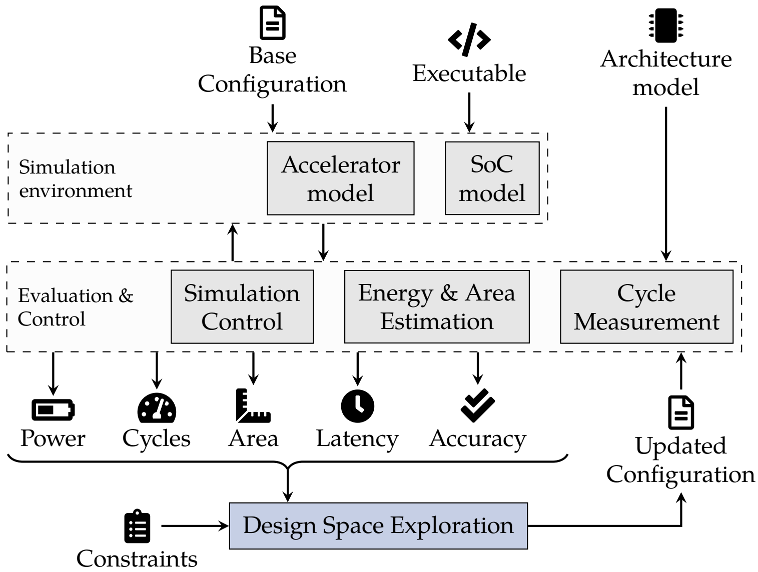


Figure 4.1: Basic concept of our simulation platform. An accelerator and SoC model represents the later chip architecture. During simulation of a DNN workload, various metrics, like energy, cycles and area are collected and then used for an automated design space exploration.

An appealing concept to build such a tool is to use a simulation environment, in which various parameters can be changed and then the impact on the accelerator is observed. Using common hardware simulation tools like Register Transfer Level (RTL) simulation, moreover, enables to check functional correctness of a design. However, an entire RTL simulation may run multiple days even for medium-sized DNN workloads and accelerators. Instead, a more approximate form of simulation is worth looking at.

Over the course of this chapter we will sketch such a simulation platform that fulfills all mentioned requirements. A basic concept is shown in Figure 4.1. It consists chiefly of a simulation environment and modules for evaluation and simulation control. The simulation environment itself has to feature a representation of the accelerator and all remaining parts of the System-on-Chip (SoC). During simulation, each component reports how they effect the design metrics. Those metrics may encompass performance aspects like latency, number of cycles per inference or efficiency aspects like

power and area, which can be computed using an architecture model. Once pruning, compression or quantization are included, model accuracy becomes an additional design metric. The entire simulation platform may be extended by a DSE for an automated architecture optimization. This component takes all design metrics and assesses them in accordance to a set of user constraints and design requirements. Based on this assessment, the base configuration is updated and fed back into the simulation platform to generate an updated set of design metrics. The DSE should happen in a goal oriented way, e.g., using an optimization heuristic that tunes design parameters towards the user goals.

In the following of this chapter, we will now fill this high level concept with actually implemented components. Therefore, we discuss two kinds of simulators: A very precise cycle-accurate but also relatively slower platform and an analytical model that offers a slightly less precise performance estimation but runs orders of magnitude faster. First, we will dive into FLECSim, our cycle-accurate simulation framework. Then, we have a detailed look at analytical forms of behavior modelling of a DNN accelerator. When considering combining both ways of simulation, we can also combine the best of both worlds: In the first step, we can narrow the large design space down using our fast analytical evaluation and then have a look at the remaining design parameters using FLECSim.

4.2 Related Work

Over the last decades supervised Machine Learning (ML) approaches like CNNs, Recurrent Neural Networks (RNNs) and Transformer networks have established themselves in an increasing number of applications, as they are able to solve more complex tasks. However, to do this, also the computational complexity of the model grew heavily, especially when looking at the large number of operations in modern CNNs for image processing tasks. Although, research has yielded more efficient model implementations such as EfficientNet-v2 [42], the hardware accelerators that execute them have to keep pace. As a consequence, many highly performant DNN hardware accelerators were presented by numerous research groups, most of which were covered in Chapter 3. The many flavors of DNN accelerators that exist today, highlight the fact that there is not a single accelerator that works for all kinds of DNN models, but that its design requirements depend on the application and domain. In response to that, many tools, simulation frameworks and especially co-design

approaches to assess DNN accelerators were introduced by research teams in past couple years. We will now have a look at this body of research to see what questions in this field are still left unanswered.

In general, we can classify the current state of the art in accelerator algorithm co-design based on how specifically they are tuned for a given application or target platform. Usually, the more specific a tool is, the more accurate it is, but it also becomes less flexible, e.g., to support different dataflows or various accelerator dimensions. Besides that we can also distinguish tools that aim for end-to-end and hardware-aware Neural Architecture Search (NAS) and those that predict performance characteristics of the accelerator.

Some prominent hardware-aware NAS and DSE tools, were already presented in Subsection 3.2.3. In addition, we would like to introduce some more HW-NAS approaches that target efficient design of DNN accelerators. Yang et al. [290], for example, are the first to investigate the impact of Network-on-Chips (NoCs) in distributed DNN accelerators. Therefore, they couple a NoC search with traditional NAS for the network topology. For their evaluation, they looked at 1×1 to 3×3 clusters of DNN accelerator tiles, showing a significant throughput increase over standalone a NAS. On CIFAR-100 their found network architecture shows a 2× to 4× improvement compared to FBNet [185]. Since, they look at comparably small NoC sizes and analyze only the system latency and not the energy and area figures, which are also crucial design parameters, they continued their research shortly after and presented a co-exploration approach for heterogeneous Application-Specific Integrated Circuits (ASICs) [291]. This work also includes assessment of energy and area besides latency. The corresponding numbers are taken directly from ASIC synthesis results. For evaluation, they investigated NVDLA and ShiDianNao, two distinct DNN accelerators with CIFAR-10 and CIFAR-100 datasets, and ResNet-9 as model. With their NAS method that immediately targets ASICs, they generated architectures that offer twice the energy efficiency and half the area compared to separated ASIC design and NAS steps. Thereby, they report only minor accuracy degradation. However, their approach still requires a fairly long exploration time.

The recently presented NAS approach aw_nas by Ning et al. [292] delivers an extensible framework for various platforms. Therefore, their open-source tool is build in modular way and covers a wide range of datasets and search parameters. To integrate hardware cost models, aw_nas offers a method to profile the target hardware using a profiling network. Each element of this network is evaluated directly on an actual

target platform. With all this information, the tool sets up a database. They verify their energy and latency estimation using Field-Programmable Gate Arrays (FPGAs), CPUs and Graphics Processing Units (GPUs) and with three different DNN layers. Although this approach requires much time for the initial database setup, it promises a fast evaluation of the subsequently proposed models.

When focusing on the impact of architectural factors a large body of research has emerged around simulation of hardware structures especially for DNN inference. Simulation tools for hardware are generally widely used to verify the behavior of RTL designs. However, such tools require exact RTL code, which adds a lot of additional complexity and overhead for SoC or DNN accelerator designs that can often reuse large portions of already established designs like CPUs or memories. One of the first approach to make simulation of large accelerator designs simpler is PARADE by Cong et al. [293]. Even though their tool does not only focus on DNN accelerators, it reuses, for instance, communication infrastructure, caches or large x86 out-of-order CPU cores to quickly and accurately model entire SoC designs.

As DNN topologies become more complex and hit a power wall, more tools are being introduced to assess and model DNN accelerators [294]. For example, MAGNet by Venkatesan et al. [272] aims to explore the impact of various design parameters, like number of PEs, sizes of memories or arithmetic precision, on the Performance, Power and Area (PPA) of a systolic array. Therefore, they first introduce their architecture and a dedicated mapper to explore various dataflows. Then they optimize found design configurations using Bayesian optimization. PPA figures are generated using synthesis from a configurable SystemC model. Across a large set of architecture configurations and DNN models, including ResNet-50 and AlexNet, they generated an architecture that delivers 2.8 T OPS/mm^2 with 4-bit weights and activations. However, this design sacrifices about 18 % accuracy. A more conservative configuration still delivers 2.3 T OPS/mm^2 with and less than 1 % accuracy loss.

The authors of SMAUG [273] followed a similar approach. Their end-to-end simulation framework helps to observe how power, accuracy and performance are affected by various scheduling and architecture considerations. To model the DNN accelerator and its peripheral components they use the gem5 [295] simulation framework. As accelerator, they focus on systolic arrays, modelled in gem5. Moreover, their tool supports RTL designs using Aladdin [193]. With all components in place, SMAUG optimizes and preprocesses a DNN workload and performs a mapping and tiling. The actual hardware simulation happens in gem5. During that, energy and performance

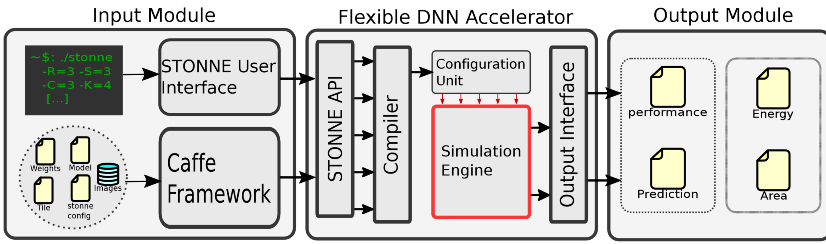


Figure 4.2: Overview of the cycle-accurate DNN accelerator evaluation framework STONNE proposed in [274].

figures are generated by Aladdin and DRAMPower, which allows for many interesting insights about the workload and the accelerator. With their tool, the authors demonstrate an up to 5× performance increase through fine-tuning interfaces and the software stack. However, simulation of a ResNet-50 with ImageNet takes about five hours, which makes the tool barely feasible for large scale DSE.

To address this issue, STONNE by Muñoz-Martínez et al. [274] extends SMAUG. At first glimpse, it also aims to accurately model the behavior of a DNN accelerator. Therefore, they focus on MAERI [153] as accelerator, which is highly configurable. But they are not limited to only that particular architecture. In contrast to SMAUG, they claim that their tool offers more degrees of freedom and allows for faster evaluation of individual designs. Their framework is shown in Figure 4.2. Taking a network configuration directly from Caffe, a ML framework, they can configure their architecture model and estimate performance and accuracy. Similar to the aforementioned tools, STONNE is also equipped with a large database of reusable building blocks, such as memories. STONNE is evaluated with five different DNN workloads such as ResNet-50 or SqueezeNet, showing its flexibility and that the cycle prediction is very close to a baseline accelerator implementation in Verilog. For all workloads, the authors report interesting findings like accelerator utilization. Moreover, since it is open sourced, subsequent works, e.g., Flexagon [264], use STONNE as cycle-accurate microarchitecture simulator. However, the authors do not report the time required for the cycle-accurate DNN evaluation and by now they have not included energy and area estimation.

One of the largest drawbacks of the up until now mentioned simulation tools is the long evaluation time. This is due to cycle-accurate execution or very demanding RTL simulation, which of course has the highest precision but also the longest run

time. As a consequence, some researchers looked into approximate or analytical models to make the assessment of various design parameters much faster, allowing for large-scale DSEs. SCALE-Sim by Samajdar et al. [277] is such a simulation tool that enables fast cycle and bandwidth estimation of DNN accelerators. To decrease the search space it limits itself only to systolic arrays, which also increases the simulation accuracy as only one accelerator model has to be accounted for. Using a configuration interface, different array and memory sizes can be defined. The memory hierarchy inside SCALE-Sim consists of two input memories for activations and weights, and a separate memory to store results. SCALE-Sim supports all common dataflows for systolic arrays: Weight Stationary (WS), IS and OS. To evaluate the impact to different mappings, SCALE-Sim also offers a mapper, which automatically determines the best one for a given workload. The simulator works on individual layers and runs each workload in a cycle-accurate fashion, but lacks RTL simulation. Hence, SCALE-Sim tries to settle between very accurate RTL simulation and fast analytical execution, bringing the evaluation time down to the order of minutes not hours. The authors evaluate their tool with a large range of different workloads and show design insights, like which dataflow suits best or required memory bandwidths. However, SCALE-Sim focuses mainly on Multiply-Accumulate (MAC) operations and neglects, e.g., pooling, which also contributes to the overall cycle-count. Moreover, it does not report the accelerator's area requirements.

Timeloop by Parashar et al. [163] is another very popular tool to perform analytical modelling of a wide range of DNN architectures. The flexible tool enables to model the underlying architecture through a set of primitives. To allow for systematic assessment of different accelerators, Timeloop has two main components: a mapper and an estimator for energy, area and performance. As inputs for those modules, Timeloop takes the workload description of a single layer and an architecture description, which fixes some components like the Dynamic Random Access Memory (DRAM) size or the number of PEs. The integrated mapper then explores the mapspace automatically and yields for each configuration corresponding PPA figures. During mapping, action counts are generated, which are then converted by Accelergy [169] into energy and area estimates. Although, Timeloop can represent all different kinds of DNN workloads, it still does not take operations for pooling and activation into account. Moreover, the mapspace has to be described in advance, which requires knowledge about the dataflow, arrangement and number of PEs and so on. Hence, it does not provide a layer to perform a DSE, but is limited to assessment of individual accelerator configurations.

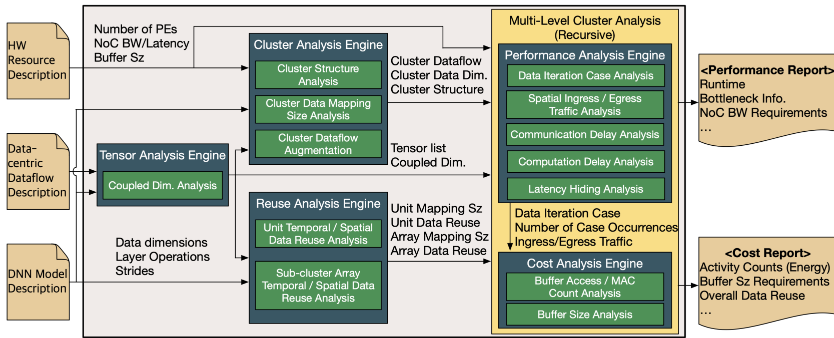


Figure 4.3: Overview of the DNN analysis tool MAESTRO presented in [191].

Similar to Timeloop, Eyexam [126] introduces a custom simulation model that accurately reflects characteristics of the DNN workload and the underlying accelerator. The tool does not try to optimize all factors at once as most NAS approaches do, but works in a step-by-step fashion. Thereby, each architectural decision in each step is annotated with the associated performance impact. Eyexam aims to maximize the number of active and utilized PEs. To measure the performance impact of each optimization decision, it uses the widely-established Roofline model. Each step moves the ratio of operational intensity to performance in the Roofline model closer to an optimal accelerator. By using the Roofline model, Eyexam can estimate the PPA figures of DNN workloads and various accelerators very quickly and enables fast insight generation, e.g., to identify performance bottlenecks. However, such an analytical approach usually remains effective only for the specific accelerator it was designed for.

A more versatile accelerator assessment tool is MAESTRO by Kwon et al. [191]. In their work, they introduce a formal description for DNN workloads and dataflows. Based on this, they then present their analysis framework MAESTRO, which is shown in Figure 4.3. The tool breaks large DNNs down and maps them onto a set of PEs. A huge range of dataflows can be analyzed in the next step using cluster analysis. The resulting numbers are translated into accelerator performance in terms of inference cycle count. Energy and area figures are either generated by their own energy model, or, similar to other works, using Accelergy and CACTI. The accuracy of their analytical model is benchmarked against RTL simulations of MAERI [153] and Eyeriss [296], showing a mere 3.9 % absolute error. In two case studies the authors demonstrate the capabilities

of their tool. Overall such an analytical model allows for rapid evaluation of designs. In particular, MAESTRO can evaluate up to 0.17 million designs per second, about 1000 to 4000× faster than RTL simulation, allowing for comprehensive and extensive DSE. This is demonstrated in a subsequent work, ConfuciuX by Kao et al. [178], which integrates MAESTRO into a DSE process guided by Reinforcement Learning (RL) and a Genetic Algorithm (GA) for fine-tuning. ConfuciuX can evaluate a large set of DNN workloads and accelerators, and yields optimal design configurations quickly. Using various benchmark workloads and objectives they can accelerate the design space exploration by up to 24×.

Another recent tool to analyze the performance and efficiency of DNN accelerators is Interstellar by Yang et al. [297]. It not only focuses on CNNs, but also supports Multilayer Perceptrons (MLPs) and Long-Short Term Memories (LSTMs). To represent all kinds of loop nesting and dataflow needed for almost all DNN operations, they use Halide's scheduling language and built a tool that transforms it via an intermediate representation into microcode that is executed on the corresponding DNN accelerator. With their approach they demonstrate an energy efficiency improvement of up to 4.2× for MobileNet and up to 1.6× for an LSTM network.

However, MAESTRO and Interstellar still have some design limitations when it comes to the hardware design space and the kinds of supported mappings. To address this issue, shortly after Mei et al. introduced ZigZag [278]. Their DSE framework claims to explore the design space even faster and thus can overcome the aforementioned design limitations. In particular, they can cover all mappings by supporting uneven mappings with various memory hierarchies. According to their findings, uneven mappings, which refers to mapping input and output activations to different levels in the memory hierarchy, enhances energy efficiency by up to 20 %. Their latency-optimized analytical hardware cost estimator takes energy and area into account by counting actions and summing up the area cost of each small component, similar to Accelergy. Performance and cycle count are estimated using a sophisticated set of equations that accumulate the cycles for each loop of a convolution operation. The mapping tool, then tries to optimize the mapping based on this set of equations. Finally, ZigZag can generate a memory hierarchy that serves an energy-efficient inference. In three case studies they evaluate their tool flow and present mappings that are up to 33 % more energy-efficient than the mappings generated by state-of-the-art works like Timeloop or Interstellar.

Modelling and simulation did not stop at generic DNN accelerator structures. To make the exploration faster and more feasible, shrinking the search space is a great method. For example, when only focusing on FPGAs as target platform. Hao et al. [298] give a comprehensive overview of approaches to build a fast NAS framework for simultaneous FPGA accelerator and DNN workload co-design. Over multiple stages those tools optimize the workload to both minimize accuracy and performance loss. For instance, Motamedi et al. [275] introduced in 2016 an approach for quick DSE targeting FPGAs. They are able to find a suitable design by evaluating all feasible solutions using a computational Roofline model. With their approach they can report a 1.9× improvement in computation speed compared to state-of-the-art works at that time.

Besides the more general NAS techniques that we discussed so far, some works have reduced the large search space by focusing on only one platform. HAO by Dong et al. [299] from 2021 presents a fast and accurate NAS approach particularly for FPGAs. Their tool flow leverages flexibility in weight and activation precision that can be realized in FPGAs. However, they do not focus on energy but rather on the relation between accuracy and latency. In 2020 Abdelfattah et al. [276] introduced their NAS methodology for FPGAs using High Level Synthesis (HLS). They use RL to optimize both architectural and model parameters. Their work is based on the CHaiDNN DNN library [300] by Xilinx. Instead of running a HLS for each proposed architecture and DNN model they also rely on a model database, similar to aw_nas. Using their tool, the authors evaluated a huge design space with 3.7 billion configurations, of which about 2000 are actually synthesized. This gave a lot of insight on how various buffer sizes affect latency and power. Compared to state-of-the-art works they report a co-designed model consisting of ResNet blocks that improves efficiency by 41 % using CIFAR-100 as dataset. However, their approach needs many hundreds GPU hours to explore the design space and train all models.

The large body of research that we have covered in this section highlights the fact that the design of an efficient DNN accelerator is not an easy task, but requires a lot of fiddling and tweaking on the myriad of parameters. Luckily, there were already many tools and frameworks to support this presented. Most of them are available as open-source, to make this co-design process more accessible. In the next sections, we will first introduce our cycle-accurate and then our analytical model that builds on top of the already carried out research.

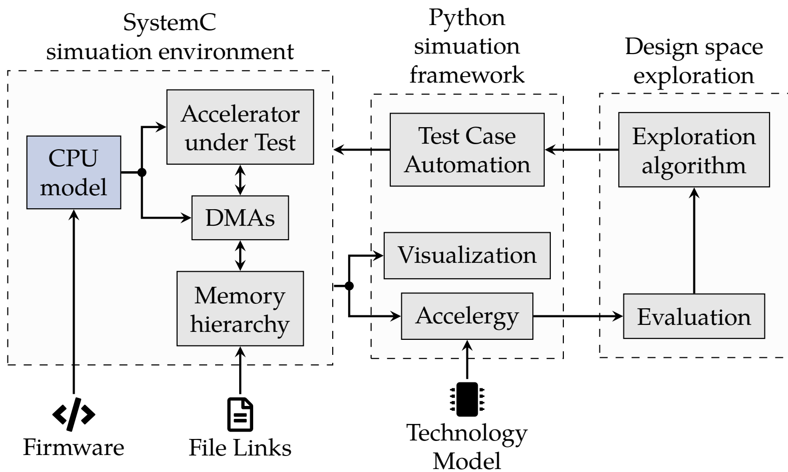


Figure 4.4: Overview of FLECSim. The SoC simulation happens in a SystemC simulation environment, which is controlled by a Python simulation framework and automation layer.

4.3 An Accurate and Flexible End-to-End Co-Design Simulation Framework for System on Chips: FLECSim-SoC

Based on the concept presented in Figure 4.1, FLECSim enables systematic assessment of various DNN accelerator design parameters regarding selected design metrics. The goal while designing FLECSim is to have high flexibility and straightforward ways to integrate new accelerators. Moreover, it should provide all necessary infrastructure to test and evaluate accelerators quickly. Figure 4.4 shows an overview of FLECSim. From a coarse grained perspective, it consists of three layers: a simulation environment that hosts models of all SoC components, a simulation framework, which controls and evaluates the SoC, and finally a DSE component for automated architecture search. The next subsections will give an in-depth description of all layers and their components.

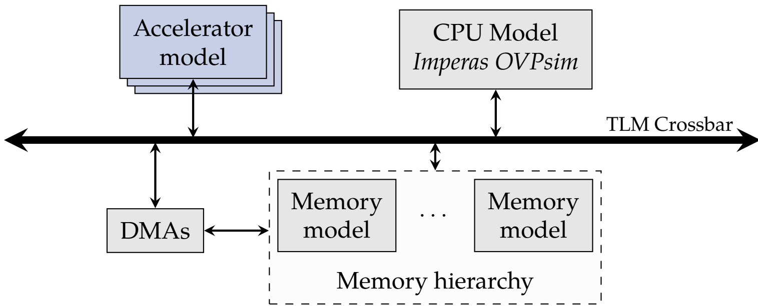


Figure 4.5: Components of the SystemC design inside FLECSim. A set of accelerators is connected to a flexible memory hierarchy and a CPU model via a SystemC TLM crossbar.

4.3.1 SystemC Simulation Environment

At its core FLECSim features a simulation environment, which is based on SystemC. This allows for a cycle-accurate simulation of all hardware components and easy integration of new components. SystemC combines precise hardware simulation and the high flexibility of C++. For example, SoC components like memories or accelerators are specified in SystemC, while file handling and gathering of results can be implemented in C++.

A more detailed view of the simulation environment of FLECSim is given in Figure 4.5. As stated before, the goal is to have a high degree of flexibility and a simple integration of new accelerators. Therefore, FLECSim features a built-in CPU model and a configurable memory hierarchy that can either be accessed directly or via Direct Memory Accesses (DMAs). All components are connected via a Transaction Level Modeling (TLM) crossbar, which makes every component accessible from the CPU in a memory mapped way. Using this infrastructure, a new accelerator can be easily hooked up to a set of DMAs to provide dataflow. In addition, it can be directly connected to the crossbar, which enables access to configuration registers. The simulation environment is configured using a JSON configuration file. It contains, e.g., information about the accelerators and their setup or the memory hierarchy and its layout. Each component collects performance statistics like the amount of data moved or the number of processed operations. These statistics are used to estimate energy consumption. Moreover, it allows identifying performance and energy bottlenecks.

CPU Platform and TLM Crossbar

A crucial bottleneck during SoC design is the mutual dependency of hardware architecture, API and software, which leads to an increased development time and consequently a long time to market. To address this bottleneck, we chose to add an entire CPU model to our simulator. This offers a high flexibility for different applications, workloads and types of accelerators as well as provides all necessary infrastructure for easy integration of new accelerators. This CPU can set up the accelerator and data movement according to the needs of the accelerator. Since the CPU can access configuration registers of all other components, it can likewise adjust memories, DMAs and other peripheral components. Our approach to model an entire CPU enables to port an application that is designed with the support of FLECSim to the later target platform without major changes.

While all other components of our simulation environment are computed cycle-accurate to deliver precise information, the CPU model execution should be fast. Hence, FLECSim uses an instruction-accurate Instruction Set Simulator (ISS), which runs a program using instruction translations, instead of executing each part of the CPU in a cycle-accurate manner. As a result, an instruction-accurate CPU model is much faster, which allows us to run a huge range of applications on the CPU from simple C programs to entire operating systems. Currently, there are some ISS available. For example, gem5 [295] or Imperas OVPsim [301]. In FLECSim we settle on Imperas OVPsim, since it inherently offers TLM ports to communicate with other simulator components. Additionally, it features a wide range of peripherals and processor models like RISC-V, ARM or MIPS cores. Hence, multiple CPU models can be evaluated, while the CPU firmware stays the same. Since arbitrary software may be executed, we can also support common DNN frameworks. For example, the light-weight bare-metal framework Darknet [302], which is entirely written in C. Besides that, we can run Open Neural Network Exchange (ONNX) models, exported from more complex frameworks like PyTorch or TensorFlow, with an ONNX runtime.

As stated before, each SystemC component inside the simulation environment is exposed to the CPU as a memory mapped device. This allows for simple and fast configuration of components, similar to how it is implemented in actual SoCs. Transmitting read and write commands between the instruction-accurate ISS and cycle-accurate simulated components is realized through SystemC TLM version 2.0 [303]. Since the ISS usually executes multiple hundred instructions in the timespan of a single cycle-accurate simulation cycle, communication between those components has to be

carefully designed. The ratio of ISS cycles compared to cycle-accurate cycles can be adjusted by a quantum and instructions per second value. Hereby, the quantum period specifies the time a processor instance has to wait before it can continue. Typical TLM-2.0 core interfaces, as they are offered by OVPsim, are implemented as blocking transports. Hence, in case of a read access, they stop the ISS. To mitigate this behavior, FLECSim sends those commands via a TLM analysis port, which immediately sends data. This approach leads to a significant total simulation time reduction, as thousands of instructions can be simulated without an interrupt by the SystemC scheduler. With our CPU model, early estimates about the overall execution time and efficiency of software and API in interaction with the hardware accelerator can be made.

From a user's perspective, the CPU model is specified using the simulator configuration file as shown in Listing 4.1. The user can specify the CPU type and the corresponding firmware with optional arguments. In addition, all quantum parameters have to be provided.

Listing 4.1: Example of a CPU configuration in FLECSim.

```
1  "cpu": {
2      "cpu_type": "RISCV32",
3      "cpu_mips": 100,
4      "cpu_firmware": "PATH",
5      "cpu_firmware_args": [],
6      "cpu_quanta": {
7          "q_global": 1000,
8          "q_low": 1,
9          "q_high": 1000
10     }
11 }
```

Accelerator Interface

With the provided infrastructure of FLECSim, accelerators can be easily integrated into the simulation environment. A new accelerator model can be directly connected to the standardized TLM communication bus. Then, all required data sinks and sources can be hooked up to the memory hierarchy using DMAs for fast data movement. In addition, small memory accesses might be handled directly using TLM data packets. FLECSim is not limited to DNN accelerators. In fact, any kind of accelerator that supports computation on a heterogeneous SoC might be modelled and integrated.

The number of accelerators in FLECSim is not limited to a single accelerator, but multiple can be instantiated, also of the same kind. To account for different kinds of accelerators, FLECSim offers a SystemC accelerator base class from which specific

accelerators are inherited. This base class provides one TLM port for configuration and status register access as well as a configurable number of DMA ports for large data movement. In addition, it already offers a set of built-in functions, like a clocked process that is called each cycle, a method to update traces that will be recorded, and a setup function, which is called at simulation start.

An exemplary configuration file snippet for an accelerator is given in Listing 4.2. Individual accelerators are identified via an accelerator name. Then each accelerator has a module address that represents the memory mapped configuration registers. Those registers might also be populated to default values using a `registers` entry in the configuration. Finally, `fixed_args` sets hard-coded parameters of the accelerator, for instance, dimensions of a systolic array as shown in Listing 4.2.

Listing 4.2: Exemplary accelerator configuration in FLECSim.

```
"accelerators": {
    "os_array": {
        "accelerator": "OSArray",
        "module_address": 0xA0000000,
        "registers": {},
        "fixed_args": {
            "XDIM": 32,
            "YDIM": 32,
            "ApproxPEs": 1,
            # ...
        }
    },
    "submodules": {
        "process_elements": {
            "submodule": "Module",
            "quantity": 1024,
            # ...
        }
    },
    # ...
},
```

The use of SystemC also comes with natural support to execute C++ code. To open FLECSim to a larger range of accelerator models, it also integrates an interface to the open-source tool Verilator [304]. Verilator converts Verilog or SystemVerilog RTL models into C++ and SystemC. Together with a wrapper that connects a Verilator model to FLECSim, we can run RTL models in addition to bare SystemC implementations. This eliminates the need to manually convert SystemC models into RTL or vice versa. For example, a new accelerator can be sculpted and tested using RTL or SystemC. However, for ASIC synthesis, the latter has to be manually translated into RTL.

Here, we can use FLECSim to verify that both the SystemC and RTL model behave identical. As a consequence, a designer does not need to maintain two accelerator models.

Memory Model and Stream Data Movement

A crucial component, especially in SoCs that incorporate a DNN accelerator, is the memory hierarchy, since huge amounts of data is required to process novel CNN topologies. In addition, near locality of data is key to an efficient inference. Hence, in such SoC designs, a highly flexible and accurate memory model that supports various hierarchies, memory types and interfaces is needed to design a highly efficient DNN accelerator. FLECSim features a simple but highly adjustable memory model that collects important metrics. The memory is represented in SystemC as a large memory mapped array that can be annotated with delays for read and write accesses. During simulation, each memory component monitors the utilization and read / write accesses. With this data, the memory itself can be optimized towards a high utilization and subsequently high energy efficiency. Multiple individual memories may be chained together to form a memory hierarchy.

Listing 4.3 shows an exemplary configuration of memories and DMAs. To instantiate a memory, one has at least to specify its size and starting address in the memory mapped address space. Memories can additionally be preloaded with data from binary or numeric files. This is, for example, interesting for DNN workloads. Here, weights for each layer can be loaded into the memory before an actual simulation starts to accelerate the simulation process significantly. An example is also given in the following listing.

Listing 4.3: Memory configuration example in FLECSim.

```
1  "memories": {
2      "dram": {
3          "module_address": 0xB0000000,
4          "length": 0x40000000
5      },
6      # ...
7  }
8  # ....
9  "files": {
10     "input_file": {
11         "file": "FILE PATH",
12         "data_type": "float",
13         "data_representation": "binary",
14         "data_organization": "sequential",
15         "address": 0xB0000000,
```

```
16            "items": 10,
17            "memory": "dram",
18            "operation": "file_to_memory"
19        },
20        # ...
21    }
22    # ...
23    "dmas": {
24        "acc_supplier_a": {
25            "direction": "to_module",
26            "module_address": 0x90000000,
27            "source_module": "dram",
28            "destination_module": "os_array",
29            "destination_port": "in1",
30            "port_size": 1024
31        },
32        # ...
33    }
```

DMAs are configured by providing a source and a destination, for example, by giving the name of the destination accelerator port or memory port name. The `port_size` parameter tells how many bits can be transferred per cycle. An instance of a DMA can later be configured in a fine-granular way by the CPU firmware. Each DMA supports a scatter gather memory access pattern with jump widths and block sizes. This enables again a high degree of flexibility to meet the requirements of any kind of accelerator. For very specific data access patterns, the DMA provided by FLECSim may also be extended or altered in SystemC to have the highest configurability. To make this extension simple, the memory interface mimics an AXI-Stream interface with the corresponding handshaking mechanisms.

Implementation

As described before, the simulation environment of FLECSim is implemented using SystemC and C++. Figure 4.6 gives a brief overview of how FLECSim is implemented and how auxiliary component are set into function. All components that are actually simulated derive from the type `SimModule`. This class provides, for example, a method that is called in each simulated clock cycle and can be overloaded in the corresponding child classes. In addition, all `SimModule` objects are observed by a registry. It gets, e.g., notified every time data is moved or processed in the module or when other metrics are available. The registry collects all information, compresses it and makes it available for later evaluation. For instance, it may be converted into a cycle-accurate activity diagram that shows for each module which operation ran how often and when. This reveals possible performance bottlenecks and gives the utilization of

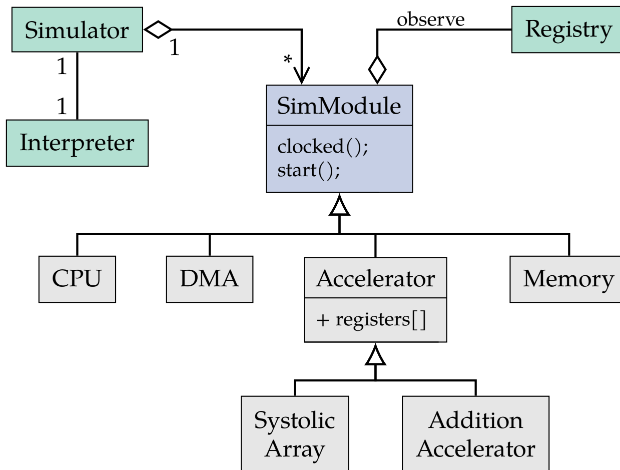


Figure 4.6: Implementation details of FLECSim, showing the individual models and how they interact.

each monitored simulation component. As described before, accelerators themselves derive from an accelerator base class that offers an additional abstraction layer and already offers necessary function prototypes. To create specific accelerator objects when the simulation is started, the base class implements a factory pattern. Based on the name given through the JSON configuration file, it instantiates all accelerators and connects them.

Additional components encompass chiefly a JSON interpreter, a file handling module and finally the main simulator class. The latter is instantiated directly by the main function and is entry point for a simulation. It first triggers the JSON interpreter to read the configuration file. Then it creates a simulation environment according to this configuration. Hereby, all components are created and connected to the TLM and memory infrastructure. Once every component is initialized it proceeds by starting all components including the simulated clock. During the simulation, all components can provide trace information that is gathered and stored afterward as a timing diagram. In addition, utilization and operation statistics are collected to compute the required energy consumption.

To test and verify all components of the simulation environment, FLECSim also features unit tests for most components using the Google Test[9] framework. Besides that, logging and handling of command line arguments are handled by third party libraries as well. Finally, we use *cmake* to have a modern build automation system that resolves all library dependencies while compiling the executable simulator. Now, FLECSim can be started directly from command line with a set of arguments like the configuration file or whether wave form traces should be captured.

4.3.2 Python Simulation Framework

On top of the SystemC simulation environment, builds a simulation framework written in Python. This allows to automate all steps required to run a SoC simulation. Moreover, Python enables direct integration into common ML frameworks. For example, a DNN can be trained first and then weights for the cycle-accurate SoC simulation can be exported. To make automation possible, the simulation framework of FLECSim has two main components: First a test case environment and then a module that computes area and energy consumption of the SoC under test.

Test Case Automation

In FLECSim a SoC designer can either configure the architecture by using the aforementioned JSON configuration files using a Python abstraction layer. Here a specific application is represented by a test case that runs on a simulated SoC. The main advantage of this simulation framework is that all steps needed for a simulation can be automated. For example, a simulation does not only require the JSON file, but also a CPU firmware, data for the accelerators and means to verify results generated during simulation. For further automation, multiple test cases can be grouped into a test suite, which runs in parallel to accelerate the design process.

The simulation framework as it is implemented in FLECSim is shown in Figure 4.7. Each test case consists of a series of steps (right side inside the test case box) and a list of components, like memories and accelerators (left side inside the test case box). When a test case is started, it first walks through the component list and checks for dependencies or conflicts. Then the steps are executed sequentially. Typically starting by cleaning old results, building the simulation environment if necessary and setting up input data. Another step takes care of building the CPU firmware. Hereby, a test

[9] https://github.com/google/googletest

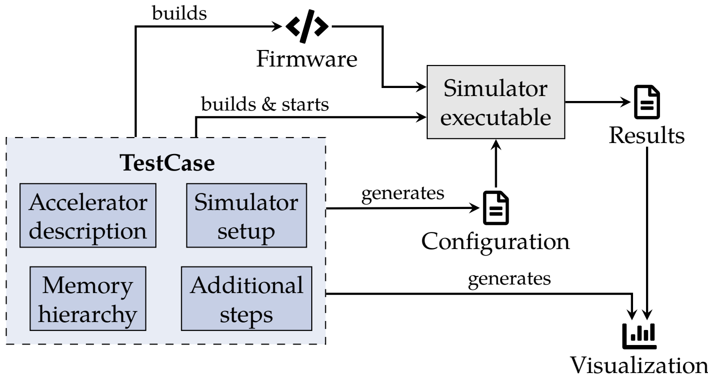


Figure 4.7: Test infrastructure of FLECSim.

case can provide a set of constants, which will be build into the firmware, for example, the memory addresses. Hence, a software engineer only has to change the component mapping in one place. Then a step generates the required configuration file from the component list, including additional parameters like the path to the CPU firmware. Finally, another step starts the actual simulation. Thereby, it constantly monitors the log file for potential error messages. Additionally, a timeout can be set, since some simulations might get stuck. At the end of a simulation, several verification steps may be added. They can either look for specific keywords in log files or compare output data, which got dumped from the simulated memory, with golden data.

Performance, Energy and Area Estimation

After the simulation has successfully finished, FLECSim outputs various metrics. Those metrics are important to properly assess the SoC architecture and hence are crucial for a DNN algorithm accelerator co-design. As we discussed in the background chapter, common important metrics to evaluate DNN accelerators are energy consumption, latency, required chip area and total throughput. In addition, the model accuracy can be considered, when optimizations like pruning, quantization or compression are included to the design space.

Our objective is to produce all those metrics directly from FLECSim. The number total number of cycles for the simulation is derived from the SystemC clock. However, this information only gives a total number of cycles and no detailed information about whether a specific module was idle or actively working on a task. Therefore, we use in-

formation provided by OVPsim for the CPU and the data collected during simulation by the registry for all other components that are simulated cycle-accurately. OVPsim itself already reports various metrics such as the number of executed instructions and the overall simulation time. It also offers traces to capture each executed instruction. Data collected by the registry from all cycle-accurate components is analyzed in our Python simulation framework. It can, e.g., create visualizations that show the activity of each module over the simulation time or do a static calculation of each component's utilization. Furthermore, plotting the memory accesses over time helps to identify patterns to arrange data more efficiently in memory, and a distribution of operation types shows what operation can benefit from acceleration. All those visualizations and analysis steps may be run separately after the simulation has finished. Since all registry data, action counts and activity data is stored, it can always be reevaluated without running the entire simulation again.

Insights about energy consumption and total chip area are generated by *Accelergy* (see Subsection 3.2.3). Therefore, our simulation framework converts data collected by the registry into action count files for energy estimation, and the JSON configuration file into a component specification for the area estimation, respectively. For most of our experiments, we use the 45 nm primitive component library provided by Accelergy and also the according memory models from CACTI. However, other technology libraries may be used. The overall system power consumption is then computed in combination with the results from the ISS and the TLM interfaces.

4.3.3 Design Space Exploration

Closing the loop of iterative architecture optimization is taken care of by a DSE layer in FLECSim. Here, the core component is an exploration algorithm, which tunes the design parameters so that a set of objectives functions are maximized. Objective functions encompass constraints like upper power or area bounds and ranges for individual design parameter, but also more loose constraints like a design goal which maximizes throughput. Multiple objective functions may be combined and weighted by factors to achieve, e.g., an architecture that favors both energy efficiency and throughput FLECSim comes with algorithms for exhaustive search and goal-directed search using GAs. The latter is provided by Pymoo [305], which also provides other optimization algorithms. While an exhaustive search does not yield a fast architecture search, it certainly helps to directly understand the impact of different design aspects on the evaluated metrics. When the search space is sufficiently understood, adequate

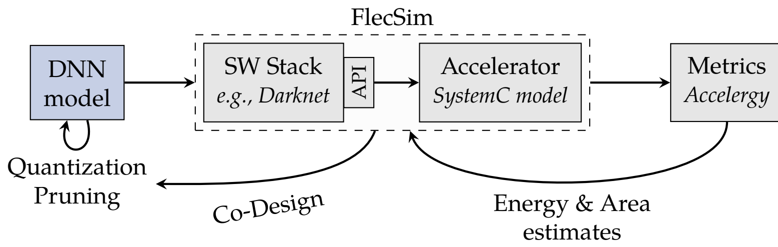


Figure 4.8: Design flow using FLECSim. An optimized DNN workload is fed into the framework. Using metrics as results, a design space exploration can search for an optimal hardware structure for a given workload.

constraints for metrics and design parameters can be set and thus a GA can explore the search space faster. To revisit already evaluated architecture configurations, each successfully ran simulation is saved. These can be reevaluated later, without the need to rerun the entire DSE. Design parameters that the exploration algorithm can alter are exposed by each accelerator under test as fixed parameters. Exemplary options are the number of PEs, available memory bandwidth or the size of local memories. Obviously, there are countless conceivable design parameters. Hence, a designer has to pick a set of interesting parameters to enable a fast architecture search.

How an actual design flow with all three layers of FLECSim looks like is sketched in Figure 4.8. An optimized DNN model is given to FLECSim, which provides accuracy for co-designing quantization and pruning. With an accelerator model, energy and area is estimated and fed back into FLECSim. These metrics are then considered by the exploration algorithm for goal-directed parameter optimization.

4.4 Analytical Modelling of Systolic Arrays for Rapid DNN Accelerator Assessment

Cycle-accurate models are very accurate and provide valuable insights about which component or operation requires the longest processing time or accounts for the largest share of the total energy. But there is one major drawback in cycle-accurate simulation, namely the long time it takes to simulate every single cycle of a device under test. For small designs this is usually nothing to worry about, but for large DNN accelerators it quickly becomes adverse to a point at which a DSE is rendered

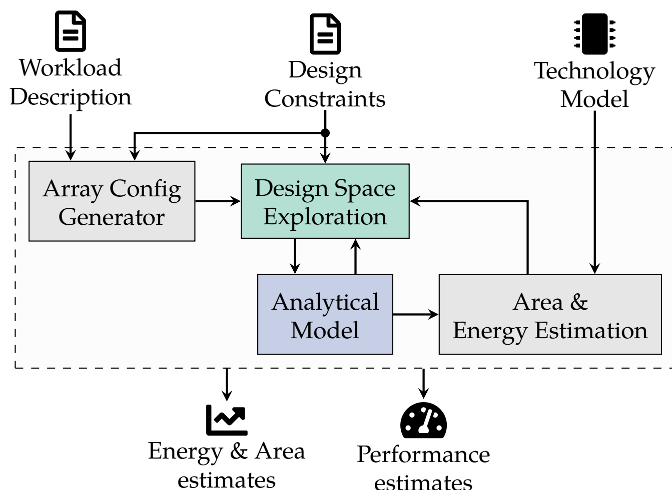


Figure 4.9: Overview of our DNN accelerator evaluation tool flow with our analytical model of a systolic array at its center.

unfeasible. However, there is one substantial characteristic in DNNs that we can leverage to compute the required operations and the number of cycles it takes to process a workload. As we already highlighted in the background chapter, DNNs have a very homogenous dataflow, no jump statements and have a relatively small number of different operations. With these characteristics, we can conceive an analytical modelling approach. This model estimates the number of cycles and operations using merely workload parameters like input dimensions and architectural characteristics. Instead of computing each time step and monitoring changing signals, this approach uses a set of equations to directly compute the desired performance metrics. Obviously, this model is largely dependent on the underlying accelerator and may offer less flexibility compared to the previously introduced cycle-accurate approach. However, by focusing on one of the most common types of accelerator and keeping the design extendable, we can still achieve a high degree of flexibility coupled with fast estimation. Running a cycle-accurate simulation with design space constraints found by an analytical evaluation even allows to combine the best of both worlds. A fast, but rough, assessment of an accelerator gives first insights on relevant design parameters, which can then be fine-tuned using FLECSim.

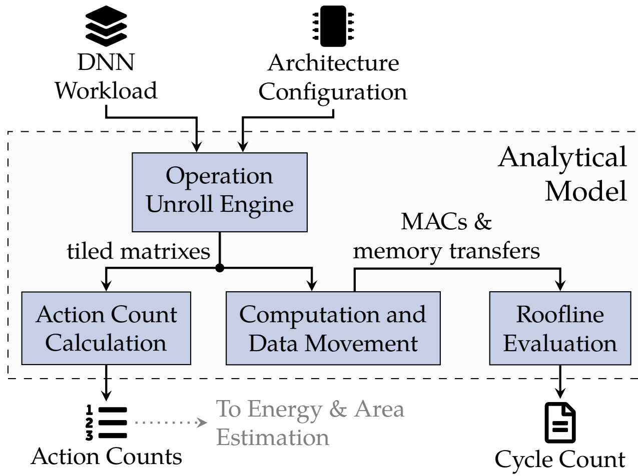


Figure 4.10: Overview of our analytical model.

In the next subsections we will take a look into how such an analytical model is constructed and how we can use it to accelerate the search for an optimal DNN accelerator. Figure 4.9 shows an overview of our DSE tool with the analytical model at its center. It models one of the most common accelerator type: systolic arrays. Based on a workload and accelerator description we create a design space with all possible configurations. In combination with an area and energy estimation, similar to FLECSim, we can then evaluate each accelerator configuration. To use FLECSim for subsequent architecture fine-tuning, we kept the output and input formats compatible.

4.4.1 Analytical Accelerator Model

The core of our analytical assessment approach is an analytical accelerator model. As input, it gets an acceleration configuration, which is taken the design space of possible accelerator configurations, and a workload description. With this it computes important output metrics like the number of operations in terms of action counts and the number of cycles needed for an inference. Our analytical model enables fast and systematic exploration of a systolic array DNN accelerator to find the best-fitting configuration for a given workload in a huge design space. Since many parameters, like buffer sizes, the number of PEs and the interface bandwidths have a large impact

on the performance, our analytical model has to deliver accurate and fast estimates of the accelerator's metrics. Therefore, it is based on the well-established Roofline model [114] (see Figure 2.20) to determine the number of cycles it takes to process a workload. Energy and area estimation is, similar to the cycle-accurate model, done by Accelergy [169]. This tool combination allows us to design a highly abstracted model of the underlying hardware architecture. Figure 4.10 gives a brief overview of its components, which will all be explained in-depth in the following subsections.

Operation Unroll Engine & Action Count Calculation

The first component of our analytical model is the operation unroll engine. It splits large input problem matrices into smaller tiles that match the accelerator dimensions. Therefore, it takes two inputs, which are necessary to compute the tiling factors: First, a description of the current DNN layer that defines the dimensions of the currently evaluated input, weight and output tensor. Second, an architecture description that contains the systolic array's dimensions and the size of the memories such as scratchpads or the PEs buffers. The tiling factors depend on the PE rows and columns, which we define as $DIM_x \times DIM_y$.

Once the number of required tiles is computed, we can give this output directly to the action count calculation module. It chiefly derives action counts, which are given to Accelergy for energy estimation. They represent how often a given action is performed by a component, for example, the number of memory accesses.

Computation and Data Movement

The objective of the subsequent computation and data movement module is to estimate the number of computational and data movement operations required to process a given workload. As input, it takes the tiling done by the operation unroll engine. In terms of operations all computational cycles of a DNN are caused by MAC operations. Their amount can be derived from a matrix-matrix multiplication between an $l \times m$ matrix A and an $m \times n$ matrix B performed in the systolic array. In case of a DNN layer the matrix A might be considered as input feature map and matrix B as weights. Obviously, for Convolutional (CONV) layers the matrices are first transformed using im2col, which is necessary to process them efficiently on systolic arrays. In total, this accounts for $l \cdot m \cdot n$ MAC operations. However, this only works when the matrix dimensions exactly fit the accelerator dimensions, which is usually not the case. Instead, we have to take care of mapping fragmentation.

Assuming an extreme scenario with two 129×129 input matrices, 2 146 689 MAC operations are needed for matrix multiplication. However, a 128×128 systolic array requires four passes, resulting in total 16 277 216 MAC operations. Although this is one of the worst problem shapes for such an accelerator, it makes clear how mapping fragmentation impacts the cycle count estimation. The just explained phenomenon is called temporal fragmentation. More generally speaking it always occurs when a dimension of an input matrix is greater than the systolic array's dimensions, and not an integer multiple for it. Considering a simpler example, where a 24×16 matrix is computed on a 16×16 systolic array. After one fully utilized pass, eight columns remain and the array is only 50 % utilized, resulting in an overall utilization of 75 %. To consider this, we introduce a scaling factor δ. It computes to $\delta = \lceil m_{dim}/a_{dim} \rceil \cdot a_{dim} / a_{dim}$ with a_{dim} representing a dimension of the systolic array and m_{dim} the corresponding matrix dimension. Looking at our example, we get $\delta = \lceil 24/16 \rceil \cdot 16/24 = 2 \cdot 16/24 = 4/3$.

A second mapping fragmentation effect that can occur is spatial fragmentation. This happens when a dimension of an input matrix A or B is smaller than the systolic array's dimensions. In this case, the array is not fully utilized and some PEs are idle during processing. Hence, the matrix has to be padded to fit its dimensions. For example, a matrix multiplication in which A and B are 8×8 on a 16×16 systolic array results in 25 % utilization. To factor in this effect, we add another scaling coefficient $\eta = a_{dim}/m_{dim}$. In the example above this coefficient would compute to $\eta = 16/8 = 2$. It has to be noted that η has to be computed for both dimensions of the array.

Putting everything factors together, we can compute the actual number of MAC operations required to compute an arbitrary DNN layer on a given systolic array:

$$macs_{scale} = l \cdot m \cdot n \cdot \eta \cdot \delta \tag{4.1}$$

Besides the number of computational operations, data movement between the accelerator and associated local memory has a major impact on the performance figures. Hence, for an accurate analytical modelling it is crucial to consider them as well. The performance of data movement strongly depends on the available bandwidth, determined by the bus-width. In general, data movement is arranged as block transfers into the accelerator. Those blocks can be treated as transfers of individual matrix rows with the dimensions $mem_{rows} \times mem_{cols}$, representing the rows and columns of data elements that are transferred. Similar to the computational cycles, we have to consider the case when a row of data moved through the bus is smaller or greater

than the bus capacity. Then, again fragmentation happens. For example, even if only 1 B is transferred over a 256-bit bus, it still effectively blocks the full bus-width. In this case, we have to pad bus transfers. As a result, every bus transfer is scaled to match the maximum bus-width, which we denote as bw_{peak}. The associated scaling overhead is considered with a factor α, which basically covers additional unused data that is transferred. Obviously, large data transfers and individual rows have to be split into multiple transfers when they are larger than the bus-width. In addition, idle periods of the bus have to be accounted for as well. This occurs, every time when data has been moved into a local memory and the systolic array is busy processing this data and is not directly requesting new data from the off-chip memory. Combining the factor for padding and idle phases we can compute the actual number of transmissions $data_{scale}$ that are required for a DNN workload, as follows:

$$
\begin{aligned}
N_{bw} &= \left\lceil \frac{col}{bw_{peak}} \right\rceil \cdot row \\
data_{scale} &= N_{bw} \cdot bw_{peak} \cdot \alpha
\end{aligned}
\tag{4.2}
$$

Roofline Evaluation

Once we know the number of MAC operations and memory transfers required to compute a workload, we can determine the cycle count. This happens in the roofline evaluation module, which hosts equations for the Roofline model, on which our analytical model is build on. In general, the Roofline model can be applied to make estimates about all kinds of computational tasks. It gives the relation of an application between performance in terms of operations per second and the operational intensity, namely how much data is needed for an operation. With this relation one of the most important thing we can determine with the Roofline model is the break-even point between compute-bound and memory-bound execution. This break-even point represents the sweet-spot for a given application as memory bandwidth exactly meets the requirements and the number of operations per second is maximized. Besides that break-even point, the Roofline model also yield the maximum memory bandwidth requirement for the application. To put the Roofline model into effect, two application and architecture characteristics have to be known: the peak computational performance, denoted as $comp_{peak}$ and the peak bandwidth, denoted as bw_{peak}. The goal of our analytical model is to identify this break-even point. Since, this can be done through a set of equations rather than cycle-accurate simulation, computing it is much faster.

With scaled data movements and MAC operations, we can directly use the Roofline model to determine the actual cycle count that a given DNN accelerator takes to compute a presented workload. The two architectural characteristics $comp_{peak}$ and bw_{peak}, which are crucial inputs to the Roofline model, are provided by the number of available MAC units in the systolic array and by the bandwidth of the bus between accelerator and off-chip memory, respectively. Now we have all variables in place to calculate the performance using a set of equations given in Equation 4.3. $op_{intensity}$ is determined by the quotient of scaled MAC operations and data movement, since it is effectively the number of data that can be provided per MAC operation. To transform this into cycles, we take the total number of MAC operations and divide it by the performance, which is provided by the Roofline model. Here we consider a latency of one cycle per MAC operation. If a MAC operation takes longer, the number of cycles can be scaled accordingly.

$$\begin{aligned} op_{intensity} &= \frac{macs_{scale}}{data_{scale}} \\ performance &= \min\left(op_{intensity} \cdot bw_{peak}, comp_{peak}\right) \\ cycles &= \frac{macs_{scale}}{performance} \end{aligned} \tag{4.3}$$

Estimation of Energy and Area

As described before, we can estimate the required energy per inference and the chip area of a given systolic array using Accelergy. This allows us to have comparable results to FLECSim and other works as Accelergy is frequently used. For energy and area estimation we need the action counts from the analytical model, an architecture description, and a technology model. Action counts are generated by our analytical model in the corresponding module based on the total number of performed MAC operations. In terms of memory accesses, which strongly influence the energy estimation, so far out analytical model only outputs the number of accesses, but not the memory size or how memory accesses look like. However, we can keep this architecture specific parameter open for now to keep our model flexible. Later, we will fix this parameter (Subsection 4.4.3), where we adjust our model to analyze an actual systolic array. Based on the accelerator structure and all architecture specifics, we can derive an architecture description to estimate the area. A technology model has to be provided externally. With all three inputs in place, we can launch Accelergy.

4.4.2 Design Space Exploration

The main goal of our analytical modelling approach is to accelerate the DSE process and at the same time to yield accurate results. To systematically explore and evaluate all feasible and viable design configurations, we first have to construct the architectural design space. Similar to the cycle-accurate simulation, our design space is derived from a DNN workload description and a set of design constraints. This happens as first step in the array configuration generation module. It discovers and constructs the entire design space for the given problem and architecture. More specifically, it generates a set of formal descriptions that are then evaluated in the analytical model itself. Those formal descriptions are essentially combinations of different architecture parameters, for example, permutations of all systolic array dimension with all available memory and bandwidth sizes. The complete DNN workload is broken down into individual layers, since our analytical model computes each individually. To compute the total latency, we simply accumulate each layer's cycle count. Since evaluating each element of the design space for all layers may consume a lot of time, it is typically sufficient to just run a subset of all layers of a given DNN workload. After a feasible architecture is found, an evaluation with the entire workload can follow.

The entire design space can be generated, by providing all elementary parameters of the systolic array accelerator, like the number and arrangement of PEs or the size of on-chip memories. To construct the actual design space with all possible accelerator configurations, the array configuration module generates all permutations of PE rows and columns and their arrangement. Then for each combination, it picks all feasible memory configurations. For example, a large 128×128 systolic array need at least a certain amount of memory to run properly. Once the design space, which can easily consist of multiple thousand configurations, is constructed, invalid configurations are discarded. Such configurations may, for example, break design objective provided by the user or accelerator constraints. The set of design goals is described in the constraint file, similar to FLECSim. For example, a user can optimize towards a highly energy-efficient architecture that still delivers a certain throughput.

Starting from the vast design space, we can now apply different optimization algorithms. Since the analytical evaluation is quite fast, even exhaustive search becomes feasible This gives also a very good overview of the impact different design characteristics have. In this case, we can even accelerate the process by running multiple evaluations in parallel. The resulting PPA figures are then post-processed and evaluated regarding their cost. Post-processing involves mainly a check for constraint

violations, like too much consumed energy or a too large area requirement. From a set of ran configuration, our DSE tool can compute the Pareto front with regard to area, power and performance. User specified goals can then be found in the Pareto front. Finally, the DSE tool can visualize the found solution space and the associated costs.

4.4.3 Implementation using the Systolic Array Generator Gemmini

Until now, we have intentionally described our analytical model in a generic form. However, at some point we have to fix some design aspects to model a specific flavor of DNN accelerator and get real-world accurate cycle estimates. This is typically done by setting the right constraints. In the following, we will elaborate how such an adjustment is done by exemplary means of the Gemmini [149] systolic array generator. Gemmini is chosen since it is available as open-source implementation and thus gives a lot of flexibility and insights about its internal structure. Moreover, it is highly configurable and supports a wide range of DNN workloads, for which a software infrastructure including an ONNX runtime is available.

Figure 4.11 gives an overview of Gemmini and its auxiliary components required to control and use it in a SoC. Gemmini is part of the Chipyard framework [150] that provides an ecosystem to design entire SoCs. In such an SoC, Gemmini is controlled by a CPU, in particular, a RISC-V Rocket core [306]. With all these components, Gemmini can execute various workloads through a rich software stack that is shipped with the accelerator. The basic architecture of Gemmini consists of a scratchpad to store operands, an accumulator memory in which results are stored and the systolic array that performs the actual computations.

As stated before, we have to adjust a few parameters of our systolic array and consider some design constraints, to integrate Gemmini into our systolic array analytical model. First the operation unroll engine has to match Gemmini's tiling and account for Max-Pooling, which is performed during write-backs to the main memory. In addition, we have to update the formulas of the Roofline model evaluation. In terms of design constraints, the size of the systolic array and the number of rows in each memory has to be a power of two, which is due to the Address Generation Unit (AGU) in Gemmini. Moreover, the systolic array has to have a square shape. In the following, we thus denote its size as $DIM \times DIM$. To estimate performance, we can neglect the tile structure of Gemmini for now, since for our analytical evaluation the total number of

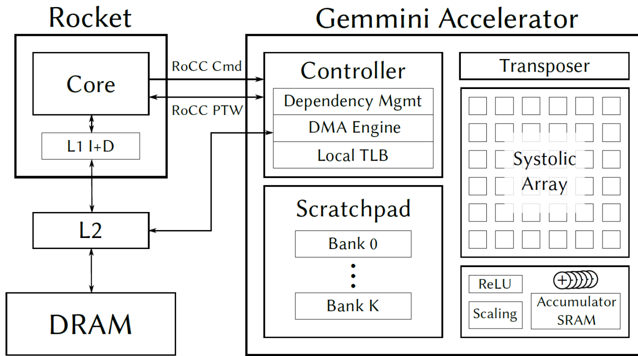


Figure 4.11: Overview of the Gemmini systolic array generator presented in [149] and how it is integrated into a SoC using Chipyard, which provides all required infrastructure to quickly design an entire SoC design.

MAC units is predominantly interesting. However, for energy and area estimation, we have to keep them in mind. As next constraint, Gemmini fixes the available bandwidth to 128 bit. Consequently, we can determine the needed architectural parameters for the Roofline model as follows: $peak_{comp} = DIM \cdot DIM$ and $bw_{peak} = 128\,bit$.

To enable proper energy and area estimation, a couple of models for some components are added to Accelergy. First, we added memory blocks for scratchpad and accumulator memory. Both are modelled as Static Random Access Memory (SRAM) banks with a configurable size and number of banks. In terms of their distribution, Gemmini configurations must have a larger number of scratchpad banks compared to accumulator banks. Accumulator memories additionally contain a set of adders to perform dedicated accumulate-on-write operations. Gemmini's execute controller that hosts the systolic array itself, is modelled very close to the actual structure of Gemmini, which is shown in Figure 4.12. In Accelergy, the systolic array is composed of an array of tiles connected by registers with an adjustable delay. Each tile then hosts a configurable set of PEs, each made out of a MAC unit and two registers.

Now that all required parameters and models for our analytical model are in place, we can move on and compute the number of MAC operations and data movements needed for a given workload. Therefore, we have to analyze all individual operations Gemmini can execute. From this, we can also directly derive specific action counts of each component, which is necessary for a detailed energy estimation. Our model of

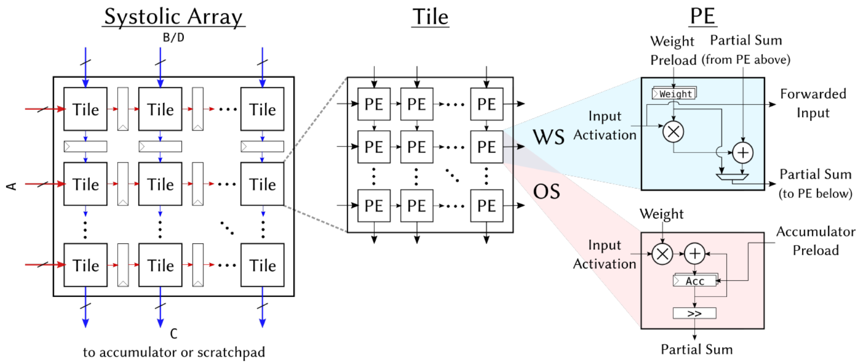


Figure 4.12: Detail view of Gemmini's systolic array following [149] that consists of tiles, which host the actual PEs. Each PE can be configured as weight or output stationary.

Gemmini primarily focuses on fully-connected and convolutional layers as well as on Max-Pooling operations, which account for all operations that occur in most CNNs. Those layers can be scheduled efficiently on Gemmini by the on-chip loop-unroll component. It has to be noted, that in particular Max-Pooling operations are not considered by other state-of-the-art modelling tools, which can lead to a significant deviation in cycle and performance estimation.

To get the actual number of computational operations, we have to look at two operations in Gemmini: `compute_preload` and `compute_accumulate`. Computing the total sum of MAC operations follows the equations we introduced before as Equation 4.1. To account for the aforementioned fragmentation effects we use the same scaling factors, but adopted for Gemmini. For spatial fragmentation, we define η to scale m and n so that their scaled values each match DIM. In doing that, we can model each of these instructions to block the entire systolic array. Temporal fragmentation has to be handled differently for the two compute instruction. In `compute_accumulate`, the scaling factor is set to one as this effect does not occur here. When a `compute_preload` operation is called, we have to scale l to match DIM as no other calculations can be performed on the systolic array within l cycles. To analyze the data movement, we can use Equation 4.2. Here all parameters stay the same. Putting everything together, we can summarize all equations needed to estimate the cycles and performance of Gemmini as follows:

$$
\begin{aligned}
\eta &= \frac{DIM}{m} \cdot \frac{DIM}{n} \\
\delta_{preload} &= \frac{DIM}{l} \\
\delta_{accumulate} &= 1 \\
macs_{scale,preload} &= l \cdot m \cdot n \cdot \eta \cdot \delta_{preload} \\
macs_{scale,accumulate} &= l \cdot m \cdot n \cdot \eta \cdot \delta_{accumulate}
\end{aligned}
\tag{4.4}
$$

Besides performance and cycles, we also want to get information about the action counts in order to estimate energy requirements. The number of MAC operations can be taken from the equation above. Another driver for power consumption are obviously memory transactions.

Specifically in Gemmini, data movement to get all inputs for a General Matrix Multiply (GEMM) works as follows: Before compute operations take place, a weight matrix of size $m \times n$ is preloaded, which is done using the `preload` instruction. Similar to other data movement, this will result in n on-chip memory accesses. Data movement between the L2 cache and the on-chip memories is initiated by the `mvin` and `mvout` instructions. Depending on their arguments, we can determine how many on-chip memory accesses are performed. As previously discussed, a $rows \times cols$ matrix can be moved between external and on-chip memories. In terms of total data movement, such blocks as a result cause a total of $\lceil {}^{col}/_{DIM} \rceil \cdot row$ memory accesses, which is also represents $data_{scale}$ from Equation 4.2. Considering the operations that are performed, we have to distinguish between accesses to the scratchpad and to the accumulator memory. Further, for the accumulator, we have to differentiate between overwrite and accumulate-on-write operations. In the latter case, adders are used to accumulate previously written results at a given address to a newly written value.

With this we have all parameters needed as input for the Roofline model and adjusted them to match the behavior of Gemmini. In the next step, we can verify the introduced analytical model, and then start to evaluate various DNN workloads and systolic array configurations.

4.5 Evaluation and Discussion

Now we have introduced two different implementations of DNN simulation and assessment environments. In this section, we will take a deeper look at both and perform a throughout evaluation. Important metrics here are the actual accuracy in comparison to cycle-accurate simulation and simulation time. Therefore, we picked representative accelerators and CNN workloads, which then undergo various tests. We also performed two case studies, showing fields in which our simulation environments support a DNN accelerator design.

4.5.1 Design Space and Setup

Before evaluating our simulation environments, we had to fix some constraints to restrict the design space for the accelerators. To verify our two concepts and make them comparable, we chose a systolic array as accelerator for both simulation environments. The static model is already fixed to a systolic array, while for FLECSim we had to design a corresponding SystemC model. In both cases, the number of PEs and size of the memories are adjustable. To get comparable energy and chip area estimates from both the static model and FLECSim we use Accelergy with the same technology model for both. In particular, estimates are based on a 40 nm reference table for all memories and 65 nm for all remaining logic like the accelerator itself.

Besides things that are configured the same way for both approaches, some parameters and design decisions are unique. We start with FLECSim. Since we can simulate an entire SoC, we used a 32-bit RISC-V processor model and a TLM memory, both provided by Imperas. This RISC-V CPU is close to the Rocket core CPU used in Gemmini, which in-turn is used in our analytical model. The DMA model is delivered with FLECSim. For the analytical model, no additional architecture constraints than those required by Gemmini, as discussed before, have to be considered.

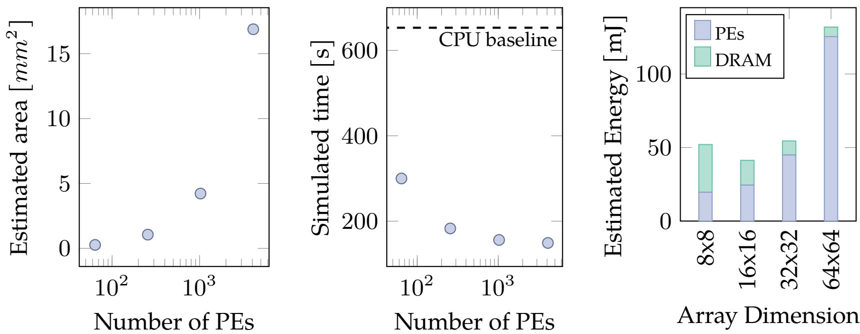
In terms of the DNN workload, we focused on applications for embedded systems, since they pose various challenges regarding energy efficiency and performance to the DNN accelerator. More specifically, we chose small CNN classification networks. Those small workloads are already highly optimized, e.g., through pruning or NAS and have as a result a small memory footprint and comparably fewer MAC operations. The first workload we selected is Tiny-Darknet [307], a CNN classifier network, that is contained in the Darknet framework. Since Darknet is available as C implementation

without dependencies, it can easily be integrated into the CPU simulation of FLECSim and Tiny-Darknet's topology can be modelled for our analytical model. Tiny-Darknet consists of 16 CONV layers that account for 980 million operations in total and has a memory footprint of about 4 MB. Second, we picked LeNet-5 [37], a very small network that allows for a fast parameter exploration. Consequently, it is very suitable to experiment with different architectural aspects. The third workload in our portfolio, is the well-established CNN classification network ResNet [40]. Its advantages are that it offers various kernel sizes and channels dimension, which is useful to test the precision of our simulation models. Moreover, it is available in different configurations from embedded to datacenter applications. ResNet's input size is $3 \times 224 \times 224$, and we set the batch size to one. For a case study of ResNet we chose a ResNet-34 configuration, which demonstrates good prediction accuracy while having a reasonable parameter and operation count. To prove correctness, all workloads are launched with pre-trained weight data, allowing us to evaluate the classification accuracy with potential architectural optimizations such as dynamic input feature map pruning.

4.5.2 Cycle-accurate Simulation using FLECSim

Using the accelerator and SoC configuration described before, we ran multiple simulations using FLECSim with different accelerator parameters to observe their effect on the performance metrics. Therefore, FLECSim performs an entire cycle-accurate simulation and monitors area requirements of all modules, simulation and execution time, and energy consumption. Here, every step of the inference is simulated, including weight and image loading, feature extraction and classification. We start with Tiny-Darknet that ran in 32-bit floating-point precision.

Darknet uses the im2col transformation for its CONV layers to enable a mapping onto a systolic array and a simpler memory access pattern. As our simulated accelerator reassembles a systolic array, we can directly map the resulting matrix-matrix multiply operation onto the array. The accelerator model performs this GEMM operation in an output stationary manner. Hence, partial results of inner products are kept local inside the PEs and input data is distributed among them. To match the accelerator dimensions, input matrices of each layer are divided into multiple chunks that are streamed from our memory hierarchy via DMAs to the accelerator. The CPU model, which runs Darknet, initiates the processing. Each layer is executed individually. To start the inference, the CPU firmware configures the corresponding DMA channels, which then start streaming data into the accelerator. When sufficient data is available



(a) Area estimation of the accelerator with different accelerator dimensions.

(b) Simulated execution time of the SoC and CPU only baseline.

(c) Estimation of dynamic energy consumption for DRAM and accelerator.

Figure 4.13: Comparison of Tiny-Darknet inference on an exemplary SoC for different accelerator sizes.

the systolic array accelerator starts to process it. Results that have to be offloaded into the memory hierarchy are moved by additional DMA channels, which are also configured by the Darknet framework. All data movement between the memory hierarchy and the accelerator is implemented using AXI-Stream and all communication between the CPU and the peripherals with AXI-Lite, respectively.

The total execution time and cycle count was taken from the trace file that FLECSim generates. We define it as the time of the last trace entry, which means that data loading and classification by the CPU and feature extraction by the accelerator have finished. Our RISC-V processor model ran at a speed of 40 MIPS, comparable to other low-power processors used in mobile devices. For accurate simulation all components and modules, we picked a quantum duration of 1 µs according to the Imperas OVPsim guidelines [301]. To avoid errors during simulation, the simulation clock cycle duration of all SystemC modules must be higher than the quantum duration. Therefore, we set the clock cycle of the processor peripherals to 10 µs.

With a configuration of FLECSim ready for accelerator assessment we started to evaluate first how different array sizes influence the performance and area. Then we performed a small case study in which we explored the accelerator dimensions and memory bandwidths, two of the major contributors the accelerator's performance.

Figure 4.13 shows the performance results in terms of execution time, area and energy of Tiny-Darknet using different accelerator dimensions. We explored accelerator sizes from 8×8 to 64×64, revealing trade-offs between execution time, energy consumption and chip area. Obviously, the area of the accelerator depicted in Figure 4.13a increases with the number of PEs. Looking at the simulated execution time, which is shown in Figure 4.13b, we see that running this workload entirely using the CPU requires 653 s with a CPU running at 40 MIPS. Adding even a relatively small 8×8 systolic array accelerator to the SoC significantly improves the inference time. With an increasing number of PEs and exploiting higher spatial parallelization this improvement grows. Due to increased data reuse inside the systolic array, less offloading per layer has to happen. However, we also observe diminishing returns when increasing the systolic array dimensions, since some important parts of the CNN workload are handled exclusively by the CPU like data movement or the im2col transformation. One drawback when increasing the number of PEs is not only an increased area but also a higher energy consumption. This effect is shown in Figure 4.13c. As the number of PEs and consequently data reuse increases, fewer energy intensive off-chip memory accesses are performed. However, this can only partly compensate for the static and dynamic energy demand of a larger number of PEs. As a consequence, a 16×16 array is more energy-efficient than 8×8 or 32×32 arrays. An 8×8 array, for instance, needs more off-chip memory accesses and a 32×32 array has more energy demand due to the higher number of PEs.

So far, we looked at the core of FLECSim and ran a simple example by sweeping just one parameter. The full potential of a simulation framework and DSE tool unfolds when we want to determine best fitting DNN accelerator for a given application systematically. To showcase this, we also performed a larger study that has the goal to make the design space of a systolic array for LeNet-5 tangible. In this experiment, we exhaustively searched the design space that is constructed by the systolic array dimension and the available memory bandwidth. An exhaustive search looks at all viable points in the design space but does not work in a goal-directed way. Hence, we also conducted a search using a GA. In particular using the Non-dominated Sorting Genetic Algorithm (NSGA)-II algorithm [308] provided by the Pymoo library [305]. The objective function of the GA tries to optimize all three performance metric at the same time. Thereby, the parameter space consists of the array dimension and the data bus width to the DRAM. These parameters are constrained to a range of 1 to 64. In addition, the bus width has to be smaller than the array dimensions.

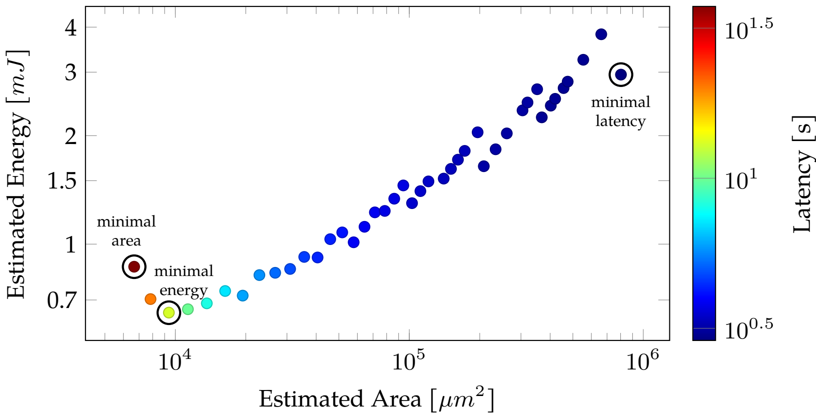


Figure 4.14: Design space exploration performed with FLECSim using LeNet-5 as workload. Solutions with minimal area, energy and latency are highlighted.

For our exhaustive search the design space has 216 configurations. All of them were evaluated. Among them, we selected Pareto optimal points as best potential accelerator designs. All Pareto points of the exhaustive search in terms of chip area, consumed energy and latency are shown in Figure 4.14. Here, we can pick three points of interest: configurations that optimize each performance metric. The configuration that yields the smallest area (6669 µm) is a 2×2 array with a 64-bit bus. Obviously, the fastest configuration (2.88 s) is a 64×64 systolic array, but it requires 802 734 µm of area, about 120× as much as the smallest accelerator. However, it consumes less energy compared to, e.g., a 62×62 array. This is due to efficient tiling of the layer shapes, which fit better on a 64×64 array. A configuration that performs an inference with the lowest power (0.64 mJ) can be found when using a 4×4 systolic array with a 128-bit bus. Similar to the Tiny-Darknet experiments, here the energy for memory accesses and the static and dynamic energy for PEs settle at a good energy efficiency.

When applying an GA to this problem, of course we got the same performance figures for a given problem, but we were able to find all needed Pareto much faster. Each evaluation takes about 3 h, leading to over 840 h in total when using exhaustive search. In our exemplary study with LeNet-5 running on a systolic array, we only needed to evaluate 65 designs, almost 70 % less compared to an exhaustive search. Figure 4.15 gives an overview of the design space that is explored when we use the

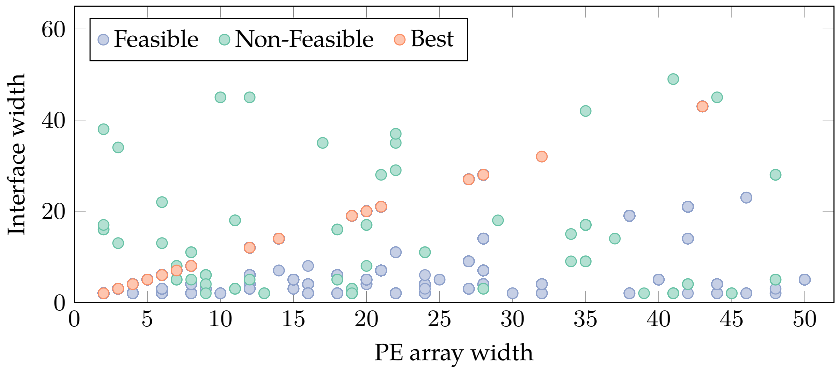


Figure 4.15: Pareto-optimal and best solutions for a LeNet-5 workload found by FLEC-Sim using an NSGA-II algorithm.

NSGA-II as optimization algorithm. In contrast to Figure 4.14 this figure shows the outcomes of different parameter combinations. All feasible solutions (●) are also Pareto optimal solutions, which deliver optimal solutions for the given objectives. Those are configurations in which the interface width is smaller than the array's dimensions. This can be explained by the utilization. A large and less utilized DRAM interface consumes a lot of energy and adds no performance to the system. The best solutions (●), i.e., a subset of all Pareto optimal solutions that maximize all objectives, are found when the interface bandwidth is equal to the dimensions of the array. Non-feasible solutions (●) are solutions that perform worse than feasible solutions and are generated by the GA on its way to optimal solutions.

4.5.3 Analytical Evaluation

Now, let us move over to our analytical model. As stated before, our main objective was to significantly reduce the simulation time and still deliver accurate predictions. Since the model is entirely implemented using Python, we can couple it directly with PyTorch and start an accelerator assessment. To figure out how accurate our model is, we compared it with a cycle-accurate execution similar to FLECSim and to the state-of-the-art work SCALE-Sim [277]. We picked 8×8, 16×16 and 32×32 as systolic array dimensions, as we target energy and area constrained applications like embedded systems. Baseline generation in a cycle-accurate manner is done using

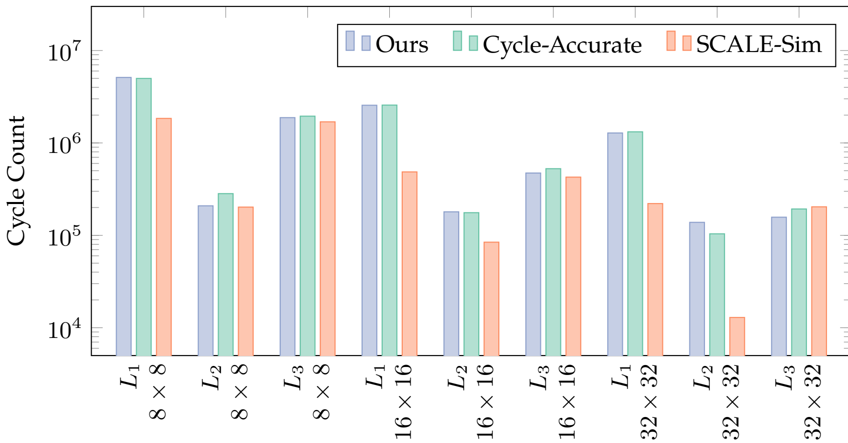


Figure 4.16: Evaluation of different layer configurations across different array sizes on a logarithmic scale. L_1, L_2 and L_3 represent 7×7, 1×1 and 3×3 convolution operations, respectively.

Gemmini and Chipyard. Since it is implemented in Chisel, the accelerator with its peripheral components can be compiled into a SystemC model using Verilator. Running the compiled SystemC model yields an accurate cycle count. All experiments, we performed with the analytical model, are executed on a single AMD EPYC 7702P CPU core running Rocky Linux, multiple cores may be used to run individual experiments in parallel.

Estimation Accuracy and Simulation Time

The estimated cycles and the required simulation time are provided in Figure 4.16 and Table 4.1, respectively. Both show a comparison of our analytical model, a cycle-accurate simulation of Gemmini and an evaluation using SCALE-Sim. The plot focuses on the first three layers of ResNet, since they contain different convolution kernel sizes: 7×7, 1×1 and 3×3. Whereas Table 4.1 also looks at an entire inference of ResNet-34. Moreover, the first three layers feature Max-Pool operations, which are usually neglected in other simulation environments, but can have a significant impact on the performance figures.

When looking at the three different array sizes and layers that are depicted in the plot, we can see that obviously the cycle count decreases with larger array dimensions.

Workload	This Work	SCALE-Sim [277]	Cycle-Accurate
L_1	1.8 s	165 s (55×)	9179 s (5099×)
L_2	0.17 s	16 s (138×)	2201 s (12 947×)
L_3	0.76 s	147 s (18×)	2667 s (3509×)
ResNet-34	28 s	1 h	> 48 h

Table 4.1: Simulation time comparison of our analytical model with cycle-accurate simulation and SCALE-Sim on a 16×16 systolic array.

Breaking the analysis down to individual layers, we observed large differences in the first layer (L_1) between a cycle-accurate simulation and SCALE-Sim. Our analytical model, however, delivers similar cycle counts to the baseline. This can be explained by the following: Layer 1 has, in contrast to the others, a Max-Pooling operation, which is not modelled in SCALE-Sim. Consequently, they are not accounted for in the cycle count. As such, the performance deviation between a layer with and without pooling can be significant. In our experiments, we observed a cycle count difference of 23 % when comparing layers with and without pooling. For this reason, including Pooling is crucial to accurately model performance. In addition, the underlying mapping also plays a role. SCALE-Sim works with a different mapping than the one that is used in Gemmini. In Gemmini, the first layer is unrolled over the number of input channels across the systolic array, which leads to a low utilization due to the low number of input channels. We are able to consider this effect in our flexible model, by setting the right parameters. As a result, our analytical model yields a more accurate cycle count during simulation than SCALE-Sim. Similar trends can be observed over different array sizes. Looking at the second layer, which has a 1×1 convolution operation, SCALE-Sim also returns a slightly inaccurate cycle count. The calculated values are too low, since SCALE-Sim assumes a higher bandwidth than what is actually available, leading to fewer stalls compared to a real-world inference. This effect becomes very visible in when a 32×32 systolic array is used. In case of 3×3 convolutions (L_3), all tools are able to represent the actual cycle count. This is also the most common type of kernel size in CNNs. However, for a proper assessment of a DNN accelerator, we definitely have to reflect all layer configurations that can occur in modern CNNs.

Besides the accuracy, the needed time to run a simulation is also an important metric. In case of a design space exploration, a faster evaluation of design points aids quicker design space search. Therefore, we again compared our approach with SCALE-Sim

Scratchpad memory \ Accumulator memory	64k	128k	256k	512k	1024k
256k	29.3	29.1	28.8	-	-
512k	28.8	28.0	27.9	27.8	-
1024k	28.4	27.8	27.6	27.6	27.7
2048k	28.3	27.8	27.4	27.3	27.3

Figure 4.17: ResNet-34 cycle count (millions) for different memory configurations.

and a cycle-accurate simulation using the same array sizes and ResNet layers. It has to be noted, that a cycle-accurate simulation of an entire ResNet-34 takes multiple days, which makes cycle-accurate simulation using Verilator infeasible for a DSE. Depending on the layer or workload, for our analytical model we found a speed-up of up to 12 947× and 138× compared to cycle-accurate simulation and SCALE-Sim. Moreover, we could observe a dependency of the simulation time and the simulated array size. This because, of course, a cycle-accurate simulation of a larger model takes longer. Our analytical tool mitigates this, since the Roofline model, that our model is build upon, is independent of the number of components in a simulation. As a result, our tool delivers results rapidly and finishes a simulation of individual layers in less than one second on average. It evaluated an entire ResNet-34 inference in 28 s, which is sufficient for a comprehensive DSE.

Impact of Memory and Array Size

We have shown that our analytical model quickly delivered the needed performance figures and allows for fast adjustment of different design parameters. In the following, looked at one of the largest challenges when designing a DNN accelerator, which is to provide sufficient local memory. On-chip memory is very expensive in terms of area, but increases data reuse and thus reduces the number of energy-intensive off-chip memory transfers. Hence, it is advisable to carefully choose the memory sizes to achieve a high efficiency. For our evaluation we assumed an off-chip memory with a fixed latency, since Gemmini has an L2-cache in between the off-chip DRAM and its local memories, making this memory hierarchy difficult to model. However, looking at the local memories is still very important, since they have a significant area impact.

Therefore, we ran ResNet-34 on a 16×16 systolic array using several memory sizes. The scratchpad and accumulator memory size was varied from 256 kB to 2048 kB and from 64 kB to 1024 kB, respectively. Configurations with larger accumulator than scratchpad memories are invalid in Gemmini and thus were not tested.

The impact of the different memory sizes is shown in Figure 4.17. In general, different sizes affect tiling of data across the scratchpad and accumulator, and as such has a minor influence on overall performance compared to the influence of a larger systolic array. Of course, larger memories have a great impact on area and energy, but we found that they only have a small impact on the performance in terms of total cycles, as long as the memory size sufficiently utilizes the accelerator. Considering a 16×16 array in which the memories are set to the largest configuration (2048 kB for scratchpad and 1024 kB for accelerator memory, respectively), we saw an area increase of $7\times$. However, at the same time we only gained 7.3 % in performance, in comparison to the smallest memory configuration tested. In comparison, increasing the array size from 16×16 to 32×32 with a fixed 256 kB scratchpad and 64 kB accumulator memory, only added 73 % area and significantly increased performance by 217 %. Due to the high area requirements, increasing the memory size might not always be the best choice for optimization. But with larger systolic arrays, larger memories may yield larger performance increases, since the available memory is better utilized for larger matrices. However, as a rule of thumb, especially in area constrained embedded designs, increasing the array size is the preferable choice to increase the performance.

Case Study: Design Space Evaluation of ResNet-34 on a Systolic Array

To showcase the insights our analytical model can generate, we deployed it for design space exploration and performed a small case study therewith. As workload, we analyzed ResNet-34 in an area constrained embedded environment. The objective is to minimize energy consumption, while maintaining a high performance. For this application we assumed a clock frequency of 700 MHz. In our case study, accelerators have to yield at least 30 FPS to be considered valid designs.

The DSE itself is limited though a set of architecture constraints. Besides the ones that are already imposed by Gemmini, we limited the array sizes of dimensions of 8, 16, 32 and 64. In terms of memories, we applied more loose constraints. Scratchpad memory sizes could be between 128 kB and 4 MB and accumulator memory sizes were allowed from 64 kB to 2 MB. As the optimization target, we chose to minimize the required energy.

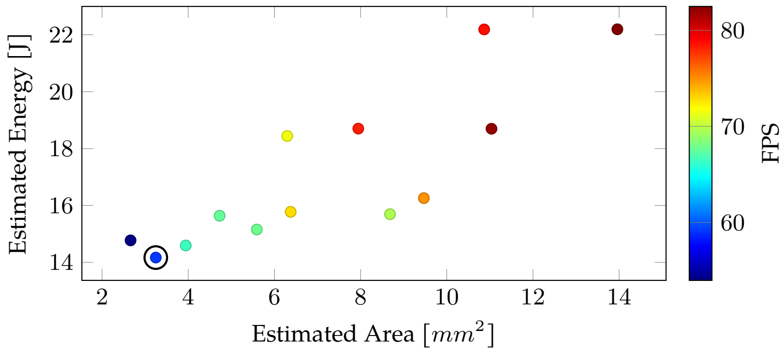


Figure 4.18: Pareto optimal array configurations running ResNet-34 with the associated FPS performance as well as area and energy requirements.

With the given constraints, the full design space consists of 57 points and 13 Pareto optimal points. Figure 4.18 shows the Pareto points of our evaluated design space with the corresponding required chip area, energy consumption and FPS. The latter is derived from the cycle estimate. The performance for different design points ranges between 34 and 117 FPS and from 53.9 to 82.5 FPS for all Pareto optimal solutions. From the plot, we observed gaps in the performance domain instead of a continuous trend. This is caused by Gemmini's architecture constraints. Since array sizes cannot be defined arbitrarily, we had to move, for example, from a 16×16 array directly to a 32×32, resulting in a large performance jump. In terms of array size none of the 8×8 array configurations satisfied the performance requirement of 30 FPS. In general, our evaluation showed similar to the memory evaluation, we performed before, that array size is the main driver for performance.

Considering our envisaged use case, we found an energy efficiency to performance sweet-spot with an array size of 32×32, scratchpad memory of 256 kB and accumulator memory of 64 kB. Notably is that the exploration did not immediately choose the largest array configuration, since it adds too much area to the design. Moreover, despite the findings from the memory evaluation, the sweet-spot solution did not take the smallest scratchpad memory configuration. A 256 kB scratchpad memory improved the performance to meet the design goals compared to a 128 kB scratchpad. As a result, for the found design configuration our analytical model estimated 3.25 mm^2 area and 14.17 J total energy consumption per inference. The design achieved a total performance of 59 FPS.

4.5.4 Case Study: Design of an eFPGA tile

To highlight that the two accelerator modelling tools support the design of all kinds of DNN accelerators, we performed an exemplary case study. In this, the goal is to design an embedded Field Programmable Gate Array (eFPGA) that is placed into a large SoC with different tiles. This large SoC is an endeavor pushed by the European project European Processor Initiative (EPI) to build a fully-European High-Performance Computing (HPC) chip. One of the applications that will later run on this eFPGA is a low-power CNN accelerator. The overall objective is to design an eFPGA tile so that it can fit the largest and most performant accelerator. However, the eFPGA tile itself is on a strict area budget that has to be met. Designing this eFPGA involves not only the DNN accelerator itself but also tools needed to generate the eFPGA matrix as well as diverse tools to determine area and power figures. Therefore, in the next few subsections, we will first introduce a highly configurable CNN accelerator design and the eFPGA tools. Then we move on to a discussion about how the algorithm accelerator co-design supports the eFPGA.

Accelerator Overview

To find a versatile eFPGA configuration, we choose a total of two distinct applications for our exploration. First the aforementioned CNN accelerator, which will also be the focus of this section, and a control-flow driven application, namely a CPU task scheduler called Picos [309] (More details can be found here: [Hot+22b]).

Our CNN accelerator is designed towards good scalability with an adjustable number of computation units, which enables a design space exploration. An overview of its design is given in Figure 4.19. The computational intense convolution operations are tackled in microkernel units. Their design and the used buffering scheme are inspired by a work from Qui et al. [61], which is chosen as basis, since they show great results using dynamic quantization for activations and weights with neglectable accuracy loss. The original accelerator achieves an efficiency of 14.22 G OPS/W with an accuracy loss of 0.04 % on a Xilinx Zynq-7000 device. We extended their approach for our case study with a coarse grained sparsity detection methodology, which reuses weights and input features. Input feature data is provided as rows and gets stored into a data line buffer, which aligns data for typical convolution kernel sizes like 3×3. Corresponding weights for the filter are kept inside a large scratchpad memory, which returns the required column of weights for each channel to the microkernels. General aspects of a neural network might be configured dynamically during runtime

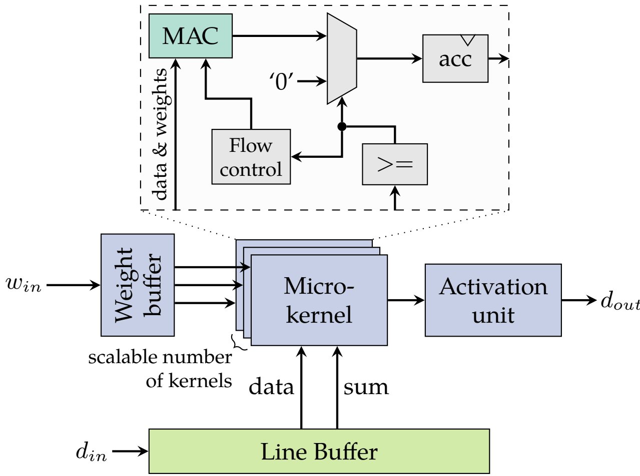


Figure 4.19: Overview of the CNN accelerator architecture.

like workload dimensions, while others such as bit-widths or buffer sizes require a reconfiguration of the eFPGA target platform with a newly generated accelerator.

As stated before, our architecture can exploit the large sparsity that is present in most intermediate feature maps. Therefore, it monitors the flow of incoming data and continuously computes the sum of a filter. Based on the derived sum, it is possible to apply magnitude pruning to sparse input blocks in the input feature map. If we detect a sparse block, associated MAC operations can be skipped. The corresponding decision is made by the microkernel itself through comparison of the kernel sum with a predefined threshold. In case of a skip, the control flow module tells the line and weight buffer to omit the current column of data and to jump directly to the next one. At the same time, it outputs zero as result through a multiplexer to the accumulator. To process the next filter in a layer, this accumulator can also be preloaded.

One of the largest points to tweak area and energy are the actual number of MAC units per microkernel. Considering a latency of one cycle, nine MACs in parallel can compute a 3×3 convolution each cycle, but only three increase the latency to three cycles. Using fewer MACs units creates an artificial bottleneck, which we can mitigate by skipping sparse blocks. Besides that we can obviously change the size of buffers and the total number of microkernels. In addition, we can also consider uniform quantization. Therefore, we can adjust the bit-width of weights, inputs and outputs.

Workload and Parameter Exploration

As described before, one of the CNN accelerator's objectives, sitting in the eFPGA, is to run DNN applications using only little energy. The envisaged use-case is face recognition. Therefore, we chose SqueezeNet [41] as base topology for the parameter exploration, because of its low parameter count and small memory footprint. Picking a network topology already specifies some design characteristics of the accelerator. For example, the kernel size is 3×3 and the input buffer size is set to 224, which is the dimension of the largest input feature map. The buffer design and the MAC units' design are also highly dependent on the width of the operands and accumulators. With linear uniform quantization, which was often proved to allow for significant bit-width reduction with only small precision loss [310], we can reduce the bit-width. To apply quantization in the most effective way, we used SqueezeNet and trained it from scratch using the CASIA-WebFace dataset [311]. From a baseline accuracy of 86 % using 32-bit floating point, we were able to quantize the network using 10 bit operations and still achieved an accuracy of 85 %. Using 8 bit we still observed an 81 % accuracy. An 8 bit design, moreover, lowers the memory impact of an inference from 480 kB down to 135 kB. As a result, we chose an 8 bit version of SqueezeNet for our accelerator evaluation and can thus set the MAC inputs and accumulator width to 8 bit and 16 bit, respectively. For the later eFPGA design, all MAC operations will be computed in Digital Signal Processor (DSP) cells. It has to be noted though, that DSPs are very suitable for all dataflow driven applications but might not be the most important eFPGA resource for control-flow applications. Hence, we have to keep in mind to not focus only on DSPs, but also to leave spare eFPGA resources for other workloads like control-flow or memory-intensive ones.

Now that the parameter space of our CNN accelerator and the eFPGA is sufficiently constrained, we can move on to the actual parameter exploration. Figure 4.20 gives an overview of our design flow that is applied to obtain the eFPGA specification. First, the model is quantized to reduce its memory footprint. Then, we can utilize FLECSim to generate and explore all remaining architecture parameters and their influence on the performance and latency. With the parameters in place, we can in the next step generate an architecture, which is synthesized using the eFPGA tools provided by the company Menta[10]. Here, we can experiment with the number and distribution of eFPGA building blocks and how they impact the performance. For example, DSPs may be replaced with memories and the number of Look Up Tables (LUTs) can be adjusted.

[10] https://www.menta-efpga.com/origami-programmer

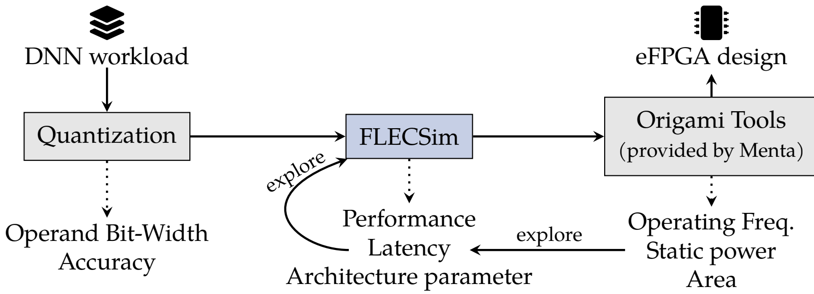


Figure 4.20: Overview of our architecture exploration design flow to design an embedded FPGA (eFPGA) tile. First the DNN workload is quantized, and then architectural parameters are explored using a combination of FLECSim and the design tools for the eFPGA.

To assess their impact, the synthesis tools yield important characteristics like area, static power consumption and maximum operating frequency. With both FLECSim and the tools by Menta in combination, we can eventually provide the necessary PPA figures to evaluate, explore and assess different architectural characteristics.

Exploration Results

With the previously described architecture exploration flow, we can systematically optimize the target eFPGA design to meet the requirements of our two applications. Thereby, both applications have to fit in an eFPGA design that consumes at most 1 % of the total chip area for the future EPI chip. The task scheduler application has to have a high number of memories and the CNN accelerator requires as many DSPs as possible. However, the eFPGA should always be capable of hosting future, yet unknown applications.

Our found eFPGA architecture is detailed in Table 4.2. The upper half of the tables shows the performance and the bottom part available eFPGA resources. With consideration of the 1 % area constraint, in a square shape, we can fit an overall of 38 columns and 40 rows of eFPGA tiles. Four of those columns are DSPs and four are memories. The eFPGA layout is shown in Figure 4.21 with DSPs highlighted in teal and memories in yellow. DSPs and memories are intentionally evenly distributed over the eFPGA columns to allow for the highest routing flexibility and have LUTs and Flip-Flops close by.

Parameter	Value	Utilization	
		CNN Acc	Picos [309]
Technology	7 nm	-	-
Matrix shape	38 × 40	-	-
Static Power	1.7393 mW	-	-
Operating freq. [MHz]	-	159	100
LUT6	9632	78 %	51 %
Flip-Flops	12 086	72 %	29 %
DSPs	52 × I16_32P	92 %	0 %
Memories	80 × 2Kx16	5 %	46 %
Inputs, Outputs	1242, 1244	-	-
Clocks, Resets	2, 2	-	-

Table 4.2: Overview of the dimensions, building blocks and static power of the found eFPGA design. Including the utilization and operation frequency of the two applications.

The total of 52 DSPs and 80 memories blocks offer enough resources for a wide range of applications including our two applications used for architecture exploration. Considering the CNN accelerator, we can host 16 microkernels, which utilize 48 DSPs. All buffers can be placed in fraction of the available memory cells, leaving enough room for additional buffers that might be needed to handle backpressure to outside connections. When the eFPGA runs at the maximum operation frequency of 159 MHz, we can achieve a full utilization of our accelerator when the available bandwidth delivers 159 MB s^{-1}. Since most computations in a CNN are independent of each other, different microkernels can operate on the same input data but with a different set of weights. Refreshing the associated weights for the convolution operation adds on average 1 MB s^{-1} to this bandwidth. Due to the availability of a high-performance NoC in the EPI chip and the high number of eFPGA I/Os, we can project to satisfy this bandwidth requirement. However, at the point of writing this thesis, the chip is not taped out, and we had to rely on simulation results. Putting the results together, we can compute a forward-pass of the quantized and optimized SqueezeNet for face detection in 150 ms with a performance of 0.48 G OPS/s. When we take the static and dynamic power of in total 4.8 mW into account, we reach an efficiency of 99.375 G OPS/W.

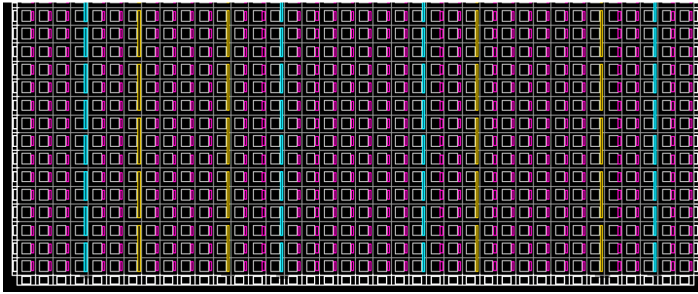


Figure 4.21: Architecture segment of the eFPGA configuration, showing DSPs, I/Os and memories highlighted in teal, white and yellow, respectively.

4.5.5 Discussion

During the evaluation we looked in-depth at the performance figures of our two simulation tools. To get a better impression of how these works contribute to the current research, we have to put them into perspective. FLECSim is designed to be a very extendable and flexible tool that can handle a wide range of accelerator designs. Therefore, we went for the most versatile simulation approach, which is cycle-accurate SystemC simulation. However, this accurate method comes with the drawback of being fairly slow when running an entire simulation of a recent DNN model. A comparison with other works in this domain is shown in Table 4.3. To mitigate the long simulation time, FLECSim uses TLM to model data movement and an ISS for the CPU. In contrast to works like PARADE [293] this increases the simulation speed, but does not reduce the simulation precision of the DNN accelerator. Since FLECSim offers a CPU simulation, we have the advantage of running versatile workloads and

Property	MagNET	STONNE	SMAUG	FLECSim
Flexible design space	✗	✓	✓	✓
CPU simulation	✗	✗	✓	✓
Software Co-Simulation	✗	✗	✗	✓
Automated exploration	✓	✓	✓	✓
End-to-end evaluation	✓	✓	✓	✓

Table 4.3: Comparison of cycle-accurate simulation frameworks for DNNs.

are not limited to just one kind of accelerator like MagNET [272] or STONNE [274]. In addition, we can interface any DNN framework, either using an ONNX description or through implementation of the corresponding drivers for, e.g., PyTorch to FLECSim. This makes FLECSim also more flexible in comparison to other state-of-the-art works. Similar to FLECSim, SMAUG also focuses on an end-to-end evaluation of SoCs with DNN accelerators. In contrast to our work, they use gem5 as simulation framework instead of bare-metal SystemC. In fact, gem5 can make some parts of the SoC design simpler, but also allows fewer degrees of freedom. For example, direct integration of RTL models via Verilator can become challenging, and so does early validation of RTL designs. In summary, SMAUG and FLECSim play in the same league regarding performance and flexibility, but both use slightly different tools and methods to eventually reach the same goal.

However, one of the best methods to accelerate the performance assessment of DNN accelerators, is to get rid of cycle-accurate simulation altogether. We can do this, because DNN workloads have a very homogenous dataflow. Once a mapping is figured out, a well-chosen set of equations can adequately describe the required computational cycles, memory accesses and bandwidth for a given DNN workload. This is exactly what analytical modelling approaches are trying to do. Our analytical model is no different. Using a set of sophisticated equations and the Roofline model it estimates the performance, energy and area of a systolic array that processes a DNN workload. In contrast to SCALE-Sim [277], which does this estimate by means of cycle-accurate prediction, our approach is thus much faster. Works like ZigZag, Interstellar and Timeloop, however, follow a similar approach and introduce modelling languages that can describe the workload and the DNN accelerator. Then they perform a mapping and predict the needed compute cycles through an analytical model. But, the design methodology and framework they use, often looks only at MAC operations as they are the major part of DNN computation. However, this neglects operations like activation and Pooling, and consequently those models tend to estimate inaccurate performance figures for accelerators that do these operations during the computation of MAC operations. For such accelerators, like Gemmini, we observed a deviation of up to 23 %. Our model includes those operations and delivers estimates that closely match a cycle-accurate RTL simulation. A quantitative comparison of the other works with our analytical model is summarized in Table 4.4.

With various benchmark DNN workloads and two case-studies we showed the feasibility of our two DNN accelerator performance assessment tools. Intentionally we chose to first explore a cycle-accurate and then an analytical approach. Our analytical

Property	SCALE-Sim	MAESTRO	Timeloop	ZigZag	Ours
Flexible design space	✗	✓	✓	✓	✓
Simulation speed	-	o	-	o	+
End-to-end exploration	✗	✓	✗	✗	✓
Metrics	cycles	PPA	PPA	PPA	PPA

Table 4.4: Comparison of analytical DNN performance models.

model shows in comparison to the current state of the art, a very accurate and quick performance prediction, which makes it usable for large-scale parameter exploration. With a well-constrained set of design options, we can then start our cycle-accurate approach FLECSim, to get highly precise PPA figures. Although the latter takes relatively much time, considering a manageable design space it is still possible to run a cycle-accurate simulation. The key to achieve a both accurate and fast DSE ultimately lies in combination of accurate, but slow cycle-accurate simulation and fast, but more imprecise analytical modelling.

4.6 Conclusion and Outlook

For many applications like embedded systems DNNs have to run highly efficient to meet the design requirements. In the previous sections, we saw that tailored hardware accelerators are necessary achieve this. However, only taking the right design decisions leads to an efficient DNN accelerator. Taking those decisions requires knowledge about how they influence the performance metrics of the accelerator. Hence, simulation environments to systematically explore and assess various design configurations are crucial to obtain the most performant accelerator for a given application.

Besides the large body of research that was already carried out, we introduced two concepts for such simulation tools. First, a cycle-accurate simulator, called FLECSim, that can model entire SoC designs including DNN accelerators. We put a strong focus on making FLECSim very easy to extend, so that new DNN accelerator models can be integrated in straightforward a manner. Our cycle-accurate tool allows for end-to-end evaluation of the aforementioned design decisions in terms of energy, performance

and chip area. Using an exemplary workload, we could demonstrate how major architectural aspects like memory sizes and the number of PEs influence the efficiency. The flexibility of FLECSim is highlight by a case study that we conducted. Here, we laid out an accelerator that is hosted on a configurable eFPGA.

Since the assessment of DNN accelerators with cycle-accurate simulators can consume a lot of time, we proposed a second approach that harnesses the homogenous dataflow of DNN workloads. This second method is an analytical model that describes the workload and the accelerator with all their parameters using a set of equations, which, e.g., directly yield the cycle count or the number of memory accesses. Our analytical model can generate those numbers for an arbitrary DNN workload, which runs on a configurable systolic array, orders of magnitude faster than a cycle-accurate simulation. Thereby, we demonstrated that our model is more accurate than other state-of-the-art works and less than 1 % off compared to a cycle-accurate simulation.

The full potential of our simulation tools unfold, when combining both works, ultimately making a DSE of DNN accelerators feasible. The huge design space that can be constructed from all the various design options in DNN accelerators, is first explored by our analytical model, either with optimization algorithms or using exhaustive search. Consequently, this leave us with a reduced set of potential optimal DNN accelerator configurations for a given set of constraints. These remaining configurations can then be evaluated with FLECSim to get reliable numbers.

However, the research field of DNN accelerator modelling is by far not completed yet. The topic still attracts a lot of attention, since the earlier and the easier a design decisions can be made, the more money can be saved in the design process. Most works, including our two solutions, have design limitations and still cannot reflect all the various design decisions that can be made. In addition, our works have not unleashed the full potential of an automated DSE. Both works have mainly focused on an accurate and fast model. A DSE layer that uses RL instead of GAs or exhaustive search, may yield better performance, similar to NAS approaches, in which RL is established already today. In general, making even the analytical model more efficient and faster opens up a possible combination of DSE and NAS. Both are closely related, as some model topologies might work best on a dedicated hardware architecture. Some few works have already explored this, showing promising results.

Chapter 5

Leveraging and Increasing Regular Sparse Activations in CNNs for Energy Efficiency

In the previous sections, we have noted that one of the key challenges in computing modern Convolutional Neural Networks (CNNs) efficiently lies not only in the vast number of computations, but also in the corresponding numerous memory transactions. Since memory requirements of modern networks exceed the capabilities of local memories, data has to be offloaded, causing unwanted energy and latency intensive off-chip memory transactions. This chapter presents an approach that aims to reduce those off-chip memory transaction through coarse-grained regular blockwise pruning. In contrast to traditional pruning methods, our method follows a hardware-centric design, which allows us to prune sparse blocks with less hardware overhead. To realize this, we first introduce a tool, Spex, that maximizes the number of regular sparse blocks in each layer. Second, we present Sparse-Blox, our hardware extension that harnesses found sparse blocks for memory and Multiply-Accumulate (MAC) operation reduction. Our hardware extension works with most common Deep Neural Network (DNN) hardware accelerator architectures. Sparse-Blox thereby adds $5\times$ less area to the design than state-of-the-art accelerator extensions that operate on irregular sparsity. With Spex, we can reduce the number of memory transfers in ResNet-50 and Yolo-v5s by 18.9 % and 12.6 % with a 1 % or 1 mAP accuracy drop, respectively [Hot+23a].

5.1 Overview, Introduction, and Motivation

CNNs have been successfully deployed in many domains and became ubiquitous in many applications. While CNNs deliver tremendous results in image segmentation or classification tasks, their computational complexity and memory requirements grew massively in the last few years. This poses a major challenge to hardware and systems architects, and an end of this trend is not foreseeable. Yet unsolved problems like autonomous driving might even worsen this challenge. As a result, tailored accelerators have established themselves, which add multiple optimization techniques like quantization and pruning to address the overwhelming energy and latency constraints. However, combining high throughput and low-energy consumption is still a major challenge yet to be solved.

In Subsection 2.5.2 we discussed that modern CNNs are very data-hungry and large intermediate results have to be buffered during inference. Since, even state-of-the-art silicon technology is incapable of hosting such large local memories with a reasonable area investment, data is offloaded to cheaper but slower off-chip memories, like Dynamic Random Access Memory (DRAM). This offloading from local Processing Elements (PEs) to off-chip memories is one of the largest contributors to the energy consumption [115] of CNN accelerators. Most offloading happens due to the large intermediate feature maps present in CNNs.

Nevertheless, due to the Rectifier Linear Unit (ReLU) activation function, which sets all negative activations to zero, CNN feature maps show large amounts of sparsity [312]. Sparse values that are present in output feature maps of different layers of ResNet-50 [40] are shown in Figure 5.1. The shown data was generated using the entire ImageNet-1K validation dataset, consisting of 50 000 samples. We can see that the first convolution layer together with the ReLU activation function already yields about 15 % sparsity in its output feature map. Deeper into the network, after the first to basic blocks (see Figure 5.1b and Figure 5.1c), we observe that the share of sparse values is growing. Also, for other networks, like ResNet-18, studies have found that feature maps have 20 to 80 % sparse activations [313].

If we take a more profound look at the output feature maps of CNNs, we can see that sparsity is not present in a regular form. Usually, sparse values are spread over all dimension and obviously vary based on the network input. As an example for two different inputs, this behavior is shown in Figure 5.2. The figure shows the first eight channels of the output feature map of the first convolution layer, as well as of the first

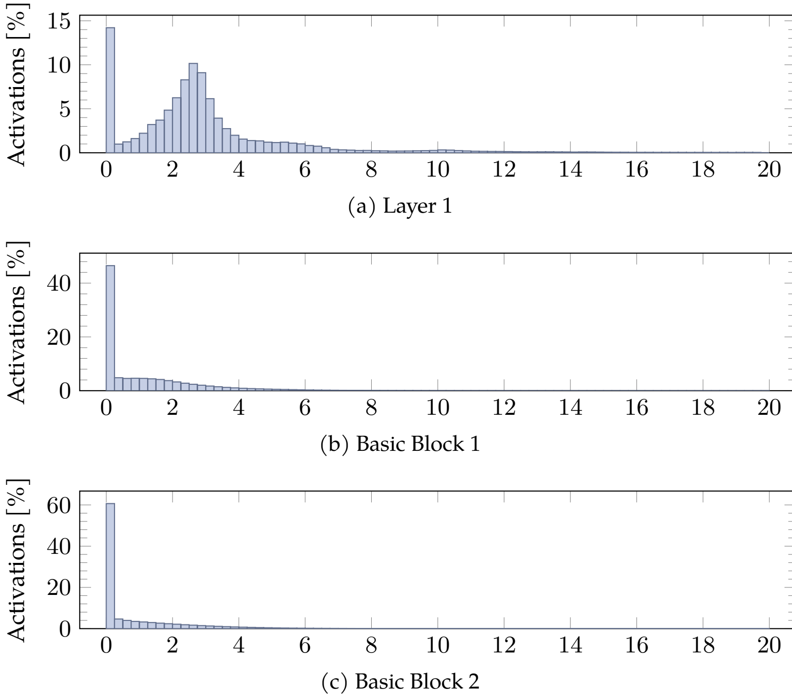


Figure 5.1: Distribution of the output feature map activations of different layers of ResNet-50. The histogram shows 100 bins from 0 to 20, values above 20 are clipped into the last bin.

ResNet basic block. Values that are exactly zero are marked in black, small activations in purple and large activations in red. Besides scattered sparse values, however, some filters show an especially large sparsity, sometimes even in contiguous areas.

Activation pruning exploits this sparsity, saves many unnecessary operations, reduces the memory footprint and finally decreases expensive off-chip memory transfers. However, pruning irregular sparsity dynamically during inference is a non-trivial task. First, non-sparse values have to be found in the large output feature maps. Afterward, the exact position has to be encoded efficiently to achieve a high compression ratio. Consequently, DNN accelerators that implement such pruning mechanisms, e.g., tend to have a comparably large hardware overhead due to the element indexing and compression mechanisms.

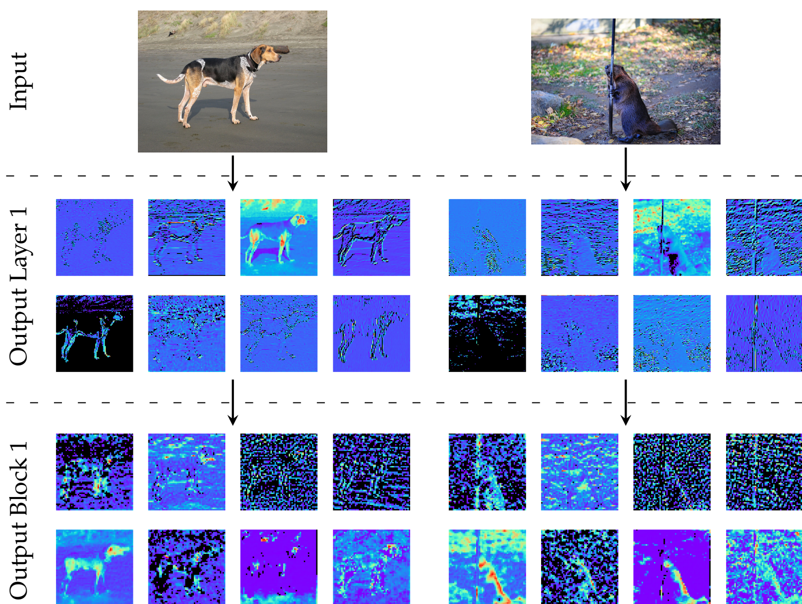


Figure 5.2: Output of the first layer and first basic block of ResNet-50 for two exemplary inputs. For each output the first eight channels are shown. The magnitude of activations is shown in color from blue to red. Exact zero values are depicted in black.

From a hardware perspective, recent works have proven that pruning approaches for regular sparsity are easier to implement with less hardware cost [314]. Especially, blockwise sparsity can considerably increase the energy efficiency compared to fine-grained pruning methods [263] To leverage regular activation sparsity in CNNs, this chapter proposes two components: First our DNN exploration tool *Spex* that increases regular activation sparsity in CNNs by determining a set beneficial of thresholds for each layer before deployment to the hardware. Second our DNN accelerator extension *Sparse-Blox*, that enables resource efficient pruning of activation pruning, previously found by Spex. Figure 5.3 gives an overview of how Spex and Sparse-Blox work together to enable a blockwise activation pruning mechanism. Our approach divides each activation feature map into blocks, whose dimensions match the underlying hardware accelerator. Then, the sum of each block is compared to a threshold upon it is decided whether a block can be pruned or not. Higher thresholds will prune more

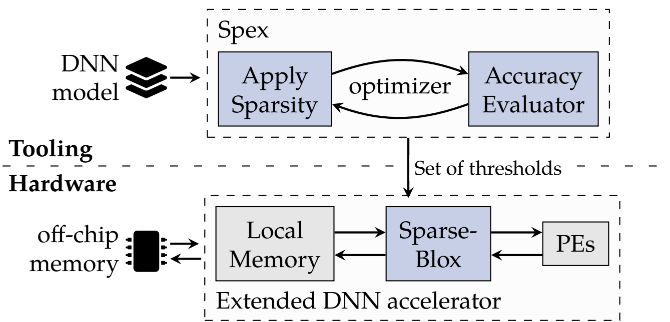


Figure 5.3: Overview of Spex and Sparse-Blox: A tool to increase and identify coarse-grained sparse blocks in CNNs and a low-overhead hardware extension to prune these blocks.

values, which may result in network accuracy degradation. This complex design space that is spanned by the correlation of pruned blocks and network accuracy is explored automatically by Spex. A found combination of thresholds for each layer is then used in hardware by Sparse-Blox to efficiently prune entire blocks of low activations. This makes compression of sparse values and skipping of MAC operations inexpensive in hardware and saves off-chip memory transfers.

5.2 Principle of Sparse-Blox and Spex

Sparse-Blox and Spex enable systematic exploitation of structured sparsity that occurs during DNN inference to save unnecessary computations and data movements. Substantial amounts of sparsity can be found in CNNs, due to the fact, that the filters are trained to detect a broad range of features in images, which might not all be present at the same time. In this case, most filters return negative activations, which become zero after applying the ReLU activation function. In the next layer, we can prune these zero values by skipping the corresponding MAC, since the result is already known to be zero. Moreover, if sparse values are compressed, we can reduce the memory footprint of the CNNs and finally even lower the number of off-chip memory transfers, saving additional energy and processing time. As we have shown in the motivation section, up to 80 % of all activations are zero in recent CNNs [280].

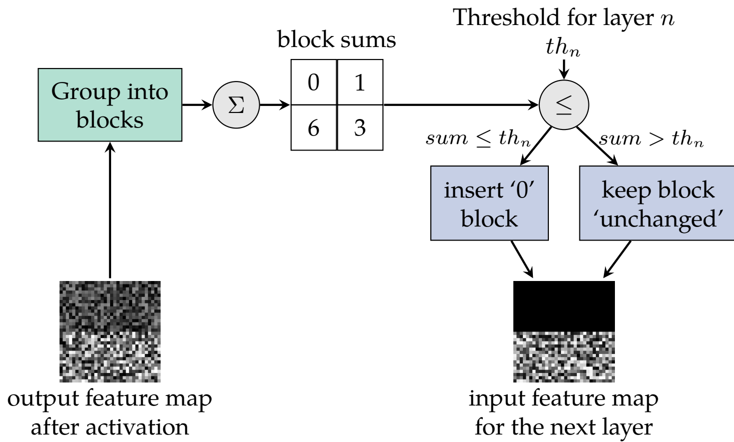


Figure 5.4: Principle of our blockwise regular pruning approach: An output feature map is divided into smaller blocks that match the hardware accelerator's dimensions. These blocks are accumulated, and the sum is compared with a previously determined threshold that determines whether a block get pruned or not. This increases easy-to-prune regular sparsity in CNNs.

Especially, regular sparse patterns support compression and computation reduction. Our hardware-centric approach enables inexpensive exploitation of structured sparsity in the form of blocks or squares in input feature maps. For example, if a full 8×8 block of sparse values is present in an output feature map, we can skip the computation of this 8×8 block completely on a systolic array that has the same size. In addition to that, we can compress this block in a straightforward manner by simply pooling each sparse block. As a result, we do not need to store coordinates for each non-sparse element, like Compressed Sparse Column (CSC) or Compressed Sparse Row (CSR) approaches (see Subsection 3.3.3), but instead we can now store the coordinates of entire blocks at once.

The general principle of our method that prunes sparsity in the form of squares or blocks, that match the underlying hardware accelerator, is given in Figure 5.4. In our case, output feature maps of a Convolutional (CONV) operation are either kept or transformed into a two-dimensional matrix form, which is achieved by the im2col transformation. Each activation feature map is then divided into smaller blocks that match the target hardware accelerator, e.g., 16×16 blocks for 16×16 systolic arrays. All elements in those blocks are then accumulated, to check the presence of a sparse

block. However, as we have seen before, sparse values are usually present in irregular patterns. Hence, entire sparse blocks are rarely present in out-of-the-box CNNs. To increase the number of sparse blocks, our method introduces a threshold that is compared to the sum of each block to determine whether a block can be pruned. If the block sum is less than the threshold, all values in a block are considered sparse.

However, the magnitude of thresholds has, of course, an impact on the network accuracy. Therefore, we present as the first part of our approach the exploration tool *Spex*. It searches for thresholds that achieve the best trade-off between activation compression, computation reduction and network accuracy impact. This search has to be performed dynamic, since patterns of sparsity in feature maps are not static but are highly dependent on the network's input. Hence, Spex runs before the network is deployed to an actual hardware accelerator and tunes the thresholds with a large test dataset. Spex can determine individual thresholds for each layer. This enables us to manage pruning-sensitive layers more carefully, while robust layers can be pruned further to optimize the computation and memory savings.

Once a beneficial set of thresholds is found, we can deploy the network to a hardware accelerator that has been extended with our low-overhead hardware extension *Sparse-Blox*. Sparse-Blox reproduces the same behavior as Spex. During inference, it computes the sum of a block, checks whether it can be pruned and compresses sparse blocks. Accumulation, comparing to a threshold and pruning of blocks can be easily implemented in hardware by only adding a small overhead. To reduce the number of off-chip memory transfers and MAC operations, Sparse-Blox buffers sparse blocks found in a layer until the computation of the next layer. Therefore, it features a small cache that stores only the position of a sparse block and not all individual elements of this block. Consequently, sparse blocks do not need to be transferred to off-chip memory. Instead, when the DNN accelerator tries to attain this block, which was found sparse in the previous layer, the computation can either be skipped entirely or a block of zero can be loaded rather than loaded from the main memory.

Sparse-Blox along with Spex aims to boost the performance and energy efficiency of CNN inference tremendously. Thereby, it adds only a small fraction to the hardware area. The next two chapters will describe the two components of our approach in-depth. Followed, by the results in terms of energy and computational savings.

5.3 Related Work

A large body of research was conducted regarding pruning using different methods and various kinds of sparsity. In general, sparsity in DNNs can be grouped into structured and unstructured or irregular sparsity and can be applied on weights (W) and input activations (IA). Especially, pruning and sparsity exploitation of weights and activations with small magnitudes has been addressed in several studies. Compared to the state-of-the-art section, we will focus in this section only on works that investigate structured sparsity. Also, we will look specifically at sparsity in CNNs. Nonetheless, it is worth mentioning that apart from CNNs, several studies have also showed positive effects of Leveraging sparsity in Recurrent Neural Networks (RNNs). For example, Narang et al. [315] performed blockwise pruning during the training of RNNs, resulting in significantly fewer parameters and noticeable speedup on Graphics Processing Unit (GPU). From the tooling and training perspective, Jiang et al. [312] present their regularization method, which looks at CNN channels. With their approach, they can prove that sparsity in CNNs can be largely increased and grouped into regular forms. On Resnet-50 with ImageNet, they demonstrate a 51.07 % reduction in Floating-Point Operations (FLOPs) with only 0.76 % accuracy degradation.

Starting with improvements on Commercial-Off-the-Shelf (COTS) systems like Central Processing Units (CPUs) or GPUs, Liu et al. [316] exploit sparsity in CNNs for classification and object detection and report significant speedups and better efficiency in inference. Lin et al. [317] also proposed an approach prune CNNs to speedup inference on general-purpose CPUs. Their framework groups N consecutive output kernels with the same input channel into blocks and removes those considered insignificant. According to these results, weight sparsity exploitation leads to a 56 ms inference speedup, but also to reduced accuracy compared to applying general weight pruning. Zhou et al. [318] presented in 2021 their approach to learn fine-grained sparsity patterns of NxM blocks to maximize the compression ratio and the performance increase on GPUs. They divided the number of operations and parameters in a ResNet-50 by a factor of four, with only 2 % accuracy drop. TETRIS by Ji et al. [319] aims to find the best possible trade-off between sparsity, hardware utilization and accuracy of networks running on GPUs. According to their evaluation results, this can be achieved by reducing the irregularity of a matrix by clustering unimportant elements together for structured pruning algorithms. However, TETRIS adds overhead by rearranging and grouping feature map elements during runtime. Moving from GPUs to Field-Programmable Gate Arrays (FPGAs), Zhu et al. [314] ex-

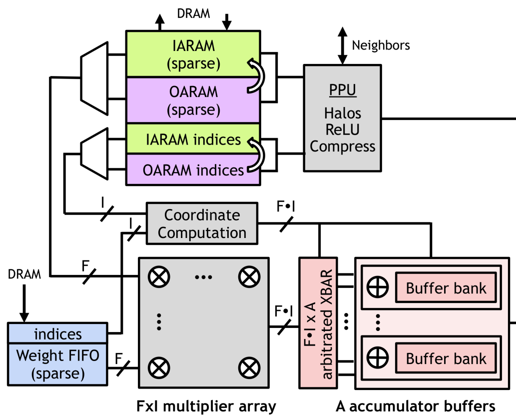


Figure 5.5: Overview of PEs of the accelerator SCNN, proposed in [151], that support sparse computation for weights and activations.

ploit structured sparsity with their FPGA-based accelerator, showing that structured sparsity is much more hardware friendly than irregular sparsity, especially when weights are trained to show structured sparsity.

As we saw, a lot of research was carried out for various COTS platforms. But, the most interesting research was conducted on tailored DNN accelerators, as they allow for the most design flexibility. To efficiently make use of sparsity in both weights and activations, the hardware has to support runtime adaptive pruning. Sparsity in activations can be exploited for power saving through gating or stalling of multiplications with zero. Kim et al. [260] demonstrate a 6× speed-up on VGG-16 by skipping computations of sparse elements in input feature maps in hardware, which outperforms Eyeriss [60] by 9× However, their approach does not save memory transactions.

To save memory bandwidth, compression techniques that pool zero activations have to be applied. Works with such modules are, for example, Cnvlutin [261] or the Cambricon series [263], which were introduced in the state-of-the-art chapter. Both show large performance advantages over the state-of-the-art works. Especially, Cambricon-S shows a significant improvement using regular sparsity in form of 2×2 blocks. Although those works show great improvements on the memory and computational footprint, they have to add hardware resources to realize the compression.

Index-based compression enables significant energy saving and reduces the number of computations. One example for this is SCNN by Parashar et al. [151] from 2017. An overview of their PEs is given in Figure 5.5. Their architecture uses a novel index-based compression scheme, which is more efficient than CSR or CSC format, and significantly reduces storage requirements. Compared to by then state-of-the-art works, they apply compression to both weights and activations. To encode this, they added extra modules to the architecture just before the PEs. In total, their encoding mechanism adds 33 % extra area. In their in-depth evaluation, they can show great improvements on the energy consumption. For example, with 90 % sparsity, only 6 % the energy of a dense accelerator is required. However, a full dense network running on their accelerator, needs 33 % more energy. Nonetheless, their evaluated image classification workloads, typically show large sparsity. Hence, they achieve up to 3.52× performance improvement on a VGGNet. Thereby, they demonstrate an increase in energy efficiency by 2.3×. In summary, they report an overall performance increase using their architecture when the workload is less than 85 % dense.

Looking at regular coarse-grained pruning in the form of blocks, Zhou et al. [263] presented Cambricon-S in 2018. With their approach, the authors aim to address one of the major problems in sparse DNN accelerators: irregularity of the sparse activations. Either the indexing and compression of the irregular non-sparse elements consumes a lot of energy, or it adds heavily to the area. In contrast, Cambricon-S prunes contiguous blocks of sparse activations at once. For CONV layers four dimensional and for Fully Connected (FC) layers two-dimensional blocks are used. Therefore, the authors experiment with various block sizes and report the corresponding compression ratios. To increase the efficiency further they add local quantization. For evaluation, Cambricon-S is implemented using a 65 nm technology node. With 16×16 PEs the authors report a total area of 6.73 mm^2, of which 15.55 % account the for compression and pruning modules for weights and activations. Those modules, feature sophisticated non-sparse masking methods, which enable online pruning, help to reduce the inner bandwidth, and even exploit some remaining irregular sparsity. Compared to DianNao [143] Cambricon-S achieves a 12.3× better energy efficiency and a 13.56× better performance on CONV layers across a selection of eight workloads, respectively.

STICKER by Yuan et al. [279] from 2020 follows a new route to exploit a high degree of sparsity in weights and activations. A high-level overview of their architecture is given in Figure 5.6. Their main driver to increase the efficiency is a multi-sparsity control and dataflow module that allows to switch between nine different sparse

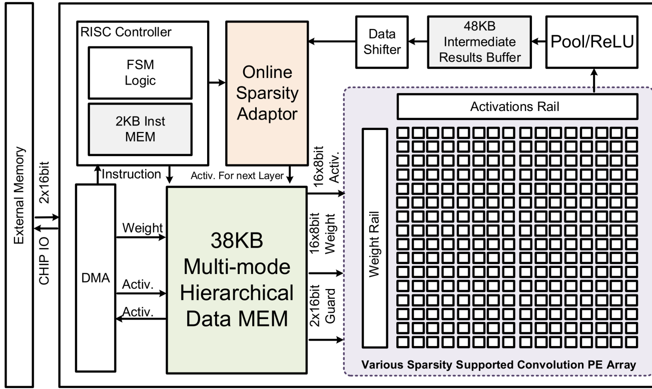


Figure 5.6: Overview of STICKER's architecture, a DNN accelerator that support online pruning; visualization taken from [279].

operation modes. In more detail their online adaptor detects the amount of sparse data in weights and inputs and switches between four operation modes: dense, medium dense, sparse, and all-zero. This operation mode is determined individually for small chunks or blocks of data. In case of dense data, no compression happens, and no extra energy is invested for that. When the data block is sparse, index-based encoding is applied to benefit from compression. Like our proposed method, they can also work with entirely zero blocks and skip them. However, they do not apply any method to increase those sparse blocks. To make the operation mode switching work, they also propose a reconfigurable memory hierarchy and multi-sparsity convolution PEs. The authors taped out a chip with 16×16 PEs and 170 kB on-chip memory to demonstrate their architecture. Using an AlexNet as workload, they can show an energy efficiency of 2.82 TOPS/W, which is a 1.8 $\times$ improvement to the state of the art. Finally, they also evaluated workloads with various degrees of sparsity. They found that their architecture reaches and energy efficiency of 62.1 TOPS/W at a sparsity of 5 % For their unit that enables online operation mode switching, they report a total area share of 5 %.

Another notable accelerator that exploits unstructured sparsity is SNAP by Zhang et al. [280] from 2021. An overview of its architecture is given in Figure 5.7. SNAP uses a novel parallel associative search to look for non-zero weights and input activation pairs in compressed, unstructured input data. Using a highly efficient search algorithm, they report an average compute utilization of 75 %. To benefit from compression for

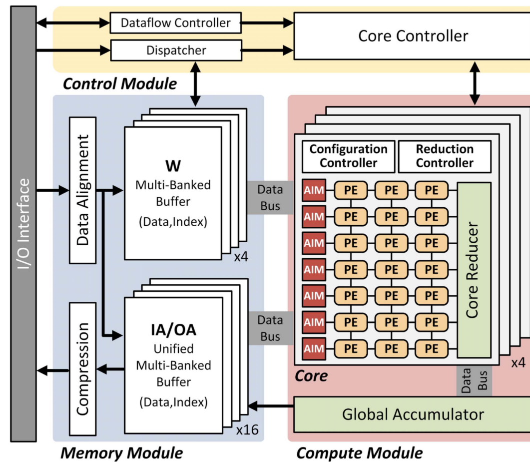


Figure 5.7: System overview of SNAP, a DNN accelerator that supports weight and activation pruning, according to [280].

increased energy efficiency, their PEs, which operate on a channel-first dataflow, can be configured in two modes to cut the write-backs of partial sums. In total, they can reduce the number of write-backs by 22× compared to state-of-the-art accelerators. To assess their design the authors built a prototype using 16 nm Complementary metal-oxide-semiconductor (CMOS) technology. It features, 252 PEs as 7×3 array. At 16-bit operands, their prototype reaches an efficiency of 21.55 TOPS/W with 10 % dense workloads. Using pruned ResNet-50 model they report an efficiency of 3.61 TOPS/W, which equals 90.98 FPS throughput, while dissipating only 348 mW. With synthetic workloads they show an effectual energy efficiency of 1.67, 5.06, and 21.55 TOPS/W on dense, medium dense (40 % dense) and sparse (10 % dense) weight and input matrices, respectively.

The above introduced and described accelerators use distinct method to leverage as much sparse computations as possible. Although they can benefit many sparse computations, compression techniques usually come with a hardware and scheduling overheads. Therefore, novel systematic approaches to increase and prune structured sparsity in dynamically changing activations have to be investigated.

5.4 Spex Exploration Framework: A Tool to Systematically Increase Regular Sparsity in CNNs

As we stated before, regular sparsity in the form of blocks is rare in CNNs. However, we have many elements in output feature maps that are close to zero (as, for example, in Figure 5.1). By introducing a threshold, we can increase the sparsity. Therefore, each output of a convolution operation, is compared to the threshold after activation and either set to zero or left unchanged. With a well-selected threshold, those close to zero values that only contribute a small fraction to the overall result, can be pruned as well, with just an insignificant impact on the network accuracy. In addition, choosing the right thresholds for each layer, yields more regular sparse blocks during inference and hence our sparsity approach can return a higher efficiency.

However, the amount of threshold combinations for a recent CNN can become massive. Assuming, for example, 100 distinct threshold values for each layer in ResNet-50 [40], we end up with $7.52 \cdot 10^{96}$ permutations, which makes exhaustive search unfeasible. Moreover, they are typically hard to pick, since the model faces various inputs that lead to changing feature maps [161]. Hence, some input samples are impacted more by a given threshold than others. Spex has the goal to explore this large design space that opens between the threshold combinations and the network accuracy systematically. After exploration, it aims to return a well-balanced set of thresholds that satisfies the design requirements and maximizes the amount of leveraged sparsity.

An overview of Spex can be found in Figure 5.8. As input, Spex takes a workload description file. The workload file contains all the required information for the exploration: First the actual DNN is defined, then the validation dataset on which the exploration operates is specified. In addition to that, the description file also contains information about the underlying hardware accelerator, e.g., the spatial dimensions or the storage size for sparse blocks. Based on those inputs, Spex can start to explore the design space. This processing will be described in depth in the following. After the exploration found viable solutions, Spex returns the found set of thresholds, along with the achieved savings in memory transfers and MAC operations, as well as the final network accuracy.

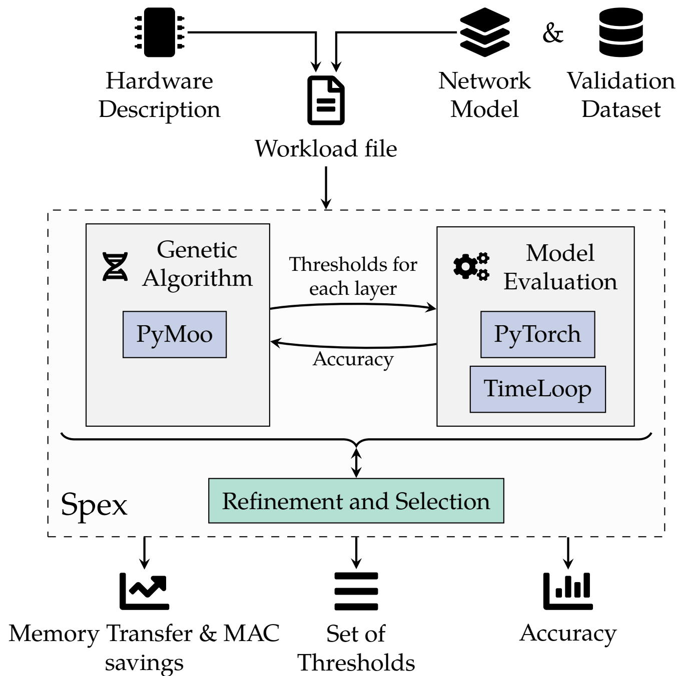


Figure 5.8: Components of Spex and its inputs and outputs. The tool operates on a given workload description and explores thresholds for each layer by applying a genetic algorithm. Found solutions and results are presented afterwards in a comprehensive visual and text form.

5.4.1 Design Space Exploration

The centerpiece of Spex is an exploration algorithm that generates a set of thresholds, which are then evaluated. This algorithm tries to minimize a set of objective functions $f_n(x)$ with are subject to another set of inequality constraints $g_i(x) \leq 0$ [320]. The goal of the optimization algorithm is to find input variable vectors $\vec{x}$ from the decision variable space $\Omega = \{x \in \mathbb{R}^n\}$ that minimize all objective functions. Elements of an input variable vector represent a list of thresholds for each layer in the network. This set is then evaluated, and the evaluator returns actual values for all objective functions, as well as the magnitude of constraint violation. The relationship of the objective functions and viable solutions is often represented using Pareto optimal solutions.

Thereby, each Pareto optimal solution represents an optimal solution to the given problem. In general, Pareto optimal solutions are characterized by the fact that no other $\vec{x} \in \Omega$ exists that yields better results on a combination of the objective functions. These solutions are also referred to as non-dominated solutions, as no other solution dominates them in the objective space. All found Pareto solutions can be visualized using a Pareto frontier. Therefore, all solutions in the decision variable space are plotted using a scatter plot and Pareto optimal solutions may be highlighted.

To guide the search for a beneficial set of input variables, we choose a Genetic Algorithm (GA) as optimization and exploration algorithm. This kind of iterative optimization algorithm picks for each evaluation a specific input variable vector, which is then iteratively optimized towards minimization of all objective functions. The GA starts with a randomly initialized start population. Each individual in this population has a genome that represents the input variables. Then all individuals are successively evaluated and annotated with a fitness function that consists of the objective functions and constraint violations. Based on the fitness score, a selection of the best individuals is taken into the next generation, along with some randomly selected ones to support the convergence into a global optimum. Then this selection of individuals represents the parents for the next generation. In this breed or crossover step, the genomes of two or more parents interleaved. Typically, for each value in the genome the value is taken from a randomly selected parent. Before the GA evaluates this next generation, some randomly selected individuals undergo mutation. During mutation, values in the genome are altered slightly on a random basis. This aims to support convergence. This entire process of evaluation, selection, and mutation is iteratively repeated for multiple generations, ideally resulting in a very well-performing end generation. The GA may be terminated after a certain number of generations or if there is no notable change in the genome or input variables.

Besides the exploration algorithm itself, the evaluator plays a significant role. To maximize the regular sparsity in the form of blocks and the energy-saving achieved by pruning them, we evaluate not only one metric but multiple.

$$
\begin{aligned}
f_1(x) &= -max(\text{sparse blocks}) \\
f_2(x) &= -max(\text{accuracy}) \\
f_3(x) &= E_{inference} \\
g_1(x) &= accuracy_{min} - accuracy
\end{aligned}
\tag{5.1}
$$

The design space objective functions and constraints are given in Equation 5.1. First, our evaluator determines the number of sparse blocks that were created during inference regarding the given thresholds. Therefore, it runs the entire inference using the validation dataset and monitors the number of sparse blocks that are present and the number of created blocks by applying the threshold to the given layer. This number of sparse blocks is fed into the first objective function $f_1(x)$, that tries to maximize their number. It has to be noted that the optimization algorithm constantly tries to minimize all objective functions, and hence it is negated. Secondly, the evaluator reports the algorithm accuracy. Straightforward metrics like Top-1 or Top-5 accuracy for classification tasks are supported, as well as pixel-accuracy or Intersection over Union (IoU) for object detection or semantic segmentation tasks. Again, we aim to maximize the accuracy using the objective function $f_2(x)$. Third, we analyze the actual energy savings that are achieved during inference through pruning entire blocks in the output feature maps. The total inference energy is subject to minimization and is represented by the third objective function $f_3(x)$. Most of the saved energy originates from fewer off-chip memory accesses, but also from fewer computations. This metric requires a hardware model, since the energy consumption only correlates with the number of created sparse blocks but is not exactly proportional. For example, if an output feature map is small, a large share of it can be stored in the local memory, resulting in fewer off-chip memory transactions in the first place. Finally, we add one inequality constraint $g_1(x)$ to the design space to drop solutions that do not meet a minimum accuracy $accuracy_{min}$ set by the designer.

Other claims and goals of our approach are already represented by the three objective functions above or do need any exploration in advance. For example, the claim that our hardware extension adds only little area to the overall accelerator design. Most components of the architecture remain unchanged regardless of the exploration results. But the local cache that stores the position of sparse blocks, may grow with the number of sparse blocks per layer. However, its size is directly proportional to the number of sparse blocks, represented by $f_1(x)$.

Based on the three objective functions and the one constraint, we set all the parameters required for a directed parameter search. During evaluation, all three objective functions are treated equally, however, it is also possible to weight them.

5.4.2 Implementation details of Spex

Spex is entirely implemented in Python to ensure compatibility with common machine learning and DNN frameworks. As stated before, Spex takes as input a workload description file that contains all required information. An example of such a file is given in Listing 5.1. Workload files are implemented using the data-serialization language YAML, which makes reading for both humans and machines easy. Our decoupled workload files, allow us to run multiple explorations with different parameters independently. Since the files can be generated automatically, we can, e.g., run parameter sweep to find the best GA parameters. First, the `problem_function` defines that we are dealing with an exploration problem for regular sparsity. Most interestingly are the `model` and `exploration` sections. The first describes the kind of DNN workload that is used as basis for exploration. Here also the according function to evaluate the accuracy is given. The latter defines all relevant parameters for the GA, like the population size or parameters for mutation and crossover. This section also defines the dataset on which the exploration is evaluated, the user-defined minimum accuracy that is evaluated in $g_1(x)$ and a set of predefined parameters that create individuals in the initial population with a given genome. The latter aims to accelerate the optimization process by introducing individuals with known solutions. The user can also provide a set of extra arguments, which will be passed to the problem when it is initialized. Here, we can specify, for example, the block size or how the thresholds picked from the variable space, to accelerate the exploration.

Listing 5.1: Exemplary workload description file.

```
workload:
  problem:
    gpu_id: 1
    problem_function: sparsity_problem

  model:
    type: yolov5s
    accuracy_function: detection_accuracy

  exploration:
    nsga: # Setup the parameters for your nsga algorithm
      pop_size: 20
      offsprings: 20
      generations: 20
      mutation_eta: 20
      mutation_prob: 0.9
      crossover_eta: 30
      crossover_prob: 0.7
```

```yaml
19      ftol: 0.04
20    minimum_accuracy: 0.33  # accuracy constraint
21    predefined_parameters: [0.005, 0.01]  # predefined threshold or
          quantization bit
22    datasets:
23      exploration:
24        type: cifar10
25        sample_limit: null # number of validation to test
26        batch_size: 128
27    extra_args:
28      discrete_threshold_steps: 200
29      discrete_threshold_method: linear
30      threshold_limit: 0.15
31      block_size: [8,8]
```

At the start, the exploration script of Spex reads this workload file and sets up the optimization algorithm. As exploration algorithm, we choose in particular the well-established Non-dominated Sorting Genetic Algorithm (NSGA)-II [308], which is a GA that shows great performance on continuous multi-objective optimization tasks. The algorithm itself is taken from the Pymoo Python library [305]. Pymoo allows selecting from a wide range of different optimization algorithms, which are available in this library. In contrast to a traditional GA, NSGA features a modified survival and crossover function that enables faster convergence. NSGA-II extends a traditional GA with two aspects. First, it is elite-preserving, i.e., a set of non-dominated solutions will always be kept for the next generation. Second, through sorting found solutions for non-dominance the computational complexity is reduced from $\mathcal{O}(MN^3)$ to $\mathcal{O}(MN^2)$ with M being the number of objectives and N the population size.

After the exploration algorithm is initialized, we can continue to set up the neural network model. Within Spex, we support PyTorch models, since the framework is extemely popular and supports a wide range of DNN operations and models. To evaluate the trade-off between increased regular sparsity and model accuracy, Spex has first to alter the PyTorch model. Therefore, we replace all DNN layers in the model with counterparts that support our regular threshold-based pruning flow. Each layer gets two new parameters that can be changed by the exploration algorithm, namely the threshold value and the block dimensions. From a functionality standpoint, in particular the `forward` function, which is called during inference in PyTorch, is changed. This allows us to manage any DNN workload without the need for expensive retraining. In contrast to the hardware implementation, we enrich the regular sparsity before the layer is computed, which is implemented using PyTorch, since CONV layers and ReLU layers are not fused.

Altered DNN layers now perform the previously described process. Foremost, the input feature map has to be present in a 2D matrix form. Hence, for CONV layers, an im2col transformation is applied, while dense or Long-Short Term Memory (LSTM) layers do need any transformation. Then the altered forward function divides this matrix into blocks of a predefined size. In this process, batches are handled independent of each other, since thresholds should be determined for each input separately. If necessary, the 2D matrix is padded with zeros to ensure even tiling into blocks. In the next step, the sum of each block is computed. Before each sum is individually compared to the threshold parameter, the algorithm counts the number of already present sparse blocks for later evaluation. Afterward, all blocks with a sum less than the threshold are set to zero. With this altered forward function, we can directly observe the impact of created sparse blocks during the inference process. For example, how newly created sparse blocks, influence the performance of the next layer. By changing merely the forward function, other layer types may be added later without the need for substantial changes.

Now, that all components of Spex are initialized, we can start the actual parameter exploration. The GA starts with an initial population. Individuals in this population are evaluated one after another. During the evaluation, each layer is annotated with a threshold value taken from the genome. Then, an inference of the provided dataset with the altered DNN starts. Therefore, the exploration task calls for each individual an accuracy function that is specified in the workload description. In addition, it computes the energy consumed by the memory accesses for the inference and the number of created sparse blocks.

These three variables provide all the required information for the GA to iteratively find an optimal set of thresholds. Over multiple populations, the GA now tries to minimize the three objective functions. The algorithm terminates either when enough feasible solutions are found or when the input variable vector only changes slightly. Finally, Spex stores all found solutions, the history with each generation's population and the corresponding objective function results in a results file, which is a serialization of all data using Python's pickle package.

As stated before, the entire design space for a modern CNN workload with typically more than 50 layers is vast. Even using a GA, finding a viable set of thresholds can consume a huge amount of time. Therefore, Spex supports various features to accelerate the exploration itself and to systematically shrink down the design space.

Since the inference of the entire dataset for each single individual in each population can already take a huge amount of time, Spex can work with partial datasets. In this case, only a subset instead of the entire validation dataset is used to determine the accuracy. Thereby, Spex ensures that all classes of the dataset are covered and a considerable number of samples per class is used. In addition, it shuffles the dataset for each iteration to factor in a large variety of the dataset. On the bare computational side, Spex supports inference on multiple GPUs. Moreover, it supports parallel inference through spawning workers on several High-Performance Computing (HPC) nodes. This is implemented with the widely used parallelization library Dask[11].

Besides parallelization of the algorithm and individual explorations, we can also apply thresholds among individuals using discrete steps. In general, this discretization shrinks the design space and speeds up exploration. Steps can be applied in linear or logarithmic form. The latter puts more emphasis or more discrete steps on small thresholds. Looking at small thresholds more in detail is especially crucial, since large thresholds applied to sensitive layers may immediately lead to large errors and consequently low accuracy, as the error propagates through the entire DNN. Hence, it is beneficial to explore those small threshold values in a more fine-grained way. Besides discrete steps, we can also set an upper bound for the threshold values. This can be determined empirically. For example, the limit may be set to a value where the algorithm error becomes so large that the model accuracy is zero. This removes all threshold combinations from the design space that will result in zero accuracy anyway.

As another method to accelerate algorithm convergence, we can add individuals to the starting population of the GA that have known solutions. For instance, we can add an individual with thresholds set close to zero that result in a small degradation. This individual then supports the convergence, as sensitive layers are already considered by small threshold values. When mating with this individual happens, this characteristic of sensitive layers might be carried on into the offspring. For less sensitive layers, mutation and further mating will allow for finding the maximum possible threshold in a bottom-up fashion. However, this has to be done carefully so that the algorithm does not stick to these solutions and ultimately does not find a good set of thresholds. Therefore, other parameters like the mutation and crossover rate have to be picked well to allow the GA to break out of this initial solution.

[11] https://www.dask.org/

Exploration within Spex happens in 16-bit floating-point precision, which allows us to use the inference provided by PyTorch without changing the underlying implementation. Most DNNs, however, utilize quantization to reduce the computational complexity and memory footprint. Quantization has an impact on the number of sparse blocks that can be found during inference. But quantization also increases the number of already present sparse blocks. For example, activations that are already close to zero, will become zero after quantization. Consequently, blocks with small elements will become sparse. For all other elements in the output feature map, the found thresholds have to be quantized as well. Here, the same quantization scheme has to be used to ensure the expected behavior. This can be done after Spex has found thresholds in floating-point precision.

After the exploration has terminated, Spex offers some tool for further evaluation and visualization. Therefore, Spex features decoupled scripts, which read a given result file. To visualize the results, Spex can generate various plots. Most interestingly are scatter plots that show the correlation of the objective functions. This kind of plot reveals, for example, how an increased number of sparse blocks impacts the network accuracy or the energy consumption. Besides that, plots that show a breakdown of the relative number of created sparse blocks per layer for all feasible solutions give details about which layer is most tolerant to our pruning method. Similarly, we can also investigate the magnitude of thresholds for each layer. In addition to visualization tools for single explorations, Spex also offers scripts to merge multiple Design Space Explorations (DSEs). For example, looking at runs with different block sizes can support the hardware design process, by choosing an accelerator size that combines a good pruning ratio with a high performance. Moreover, combining multiple runs with different GA configurations helps to tune important parameters of the algorithm like the mutation or crossover rate and probability.

5.4.3 Energy estimation using Timeloop and Accelergy

Two of the three objective functions $f_1(x)$ and $f_2(x)$ are covered by altering the PyTorch forward function, as described in the previous section. However, the objective function to reduce consumed energy during inference ($f_3(x)$), is not considered yet. While PyTorch can report model accuracy and the number of created sparse blocks, it cannot estimate the inference energy, as this strongly depends on the underlying hardware. Hence, it is determined by Timeloop [163] coupled with Accelergy [169].

As stated before (see Subsection 3.2.1), Timeloop is a widely used tool to find an optimal mapping of a DNN workload onto a given hardware accelerator. Since the actual mapping of the workload has a strong impact on the number of memory accesses, the mapping has to be fixed beforehand. The corresponding energy consumption caused, among other things, by memory transactions is computed by Accelergy. Coupled with the CACTI plugin, Accelergy can give a close estimate of the energy consumed in memories. Accelergy and CACTI thereby use the 45 nm technology library, which is provided as open-source.

The energy saved through our sparsification method in particular depends on the size of the local memory. If more data can be kept local between layers, less offloading happens and compressing sparse blocks has a smaller effect. In contrast, layers that cause a lot of offloading yield larger energy savings from our approach, even if the relative number of sparse blocks is the same.

To estimate the actual energy savings, Spex runs performs an entire mapping and tiling process using Timeloop in advance. Since, Timeloop only supports mapping of a single layer, Spex triggers a mapping for each layer individually. Timeloop then reports for each layer valuable information for our energy estimation. First, the number of off-chip and local memory accesses, but compute cycles required in the PE array to process the DNN workload. Accelergy transforms these numbers into the energy needed to perform the given action. All relevant energy information is collected in a Comma Separated Values (CSV) file. When Spex evaluates the objective functions of a threshold combination, this CSV file is read again to compute the energy this specific pass consumed. Therefore, it deducts the number of created and present sparse blocks form the number of off-chip memory transaction, if these values are offloaded otherwise.

Sparse blocks that otherwise would have been stored in local memories may free up resources in the local memory. However, this case is not covered by Spex and Sparse-Blox, as this requires expensive changes to the scheduler. For example, the accelerator then has to decide on the fly whether other data can be placed locally. This is contrary to our design goal, which is to have a low-overhead hardware extension rather than an overly complex one.

Besides the energy, Accelergy can also give an area estimation. Spex uses this estimation to compute the area overhead the later hardware extension Sparse-Blox adds to the design. Therefore, Spex adds all components of Sparse-Blox to the architecture description. This includes the cache that stores found sparse blocks from on layer

to the next, an adder tree and the threshold comparison. The cache is the largest contributor the area overhead. Depending on the maximum number of sparse blocks that are found in a layer, we can define the size of this cache. The area estimation itself does not influence the exploration results, but it is useful to compare our approach to other pruning methods in terms of the area overhead.

5.5 Sparse-Blox: A Hardware Extension to Prune Regular Activation Sparsity

The advantage of our pruning approach becomes apparent when it is integrated in a CNN hardware accelerator, e.g., for low-power embedded vision applications. Our hardware extension Sparse-Blox can be added to common hardware accelerators like systolic arrays or vector processors. As stated in Section 2.5, DNN inference is dominated by a series of matrix-matrix multiplications. Hardware accelerators can leverage their spatially independent computations by connecting hundreds of PEs in mesh. This allows to reuse operands like weights or inputs in neighboring PEs, reducing the number of expensive load and store operations. These operations cause DRAM movements, which not only increase latency, but also have an impact on the energy consumption.

Typical systolic array sizes are power of two such as 8×8, 16×16 and so on depending on the available area and energy budget. Vector processors typically also feature, e.g., 8 or 16 parallel lanes. But even if one works with a large array, e.g., of 64×64, the matrix multiplication of a convolutional layer must be broken down into smaller tiles. As an example, the second convolution layer in the first residual block of ResNet-50 [40], has the following dimensions for input and weight matrices: 576×3136 and 576×512 This requires 3528 or 1 806 336 passes on 64×64 or 8×8 arrays, respectively. Our activation pruning method aims at these numerous passes on a given hardware accelerator. With the thresholds found by Spex and the hardware introduced by Sparse-Blox, we can reduce the total number of passes by pruning them dynamically during inference. In addition to fewer required computations, Sparse-Blox can also compress those sparse blocks. This helps to use the limited and expensive local Static Random Access Memory (SRAM) memory more efficiently and reduces off-chip memory transactions, which in turn saves energy.

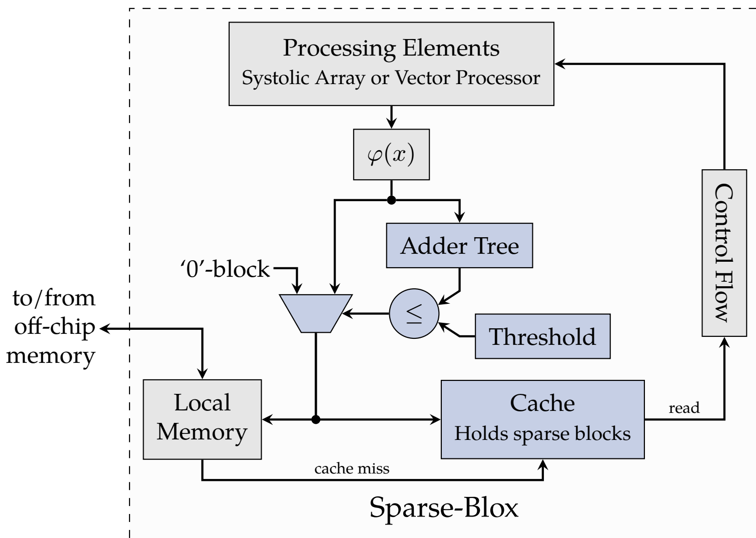


Figure 5.9: Proposed hardware accelerator extension Sparse-Blox that enables our regular activation pruning scheme to save off-chip memory transactions. Only a few elements have to be added to an existing accelerator: a cache to store sparse blocks, freeing up valuable storage in the local memory, an adder tree to determine the block sum and a logic to compare the sum with the threshold previously defined by Spex.

As we stated before, Sparse-Blox, the hardware extension to enable inexpensive pruning of regular activations in CNNs, can work with most common hardware accelerators. An overview of the hardware is given in Figure 5.9. Our hardware accelerator leaves the typical components of hardware accelerators untouched, and only adds a few components, which are highlighted in the figure. All components of are Sparse-Blox are inserted into the data path from the PEs to the local memory. This allows Sparse-Blox to monitor the magnitude of intermediate results and consequently prune entire blocks.

Its function is outlined in Pseudocode 5.2 and described in the following. At the beginning, when the local cache that holds pruned blocks is empty, we compute the matrix multiplication as usual. Thereby, a sum for each block is accumulated by an adder tree, which is the first component that is added to the hardware accelerator. This sum is compared to the threshold that has been defined by Spex beforehand.

Pseudocode 5.2: Pseudocode representation of Sparse-Blox's dataflow.

```
Data: Input feature map: I_map (im2col transformed)
Result: Output feature map: O_map
1  I_blocks ← split_in_blocks(I_map)
2  foreach block ∈ I_blocks do
3      if available_in_cache(block) then       /* check whether block was pruned before */
4          O_map ← 0                                            /* Place 0-block in output */
5          continue
6      out = gemm(block, W)                                         /* Do matrix multiply */
7      sum = compute_sum(out)                           /* Compute sum using adder tree */
8      if sum ≤ threshold then
9          store_to_cache('0-block')                          /* Prune and add cache entry */
10     else
11         O_map ← out                                         /* No pruning, store result */
12 end
```

Thresholds are stored in a simple register and are configured by the General-Purpose Processor (GPP) that controls the accelerator via AXI-Lite. It is also conceivable to add a buffer instead of a simple threshold register, to also prefetch thresholds for upcoming layers. If the comparison of the sum is less than the threshold, pruning happens. In this case, a multiplexer is set to output a '0'-entry. Additional control logic, which is not shown in the figure, discards the partial results. Instead, an entry in a dedicated cache, which stores all positions of sparse blocks, is created. Since, this cache does not need to recall all elements in a block, but only its position, an entry to this cache requires much less memory storage, than an entire block. This is how Sparse-Blox can compress data on the fly during inference. For example, an 8×8 block requires at least 64 words in the local memory, while one compressed '0'-block only needs two words for the corresponding x and y coordinates in the two-dimensional output matrix. A sparse block that is not stored in the local memory additionally provides space for further data that would otherwise have been offloaded. As a result, Sparse-Blox can also reduce the number of energy-expensive off-chip memory transactions. Of course, we also need to adopt the control logic when the PE array request data that is not present in the local memory but in the cache. Therefore, the cache is logically located before the local memory. Consequently, a read request is first addressed by the sparse block cache. If the requested data is found in the cache, the entire computation can be skipped as all elements of this partial operand are zero. By skipping entire computational blocks, Sparse-Blox can also reduce the number of computations through straightforward regular pruning and speed up the inference.

If requested data is not available in the cache, the computation runs like without modification by Sparse-Blox: Data is loaded from the local memory and computed in the usual way.

5.6 Evaluation and Discussion

We evaluated the performance and energy saving that we could achieve with Spex and Sparse-Blox extensively. As workloads, we chose Computer Vision (CV) tasks, in particular, CNNs, since they are computationally expensive, have large memory footprints and are crucial to many modern tasks like autonomous driving or automated manufacturing. First, we will show how we use Spex to find optimal thresholds for each layer that maximizes pruning and maintains a high accuracy. As stated before, the design space can grow incredibly large, therefore, we show how we tuned Spex to yield satisfactory results in a short amount of time. After that, we will discuss the found threshold combination for various workloads in terms of their distribution and magnitude. Once we found feasible thresholds, we will move on to the hardware part. Here, we will look at the area aspects of Sparse-Blox in comparison to other accelerators.

5.6.1 Selected Workloads

Most applications that require sophisticated environment perception need some form of CV to process camera inputs. Currently, one of the most accurate methods are CNNs. However, these neural networks are not only computationally demanding, but their memory requirements are also huge. Due to the rich information that is encoded in images, multiple filters have to run in parallel over the same image all producing intermediate feature maps, requiring memory. Typically, these large intermediate feature maps have to be, at least partially, offloaded into an off-chip memory, causing many energy expensive transactions. Since our pruning approach aims at reducing those transactions, CNNs are a perfect candidate workload for Spex and Sparse-Blox.

As specific workloads, we chose networks and corresponding datasets that are commonly used as benchmarks: ResNet-50 and YOLOv5s. We picked these CNNs intentionally for comparability with other works, although the overall sparsity can be much larger depending on the application. For instance, far-range LIDAR evaluation shows a lot of sparsity in feature maps [49]. Details on both workloads can be found

Parameter	ResNet-50 [40] with ImageNet-1K [1]	YOLOv5s [84] with COCO [69]
# Explored Layers	52	213
# Parameters	25.56 million	7.23 million
Input size	$3 \times 224 \times 224$	$3 \times 640 \times 480$
Filter sizes	$3 \times 3, 1 \times 1, 7 \times 7$	$3 \times 3, 1 \times 1, 6 \times 6$
Largest feature map	$111 \times 111 \times 64$	$320 \times 240 \times 32$
Baseline accuracy	80.86 % (Top-1)	37.4 $mAP^{val}(0.5 : 0.95)$

Table 5.1: Workload details used for evaluation with Spex and Sparse-Blox.

in Table 5.1. The first network, is extremely popular in the area of image classification and is trained on the ImageNet-1K dataset. The ResNet-50 model is a pretrained version, which is provided with PyTorch, allowing for great comparability. It achieves a baseline Top-1 accuracy of 80.858 %. YOLOv5s makes a great addition, as it is a CNN for objection detection rather than image classification. The small configuration of YOLOv5 is selected for its great accuracy at a reasonable size. It is trained using the COCO dataset. Since, COCO has fewer classes than ImageNet, it might be harder to exploit large amounts of sparsity in YOLOv5s. Similar to ResNet-50, a pretrained version of YOLOv5s is used from Ultralytics[12]. Our exploration will analyze the mean Average Precision (AP) (mAP) in the IoU confidence interval 0.5 to 0.95 as accuracy metric. The network's baseline accuracy is 37.4 mAP. In all experiments, we set an appropriate lower accuracy bound, which maintains a high model accuracy and maximizes pruning.

5.6.2 Evaluation Setup

We performed extensive experiments using Spex to understand the vast design space. Therefore, we investigated the impact of different block sizes, threshold combinations, GA parameters and CNN workloads. To cover a range of hardware architectures we looked at block sizes of 4×4, 8×8 and 16×16, to represent embedded and edge systolic array accelerators, as well as at 1×16 with is common for vector processors.

[12] https://github.com/ultralytics/yolov5/ Tag: v7.0, Accessed 2023-10-17

In total, we analyzed over 400 individual experiments. Our results are generated with the support of a cluster system that features ten NVIDIA Ampere A100 GPU nodes. The total runtime for all experiments summed up to about 160 hours.

Algorithm Constraints

The main objective of our exploration using Spex was to find a set of threshold values that meet two criteria: to keep the accuracy as high as possible, but at least over a given lower bound, and to maximize the number of sparse blocks created overall. Although these factors are contradictory, the goal is to find sweet spots in the complex design space. In order for the exploration to yield viable solutions in a reasonable amount of time, we have to constrain and shrink the design space.

As stated before, DNNs may be extremely sensitive to large threshold values, since errors propagate through the network. At first, we added a constraint to limit the threshold maximum. To identify this, we gradually increased the threshold value for all layers in both workloads and observed the point at which the network delivers zero accuracy on the validation dataset. Based on this, we left a 50 % margin on top of the threshold to allow for higher thresholds in less sensitive layers. As a result of this process, we set the upper threshold limit for ResNet-50 to 0.8 and for YOLOv5s to 0.1. For block sizes of 4×4 we increased the threshold limit for both workloads by 50 %, since smaller blocks can handle higher threshold values. In terms, of discretization of the threshold steps, we decided to use logarithmic steps. Since they, as described previously, help to explore small threshold values more in-depth. In our experiments, we used 400 discrete thresholds steps.

To reduce the exploration time, we utilized the possibility of Spex to introduce individuals to the stating population of the GA. We picked individuals with the following threshold value with the corresponding accuracy for all layers: For ResNet-50 we chose individuals with thresholds of 0.15 and 0.25, resulting a Top-1 accuracy of 73.8 % and 55.6 % when using 8×8 as block size and in 70.0 % and 31.5 % using 16×16 blocks. For YOLOv5s we picked 0.005 and 0.01 yielding scores of 32.53 and 23.65 mAP using 8×8 blocks and scores of 29.05 and 16.31 using 16×16 blocks, respectively. The first individual for each workload yields a comparability good accuracy and the last a not acceptable accuracy degradation. As a result, those individuals form an upper and lower bound for potential threshold values. However, these are not hard bounds, as the algorithm can still break out using mutation and crossover. Only the aforementioned upper threshold bound represents a hard limit.

Crossover probability \ Mutation factor	10	15	20	30
0.7	3	3	3	3
0.9	4	4	4	3

(a) Generations until all solutions are feasible.

Crossover probability \ Mutation factor	10	15	20	30
0.7	342	343	343	345
0.9	336	336	336	343

(b) Number of found feasible solutions (total 375).

Figure 5.10: Result space for mutation and crossover parameters of the GA with 15 individuals and 25 generations. Evaluated on a ResNet-50 with 16×16 blocks.

Finally, to speed up the evaluation of individuals, a subset of the validation dataset is used in Spex. Hence, we set the sample limit to 4096 validation data samples for ResNet-50 and 2048 for YOLOv5s. Those sample limits cover all classes that are present in the corresponding datasets. It has to be noted that a randomly selected subset might accidentally pick samples, which are more or less robust to created sparsity, leaving a small uncertainty in the accuracy. However, a subset of the validation dataset still has a statistical significance, and we can evaluate selected non-dominated points with the entire dataset.

NSGA-II Tuning

Once the design space is sufficiently constrained, we can tune the GA to achieve feasible solutions quickly. GA parameters typically encompass factors, such as mutation and crossover rate and probability. But also straightforward parameters like the population size and the number of offsprings per crossover have an impact on the convergence speed. In addition, the termination criterion plays an important role. The algorithm should be able to terminate, when the found set of thresholds only changes slightly among generations. To determine adequate values for those parameters, we first perform an exhaustive search. Therefore, we run a sweep with different GA configurations and observe how long it takes for the GA to converge.

The results of this sweep are shown in Figure 5.10. For four different mutation factors and two crossover probabilities, we show the number of generations it takes for all individuals to be considered as valid and the total number of feasible solutions after 20 generations with 15 individuals. Individuals are considered valid when the accuracy is one percent point below the baseline. All those experiments are performed with the

measures described above to speed up the exploration. In general, a larger mutation factor results in less deviation from the individual's original genome. At first glance, we saw that a larger mutation factor and a smaller crossover probability, yielded better results. These results can be explained by the artificially introduced feasible individuals to the starting population, which then stick with the population until the end. Hence, with a lower mutation factor and smaller crossover probability, the algorithm focuses more on these individuals and in the worst-case sticks with them until the last generation. However, these individuals make the convergence much faster and allow the GA to produce valid solutions in a short amount of time.

As a result, we set the following parameters, which balance convergence speed and diversity of solutions: the mutation factor is set to 20 and the crossover probability to 0.9. This configuration is applied for all following experiments. In addition, in all experiments our GA evaluated a population of 20 individuals over 25 generations, in total 500 individuals. The number of offsprings was fixed to 20 so that the population is not growing. This equalizes runtime and allows numerous distinct solutions.

Hardware Model

To see how our pruning method increases the energy efficiency, we have to look at the energy consumption of an architecture that uses Sparse-Blox. Moreover, to prove our claim that Sparse-Blox adds fewer area to the DNN accelerator compared to other sparsification methods, we have to investigate its area utilization. To get both values, we added a hardware model in Timeloop [163] of a systolic array, which has the same number of PEs as it is configured in Spex. The systolic array itself features an on-chip buffer memory with 256 kB, a register for each PE and 128 24-bit accumulation buffers. Timeloop is coupled with Accelergy [169] to get area and energy estimates. For our experiments, we picked Timeloop together with Accelergy instead of our approaches, which were introduced in the previous chapter (Chapter 4), to allow for comparison and benchmarking with other works. Besides the systolic array itself, we added all necessary components of Sparse-Blox: A configurable cache, whose size is determined by the used benchmarks and systolic array size as well as by the maximum number of found sparse blocks per layer by Spex. Furthermore, we added an adder tree for sum computation and the decision logic.

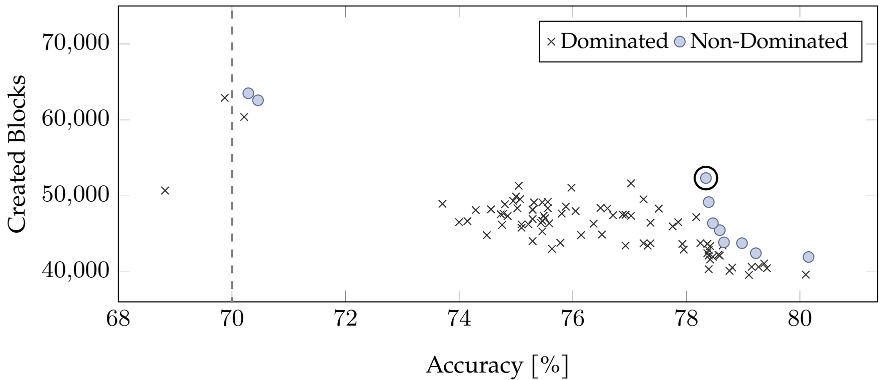
With all the components in place, Timeloop can compute an efficient mapping for the given DNN layer. This also yields the number of required off-chip memory transactions and the utilization of the local memories, which are both particularly

interesting when we want to investigate how much energy our pruning method can save during inference. Therefore, we looked into the number of sparse blocks Spex can create for a given layer. To assess the energy consumption with our method in place correctly, we had to keep one consideration in mind. We cannot simply reduce the number of off-chip transfers by the number of sparse blocks, but also have to look at the local memory. Sparse blocks do not take any space in the local scratchpad memory and consequently blocks that would otherwise have been offloaded can stay in the local memory. To account for that, we effectively ran Timeloop twice, once with the entire workload and then with a workload whose dimensions are shrunk by the number of found sparse blocks. When running Accelergy in the second run, we added extra accesses to the cache for each access to the local scratchpad, and for each write into the scratchpad we counted an action in the adder tree and decision logic. By doing that we could investigate the energy using estimates, without the need to synthesize the architecture and run an entire Register Transfer Level (RTL) simulation, which makes the design process much faster.

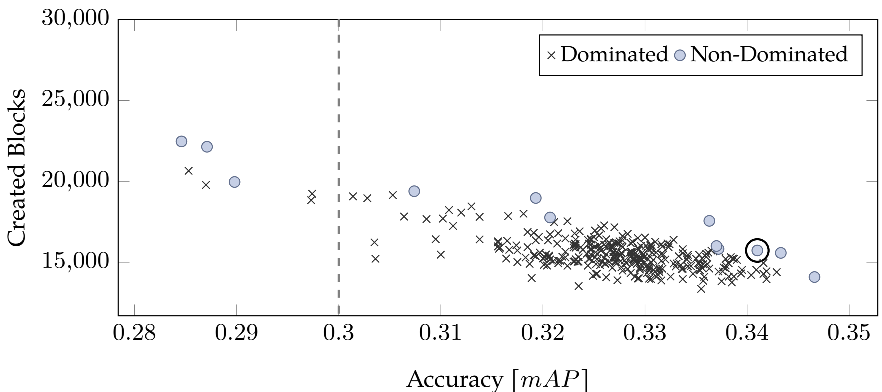
5.6.3 Number of Found Sparse Blocks and Savings in during Inference

The most relevant finding of Spex is the number of sparse blocks created with a given set of thresholds and how this impacts the model accuracy. More created sparse blocks allow skipping more computations and are saving more memory transfers during inference. However, we did not want to sacrifice too much prediction accuracy, therefore, Pareto-optimal points have to be evaluated regarding the requirements of the application. For ResNet-50 and YOLOv5s all feasible solutions are shown in Figure 5.11a and Figure 5.11b, respectively. The figure shows the average number of created blocks during inference of the entire validation dataset, using a block size of 8×8. The Pareto frontier is shown through the non-dominated solutions in light blue.

Both plots reveal findings that support our initial hypothesis: The correlation between created sparse blocks and the drop in accuracy is not linear, but there are sweet spots. Looking at the commonly used image classification CNN ResNet-50, which has in total 345 400 8×8 blocks per inference, Spex could create on average 52 349.4 new sparse 8×8 blocks per inference. This equals to 15.2 %. Thereby, the accuracy of the algorithm dropped by merely 2.5 percent points. Before our pruning approach was



(a) Solutions for ResNet-50 with block size 8×8 tested on 4096 samples. An accuracy constraint of 70 % yields an average of 52 349.37 blocks per inference created with an accuracy of 78.34 %, which is marked in the plot. The accuracy constraint of 70 % is shown with a dashed line.



(b) Solutions for Yolo-v5s with block size 8×8 tested on 2048 samples. An accuracy constraint of 0.3 mAP yields an average number of 15 728.78 blocks created per inference with an mAP of 0.341 The accuracy constraint of 0.3 mAP is shown with a dashed line.

Figure 5.11: Pareto optimal solutions, showing the accuracy over the amount of sparse blocks created per layer.

applied, ResNet-50 already has on average 4.1 % sparse blocks. Considering all sparse blocks, an inference of ResNet-50 yielded on average 19.3 % regular sparsity, with 8×8 blocks. As a result, we could save 814 million MAC operations on average in a single inference. Looking at the memory model, we reduced transfers to the off-chip memory from 53.6 million down to 43.2 million. This equals an absolute of 41.37 MB read and stores considering a 32-bit floating point format. For YOLOv5s, with a total of 945 560 8×8 blocks, Spex could create 15 729 or 1.7 % sparse blocks with an accuracy drop of 1.2 mAP. Combining this with the already present sparsity, our pruning approach saved in total 112 million or 1.81 % of all MAC operations. The corresponding off-chip memory transactions on an 8×8 systolic array could be lowered from 71.96 million to 70.74 million, resulting in 4.89 MB fewer data movement.

Besides the memory transactions and MAC operation savings, we also have to look at the energy savings. Therefore, we looked at the energy estimation Accelergy provides when applied to Timeloop's mapping. We then added Sparse-Blox to the architecture description and accounted for the reduced number of MAC operations and memory transfers. For ResNet-50 Accelergy estimated on an 8×8 array, a total energy of 9.96 mJ. Sparse-Blox could lower this to 8.06 mJ, saving 19.1 %. Considering YOLOv5s mapped on the same 8×8 array, with a total inference energy of 15.84 mJ, our approach reduced the energy by 1.76 % down to 15.6 mJ.

For ResNet-50 the relative amount of sparsity is close to the number of saved memory transfers. However, in YOLOv5s shows a gap. This can be explained with the size of the intermediate feature maps. Since YOLOv5s has smaller feature maps, more intermediate results can be kept local to the accelerator, and during a ResNet-50 inference more data has to be offloaded. Here we could benefit more from compression of this offloading. However, this finding indicates that we can also choose a hardware architecture with smaller local memories. Then we can benefit from a smaller accelerator area, and Sparse-Blox prunes and compress more off-chip memory transfers.

Besides 8×8 blocks, we analyzed other block sizes for common accelerators. Figure 5.12 shows the number of sparse blocks that are present before and after sparsification with Spex for our two workloads and four different block sizes. We observed that the smaller the block size is, the more sparsity Spex could create. This is because in this case the feature map can be divided into smaller and considerably more portions. Each of them in turn stores less information and pruning them leads to less impact on the resulting accuracy. Hence, smaller blocks are more robust to sparsification and pruning.

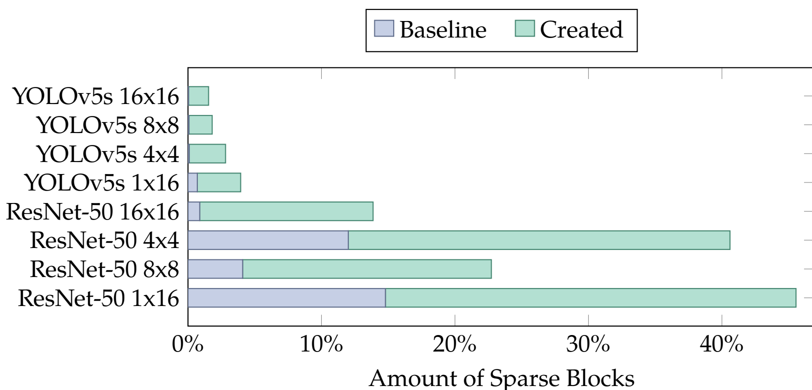


Figure 5.12: Average sparse blocks per layer that were present before and after thresholds were applied by Sparse-Blox. Independent of the workload, a higher amount of sparse blocks is observed the smaller the block sizes become.

For example, when comparing 4×4 and 16×16, each block covers 16 and 256 elements, respectively, 16× more. However, in terms of sparsity before sparsification on a ResNet-50 workload, 4×4 blocks yielded 12.01 % and 16×16 0.88 % sparse blocks of the total share of blocks. Even after the threshold search by Spex with a 1 % accuracy constraint, 16×16 could only prune 13.86 % of all feature maps, while 4×4 achieved up to 40.60 %. Looking at block sizes suitable for vector processors like 1×16, we saw that this block size even outpaces 4×4 blocks, despite covering the same number of elements in the feature map. This observation can be again explained by the block size. Narrower blocks in the flattened or im2col transformed feature maps affect less information dense portions of the whole feature map.

Finally, to get an impression in which layer our sparsification method is the most beneficial and what layers were more robust to pruning, we also looked at the distribution of sparse blocks for each layer. Like Figure 5.12, Figure 5.13 shows the number of sparse blocks present without sparsification with Spex and the number of blocks Spex was able to create. Here, we looked at all Pareto-optimal solutions of ResNet-50 with a block size of 8×8 that adhere a 1 % accuracy constraint. Very clearly, we can see big differences in between layers. Some showed large sparsity, while others did not. The same goes for the created sparsity through Spex. When looking at the layer characteristics, we saw that the layers with large sparsity predominantly have 3×3 convolution kernels, and the ones with less sparsity have kernel sizes of 1×1.

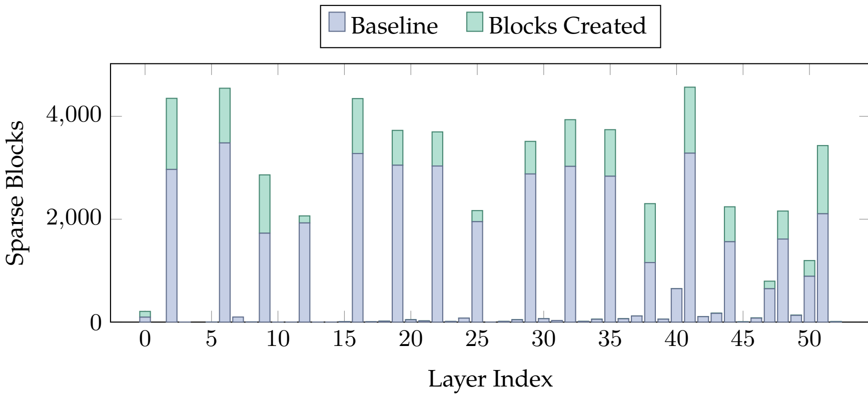


Figure 5.13: Breakdown of naturally occurring and created sparse 8×8 blocks itemized by layer for ResNet-50.

This can be explained by the underlying im2col transformation. With smaller kernel sizes, one dimension of the transformed feature map matrix is significantly smaller, as kernel size contributes squared. Dividing smaller matrices into blocks results in fewer blocks overall. In addition, each of these blocks contains more information. Thus, 8×8 sparse blocks impact many results at once, resulting in either lower thresholds and fewer sparse blocks or worse accuracy.

In summary, the designer still has to carefully determine the size of the blocks depending on the hardware architecture and the network topology. Spex, with all the insights it can generate, helps to make this design decision. Since the size of sparse blocks is equal to the size of the accelerator array, smaller blocks and arrays might lead to a reduced performance. At the same time, more sparsity that can be achieved by smaller blocks require more cache memory and thus more area.

5.6.4 Comparison to state-of-the-art Sparse DNN Accelerators

The main claim of our approach is a straightforward integration into common hardware accelerators, as well as a relatively small area overhead. To prove this, we performed an area overhead comparison of Sparse-Blox with other DNN accelerators working on irregular sparsity. We selected the well-established work SCNN [151] from 2017, as well as the more recent accelerators STICKER [279] and SNAP [280].

	SCNN [151]	STICKER [279]	SNAP [280]
Sparsity module			
Reference	24.8%	4.76%	12.5%
Sparse-Blox	4.95%	1.73%	4.81%
Overhead reduction	5.01×	2.75×	2.59×

Table 5.2: Area overhead comparison of Sparse-Blox against other state-of-the-art sparse DNN accelerators.

Those related works offer a wide range of different combinations. However, as none of them are available as open-source, we modelled each accelerator in Accelergy [169] to get an impression of the area consumption. As area model, Accelergy uses the same 45 nm component library as for the energy estimation above. Memory energy and area are generated with CACTI.

To get the area overhead Sparse-Blox adds to state-of-the-art accelerators, we first modelled the base accelerator in Accelergy according to the paper. Then we added the components of Sparse-Blox, a cache, an adder tree, and the corresponding decision logic. We matched the size of sparse blocks to the organization of PEs in the reference accelerator. The size of the cache that holds sparse blocks was set according to the results of the exploration with Spex, since then we can estimate the space requirements. The results of this qualitative comparison can be found in Table 5.2. We present the overhead in terms of percent it added to an accelerator without any sparsification method in place. The reference overhead was taken directly from the paper.

Since all three accelerators work on different PE array dimensions and feature different sizes of on-chip memories, which largely contribute to the area, we give the relative numbers in the table for better comparison. SNAP, the most recent work in our study, requires 12.5 % for their sparsification method. To determine the size of the caches, we then explored the impact to regular pruning on the accuracy. Considering the multiple 3×7 PE arrays of SNAP, we could find a solution that pruned 24.1 % of all blocks in ResNet-50 with 1 % accuracy drop. In total, this equals to about 312 000 blocks for an entire inference or about 6000 per layer. Consequently, we set the cache size to hold 6144 entries. With this cache in place, Sparse-Blox can enable pruning in this PE configuration while using about 2.5× less area. As a result, the area overhead is only 4.81 %.

Looking at STICKER, its sparsification module accounts for 4.76 % area overhead. The accelerator features a 16×16 PE configuration. As we saw before, the found number of blocks that can be pruned for ResNet-50 is lower compared to smaller block sizes. However, we could still prune about 500 blocks per layer. Hence, we added a cache with 512 entries to our model for comparison, and Accelergy returned 1.73 % extra area, 2.75× less. The third reference accelerator, SCNN, has a compression module that adds 24.8 % area. With its 4×4 PE array, we found a threshold combination that leverages 348 764 sparse blocks in ResNet-50 or 6707 per layer. This configuration results in 26.6 % pruned sparsity with only 1 % accuracy degradation. With a cache that stores up to 8196 entries, we estimated the area overhead of Sparse-Blox using SCNN's accelerator configuration to be only 4.95 %, which equals to 5× less.

Looking at the area breakdown, the largest share of Sparse-Blox accounts for the cache, since it has to hold all sparse block positions in between layers. To have sufficient space, the caches for SNAP, STICKER and SCNN, can store up to 6144, 512 and 8196 16-bit positions, respectively. Even if regular activation sparsity exploits slightly less compared to irregular sparsity, we demonstrated a reduction in the number of computations and memory transfers by on average 25 %. At the same time our approach, has 5× less area overhead compared to three state-of-the-art accelerators.

5.6.5 Discussion

Looking at the evaluation results, we gained a lot of insights on CNN workloads regarding sparsity and the effects of pruning. We saw that, for example, ResNet-50 can benefit largely from accelerators that leverage the present regular activation sparsity. However, YOLOv5s showed much less sparsity, due to the much smaller dimension of feature maps. As a result, the information in this feature map is denser, and the corresponding layer is more sensitive to pruning. But we also saw during our evaluation that our threshold-based method to increase the sparsity can improve the efficiency. With our tool Spex, we were able to increase this regular activation sparsity systematically, while maintaining accuracy.

To bring our contribution in line with the state-of-the-art works that looked at dynamic pruning of sparse input activations, we carried out a qualitative analysis of all works, which is given in Table 5.3. Throughout all studies, positive effects of pruning on energy efficiency and performance were reported. Most works focus on irregular sparsity, since it yields the biggest performance gains, especially when it can be

	Pruning Threshold	Kind of Compression	Notes
Cnvlutin [261]	✓	none	
EIE [238]	✗	irregular	Only MLPs
SCNN [151]	✓(only W)	irregular	Index-based encoding overhead
Cambricon-S [263]	✓	2×2 blocks	
STICKER [279]	✓	irregular	Thresholds for sparsity modes
SNAP [280]	✗	irregular	Sophisticated encoding through search
Ours [Hot+23a]	✓	arbitrary blocks	Adjustable per-layer thresholds

Table 5.3: Comparison of different DNN accelerators that support dynamic pruning.

compressed to reduce memory transfers. Obviously, there are works not listed here that investigated various forms of regular pruning [241; 321]. Only Cambricon-S [263] from the related work investigated, how sparsity in regular blocks impacts the performance. But, since they can apply different block sizes for each layer, their decoding mechanism adds a lot of chip area. Our caching approach only prunes blocks of a fixed size. However, it reduces the architectural complexity significantly.

This could also be shown while looking at the area distribution of state-of-the-art DNN accelerators using irregular sparsity compared to the same architecture with Sparse-Blox in place. Here we saved around 25 % of all computation while demonstrating a 5× lower area overhead. However, it has to be noted, that the irregular sparse accelerators may prune more activations. For reference, an unmodified ResNet-50 has around 53 % irregular sparse activation values. Considering these results, we saw that, even if the amount of sparsity exploited is about 2× less, we only had to add a fraction of the area overhead. Depending on the tolerable accuracy degradation, the exploited sparsity can, however, be increased.

To investigate this complex pruning threshold to accuracy trade-off, we also introduced Spex. It is typically not covered by the state-of-the-art works, how exactly the thresholds were defined. Our work covers this optimization problem in-depth, by using Spex for all kinds of workloads with different regular pruning mechanisms. To allow for further research we open-sourced Spex and Sparse-Blox.

5.7 Conclusion and Outlook

The high energy consumption of modern DNN topologies is one of the most challenging obstacles for bringing them to embedded systems. As a result, many optimization strategies have been explored in numerous studies. One promising technique is pruning, that eliminates unnecessary computations with weights or activations that are close to zero and hence do not contribute to the result.

In this chapter, we investigated one specific pruning technique thoroughly, namely dynamic and regular activation pruning. Since regular sparsity can be leveraged in hardware in a simple manner and activation pruning yields the best energy efficiency gains. Therefore, we presented Spex and Sparse-Blox, a tool and a hardware extension to prune regular activation sparsity in DNNs. We focus on regular sparsity in the form of blocks that match the dimensions of the hardware accelerator. Hence, online pruning inside a hardware accelerator can be realized in a straightforward and inexpensive way. To increase regular sparsity, Sparse-Blox can prune blocks of activations based upon a per-layer threshold.

Since increasing sparsity and pruning it has an impact on the accuracy, we also presented Spex. It explores the large design space of threshold values that opens up for state-of-the-art DNN workloads. With Spex, we can get a lot of insights of the workload, like which block size is favorable, which layer can tolerate more pruning or what dataflow is the most efficient. Once a beneficial threshold combination is found by Spex and pruning in hardware realized by Sparse-Blox, we can profit from structured pruning. It saves many MAC operations and, more importantly, energy expensive off-chip memory transfers. With the two contributions, we conducted extensive experiments on benchmark state-of-the-art CNNs. Considering a ResNet-50, for example, we could create up to 15.2 % new sparse blocks using a block size of 8×8. In combination with already present sparsity this equals to a reduction of 814 million MAC operations and 19.1 % of the total inference energy. We also compared our introduced hardware overhead for exploiting regular sparsity by Sparse-Blox with state-of-the-art DNN accelerators that exploit irregular sparsity, showing a 2.5 to $5\times$ smaller area overhead. Sparse-Blox offers an innovative design methodology between exploited sparse operations and area overhead.

For the future, there are still some open questions when it comes to pruning and sparsity in DNNs. For example, it is conceivable to increase the number of sparse blocks even further when training in particular for this kind of sparsity. Many studies

have already investigated regularization methods [161] to increase regular sparsity. As a result, we project that one can benefit even more from this kind of pruning. Moreover, one can think of more advanced methods to compress the sparse blocks. For example, a smaller block size may be applied, and the accelerator and framework extended to support clustering or merging of individual small blocks.

Chapter 6

Exploiting Mixed-Precision CNN Inference to Increase Robustness and Energy Efficiency

So far we looked at various aspects of Deep Neural Networks (DNNs) and how we can make them more efficient in terms of energy consumption and throughput. Co-Designing accelerator and workload as well as leveraging and increasing present sparsity in workloads both appear to be great choices to achieve those goals. Yet, we have not really discussed another very popular optimization strategy for DNNs: quantization.

Like pruning sparse areas in feature maps, quantization can save energy by compressing the numerous off-chip memory transactions and by making Multiply-Accumulate (MAC) computation simpler. Quantization has been shown to open large opportunities for efficiency boosts, while only having small impact on the network accuracy, consistently over multiple studies. The contributions covered in this chapter, will investigate two interesting aspects of quantization. First, we elaborate how quantization can actually reduce the energy consumption of DNN inference, especially when applying mixed-precision. Moreover, we look at robustness in quantized Convolutional Neural Networks (CNNs), e.g., how a quantized network that delivers a high accuracy reacts to perturbed inputs like rain. In case the accuracy is impaired by the perturbation, we introduce a systematic method to recover the accuracy to baseline [Hot+22a; Hot+23b].

6.1 Overview, Introduction and Motivation

The major advancement of DNNs in the past two decades have made those algorithms indispensable for a myriad of modern applications. Assistant robots or consumer products rely on CNNs. Even strongly safety-critical tasks like autonomous driving [Hoe+23b] use them for camera based environment perception, since it is impossible to fully specify a traditional algorithm. For example, there are just too many representations of a single vehicle that is present on a road. CNNs learn this representation from a reasonable number of samples. However, to deploy them into constrained embedded system still two challenges persist today. The first challenge was already discussed at multiple spots in this thesis, namely the vast number of computations and energy-intensive memory transactions [115] that make it hard to achieve real-time behavior. Considering image segmentation tasks using CNNs, which are essential to environment perception, the computational load is even higher compared to classification task.

The second challenge, however, has not been touched yet, which is a trustworthy inference. Trustworthiness by itself comprises two parts [322]. First, security, which is to protect the system from intentionally provoked unwanted behavior and prevention of unauthorized information extraction. Second, dependability, which is the ability of a service to fail less often than acceptable [322]. Dependability again has multiple attributes like availability, reliability or safety, which is defined by the ISO standard 26262 [323]. According to the standard, functional safety can be formulated as 'absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E systems' [323, Definition 3.67, p. 14]. A further component of dependability is Safety Of The Intended Functionality (SOTIF). Following to the ISO standard 21448, it is defined as 'absence of unreasonable risk due to hazards resulting from functional insufficiencies of the intended functionality or its implementation' [324, Definition 3.25, p. 8].

Especially for autonomous systems that act in the real-world exist many cases in which we have to ensure that, e.g., a fully camera-based perception works as indented by the specification [325]. As hard as it is to formally describe all representations of an object, it is also hard to cover all situations in a training dataset for a DNN. For example, when it is raining, the camera-based environment perception system should nonetheless detect all potential obstacles and all other entities [326]. This is referred to as robustness, which is defined by the ISO 26262 for hardware as following: 'robust-

ness is the ability to be immune to environmental stress and stable over the service life within design limits' [323, Definiton 1.100, p. 14]. Regarding Machine Learning (ML) algorithms like CNNs the ISO 21448 norm specifies robustness especially as the occurrence of inputs that were not present in the training dataset. This is the case when the CNN input is corrupted by environmental factors, like weather conditions or brightness variations. In contrast, if it is maliciously created, it is referred to as adversarial attack and thus a security issue. While the human vision with semantic context is very robust to such input corruptions, CNNs are still struggling with that and result in less model accuracy.

To address the computational load of CNNs, one can apply common approaches like pruning or algorithm and accelerator co-design, which we both introduced and thoroughly discussed in the previous chapters. Furthermore, another method to compress CNN inference and reduce the computational complexity is quantization. In the state-of-the-art section, we already discussed that quantization if an effective method to improve the efficiency of DNNs. Usually, those workloads tolerate imprecise computation of operations with only a slight accuracy degradation. Instead of computing everything in 32-bit floating point, we can use, e.g., 8-bit fixed-point integers and hence save energy and area during the computation. Additional energy can be saved during inference when we compress off-chip memory transactions [281]. Moreover, we can use the local memories more efficiently, since the memory footprint of, e.g., eight over 32 bits reduces significantly. The freed local memory may now, be used to keep more data local to the accelerator. However, strong quantization like strong pruning has of course an impact on the accuracy. To benefit even more from this compression and mitigate the accuracy degradation, we can use mixed-precision quantization schemes that allow to apply any precision on a layer-granularity, even individually for weights and intermediate results. Typically, some layers are more sensitive to radical quantization, while others can tolerate less precise computation. Those layers may be computed with less precision, while quantization sensitive layers may be executed more accurately. As a result, we can achieve a high compression and a high model accuracy at the same time. This kind of compression allows us to save numerous off-chip memory stores and reads.

However, the two aforementioned challenges are somewhat interlinked. Quantization with mixed-precision achieves high compression with only a slight accuracy degradation. While this can be tolerated in some applications, the robustness of a DNN can be troubled by the imprecision quantization or pruning superadd. This effect worsens when the input gets corrupted, e.g., by unforeseen changes in bright-

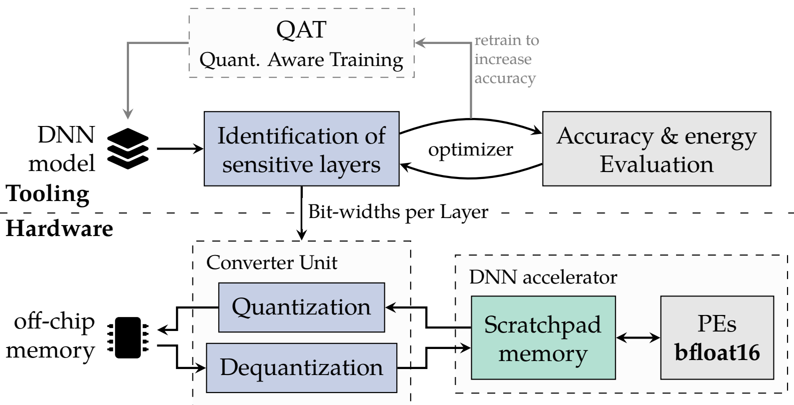


Figure 6.1: Overview how our tool finds an optimal set of bit-widths for mixed-precision DNN inference in an embedded HPC SoC with a bfloat16 accelerator.

ness or weather conditions. To mitigate this effect, adjusting the precision may be leveraged. For example, the network may be executed with a higher precision when it is subject to those input corruptions or perturbations to restore the initial model accuracy. According to the robustness definition, mixed-precision might allow us to move some inputs from a hazardous operation into the non-hazardous area.

In summary, we project that dynamically adjusting computational precision enables more flexibility while quantizing a DNN and hence allows bringing together a high accuracy with an increased robustness and energy efficiency. Energy can be saved by compressing off-chip memory transaction, as shown by other state-of-the-art works. While using mixed-precision only for more energy-efficient inference is not entirely novel, looking at robustness at the same time has not been investigated so far. Robustness is increased by adjusting the precision depending on perturbation. For example, we can work with a higher degree of quantization when facing clear weather conditions to benefit from higher compression. In turn, we can apply less quantization when dealing with uncertain situations, e.g., fog or heavy rain, to regain model accuracy and still benefit from compression.

In the following sections, we will first have an in-depth look on how we modified Spex to also explore the impact of quantization with perturbed inputs on the model accuracy. Then we discuss the design of our hardware number converter that is

hosted on a reconfigurable fabric. An overview of how those two components interact and how our envisaged design flow looks like is given in Figure 6.1. An optimizer based on Spex looks for the best quantization bit-widths for each layer that maximizes model accuracy and energy consumption. The validation dataset can be artificially corrupted to also evaluate the quantization effects on perturbed inputs. Then the quantized network can run on a target platform that has a converter unit connected in between the DNN accelerator and the off-chip memory, to make the increase of robustness and energy savings possible. To demonstrate that mixed-precision in fact supports the model robustness as well as the energy efficiency, we evaluated our approach with three image segmentation CNNs under the influence of various weather conditions.

6.2 Our Mixed-Precision Concept for Increased Robustness and Energy Efficiency

As we saw during the motivation, especially mixed-precision quantization looks promising to address the two major challenges of bringing CNN inference to autonomous and embedded systems. In this section, we will sketch our mixed-precision concept to increase robustness and energy efficiency of modern embedded platforms with fixed DNN accelerators. Significant energy savings can be achieved by compressing integer values and thus reducing the off-chip memory transfers. Moreover, in such an envisaged Computer Vision (CV) system that is responsible for a safety relevant task like environment perception, we have to ensure robustness towards corrupted and perturbed inputs.

The goal of our approach is to respond dynamically and quickly to changing conditions. Compared to traditional approaches like deraining techniques, which add many more computations to the workload [327], we project that we can leverage mixed-precision inference to increase the network robustness. If our system, for example, receives such an input, we can use a set of weights with a higher precision and apply less quantization to intermediate results going to the off-chip memory. Since the memory consumption of CNN weights in comparison to its intermediate results is moderate [328], we can easily store multiple sets of weights in a reasonable large off-chip memory. In short, the network can run with a higher precision, in case the input is affected by perturbations. At the same time, we can still benefit from

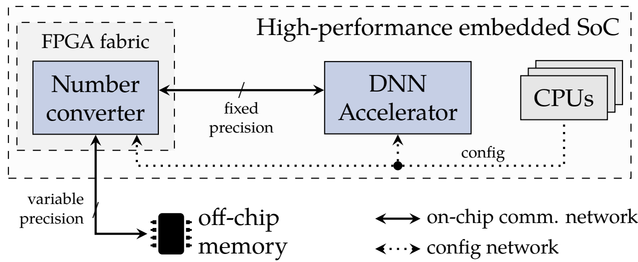


Figure 6.2: Overview of our envisaged dataflow. A number converter enables mixed-precision DNN inference to increase robustness and energy efficiency through compression of off-chip memory transactions. Therefore, the DNN accelerator itself does not need to be altered.

the energy reduction through compression. It has to be noted, that in the case of very severe weather conditions, we may have to add a deraining method as well. However, in this case, we may also be able to save energy in the computations of less quantization sensitive layers.

As stated before, our target platform is an embedded high-performance System-on-Chip (SoC), which can manage tasks like autonomous driving. Such platforms systems typically offer Central Processing Units (CPUs), a DNN accelerator and a reconfigurable fabric to allow for further extension, like e.g. AMD's Versal [329] or the SoC build by the European Processor Initiative [132]. However, mixed-precision or even just two quantization steps are rarely supported in Commercial-Off-the-Shelf (COTS) systems, since they add heavily to the hardware resource budget. Approaches like bit-serial accelerators [223; 226] are one method to implement mixed-precision with support for arbitrary precision, but add much to the latency. More often we can find accelerators that support, for example, bfloat16 as number format, which offers a great trade-off between efficient inference and the capability for training DNNs.

Yet to enable mixed-precision in such systems, we can harness exactly this reconfigurable fabric. Therefore, we introduce our hardware extension and Design Space Exploration (DSE) tool that bring high-efficient CNN inference and robustness against perturbed inputs closer together. Our concept and how we alter the dataflow for the DNN inference is shown in Figure 6.2. Quantization is enabled by a number converter that has access to the on-chip communication network. It dynamically quantizes and dequantizes dataflowing between the on-chip DNN accelerator and the off-chip memory. In particular, the off-chip memory stores integer values with

arbitrary precision and the DNN accelerator operates with bfloat16 operands. The flexible precision adjustment balances model accuracy, robustness, and energy efficiency. As a result, we also preserve the capability of online network fine-tuning in bfloat16, when bypassing the number converter. This approach increases flexibility and can bring mixed-precision inference to many platforms, without changing the DNN accelerator itself.

For mixed-precision inference, we also have to identify the right degree of quantization that maximizes energy efficiency and accuracy. This process has to be done for different perturbations to study whether the right precision can restore the original model accuracy. Hence, we also present our DSE tool to automatically determine the precision levels. Since the exploration is to some degree similar to the sparsity exploration we looked at in the previous chapter, we base and integrate our tool into Spex (see Section 5.4). This allows also for combination of quantization and pruning.

6.3 Related Work

As stated before mixed-precision and more generally quantization is a very active research field, with numerous published works looking into the various aspects. The state of the art is surveyed more generally in Subsection 3.3.2. In this section, we will particularly focus on architectures that support mixed-precision to optimize the energy efficiency, tool to identify the best quantization scheme, as well as on approaches to increase the robustness of CNN inference when it encounters perturbed inputs.

6.3.1 Hardware Architectures for Mixed-Precision Inference

Mixed-precision inference is a well-known method to optimize the energy efficiency of DNN inference by compressing those off-chip memory transfers [330]. Common CNN hardware accelerators typically only support a fixed bit-width and therefore cannot put the energy efficiency gains achieved by variable quantization into effect. Approaches presented by Han et al. [239] or Chen et al. [267] combine trained quantization with compression methods to achieve a better inference efficiency. However, their early works add large additional hardware resources for implementation and do not support per-layer quantization.

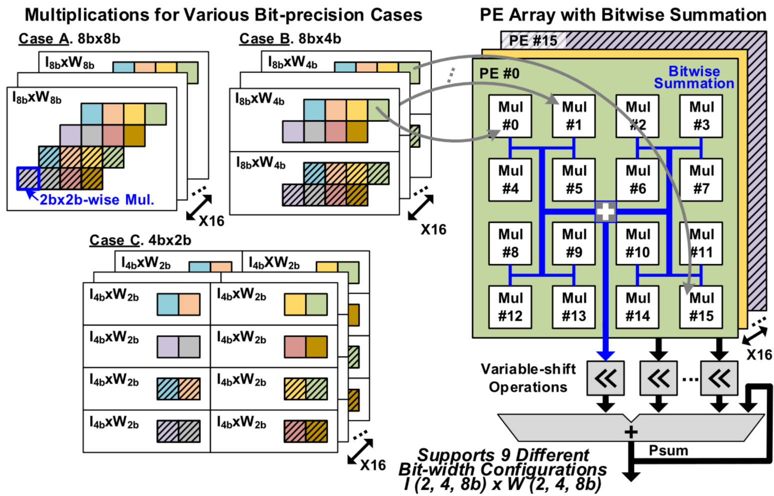


Figure 6.3: Overview of the Processing Elements (PEs) in BitBlade presented in [283] that realize multipliers with variable precision.

A few hardware accelerators have been proposed to support arbitrary precision CNN inference. Bit-serial accelerators are a straightforward method to implementation mixed-precision. Notable examples are Stripes [223], Loom [224] and UNPU [225], which are all Application-Specific Integrated Circuit (ASIC) implementations, as well as BISMO [226], an implementation for reconfigurable devices. The latter outperforms Graphics Processing Units (GPUs) with about 6.5 T OPS on a small Xilinx PYNQ-Z1 board. Although bit-serial architectures provide a simple interface for different bit-widths, they introduce additional hardware overhead and, in particular, suffer from lower energy efficiency compared to non-bit-serial architectures [331]. Besides bit-serial accelerators, some works have investigated architectures that allow for flexible rerouting or power-gating of individual multipliers. Envision [332] uses dynamic-voltage-accuracy-frequency scaling (DVAFS) which allows disabling parts of multipliers based on the targeted bit-width. With their approach they can switch from 16- down to 4-bit and significantly reduce the energy consumption. However, this approach suffers from a significant underutilization of MACs and not all quantization levels are available. Bit Fusion's [333] architecture consists of multiple single bit PEs that can be connected to build arbitrary bit-widths. In comparison to the bit-serial accelerator Stripes, BitFusion can reduce the energy consumption and increase the

performance by 4.0× and 2.6× on a ResNet-18 workload, respectively. However, their flexible routing infrastructure to enable connection of individual PEs adds area to the architecture and thus can be outperformed by traditional accelerators. BitBlade [283] tries to address the issue of underutilization and area overhead by using a bitwise summation and a tiling scheme, which improves the utilization of multipliers and makes the fetching from off-chip memories more efficient. Figure 6.3 shows the precision-scalable PEs of BitBlade, which allows grouping of single-bit multipliers for arbitrary precision. The accelerator is based on the Bit Fusion architecture, but the authors apply a series of algorithmic and architectural optimizations to overcome area and power overheads. With BitBlade they report 52 % less area compared to Bit Fusion and 4× the throughput. In comparison with other hardware accelerators, they show 7.7× performance and 1.6× energy efficiency improvement across three workloads with different configurations.

As stated before, hardware support is only one step to mixed-precision inference, but the actual bit-width for each layer has to be determined. Hence, multiple studies investigated the interplay of hardware and quantization. HAQ [281] by Wang et al. explores per-layer quantization for bit-serial accelerators using Reinforcement Learning (RL). With their approach they show a reduction in latency of up to 1.9× with almost no accuracy loss compared to uniform and equal quantization for each layer. While HAQ supports only a limited space of precision, ReLeQ [334] by Elthakeb et al. allows for more bit-widths and achieves a 1.22× performance improvement over Stripes. However, ReLeQ is limited to weight quantization only. Both, HAQ and ReLeQ are tailored to bit-serial accelerators and cannot be applied to other hardware architecture without modifications or additional components.

To address this, Wu et al. [335] use gradient descent with is more efficient than RL methods. Their work can also be applied to any kind of accelerator. The authors use a super network to sample network topologies in which Convolutional (CONV) layers are quantized and analyze their number of FLOPS and accuracy. A ResNet-34 workload trained on the ImageNet dataset can be compressed 19× with less than 1 % accuracy loss. For increased computational efficiency in searching the best bit-widths, Chen et al. [336] state the search problem as constrained optimization problem. By approximation with Taylor expansion and efficient computation of the Hessian matrix, they can quantize a ResNet-50 by 12.24× and 8× for weights and activations, respectively. This outperforms approaches like HAQ or PACT [221].

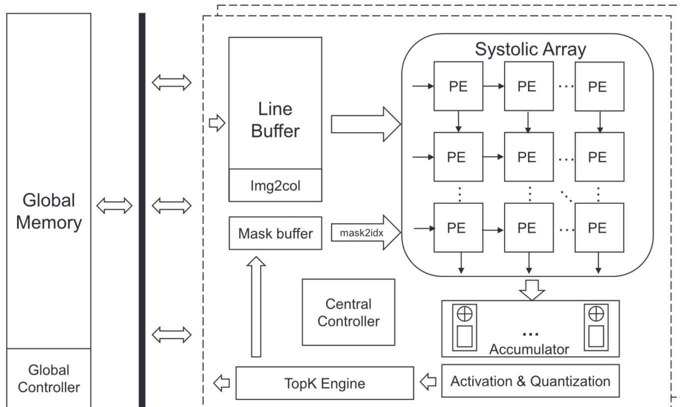


Figure 6.4: Overview of the Structured Dynamic Precision (SDP) accelerator, presented in [284].

Some works looked into co-design approaches that take both the hardware accelerator and the search for quantization levels into account. For example FILM-QNN [337] by Sun et al., which focuses on an implementation on Field-Programmable Gate Arrays (FPGAs) using High Level Synthesis (HLS). Their approach leverages efficient value packing into the available Digital Signal Processors (DSPs). FILM-QNN realizes per-layer mixed-precision and achieves accuracy comparable to a 32-bit inference. Thereby, it can achieve up to 537 FPS for MobileNetV2 on a ZCU102 platform, outperforming state-of-the-art works. Another co-design work looking at HW-NAS with focus on quantization and targets FPGAs is AnaCoNGA [282]. Their work uses a Genetic Algorithm (GA) to determine per-layer precision for a bit-serial accelerator. Compared to HAQ, AnaCoNGA can improve the accuracy considering the same workloads. In addition, the exploration of the design space takes about 24 % less time.

For more generic architectures the recent work by Huang et al. [284] introduces Structured Dynamic Precision (SDP), which is a co-design approach for fine-granular mixed-precision to increase the accuracy and speed up the inference. Their work is inspired by fine-granular but regular magnitude pruning approaches. Similarly, the authors try to identify important regions in feature maps and compute only those. Therefore, they are quantized to 8-bit and grouped into 4-bit blocks, which are selected based on a non-zero Top-K algorithm. Since their quantization method is highly specific, they also present a corresponding modified systolic array accelerator, which

is shown in Figure 6.4. With their approach the authors demonstrate almost baseline accuracy, higher than previous works like DRQ [219]. Looking at the inference performance of a ResNet-18, the SDP approach shows about 29 % improvement over DRQ. Due to reduced precision, SDP can also reduce Dynamic Random Access Memory (DRAM) accesses by up to 45.7 %.

6.3.2 Robustness-aware Quantization of CNNs

Robustness towards perturbed or corrupted inputs is one of the main challenges that still exists before CNNs can be deployed in safety-critical systems such as computer vision tasks in autonomous driving vehicles [338]. Hence, some research has been carried out to address robustness issues of CNNs using various strategies at different system levels. Zendel et al. [339] surveyed in-depth potential hazards in CV systems. In particular, Kamann et al. [326] give a great overview of mitigation strategies for image corruptions.

Leaving hardware aside for now, most studies have looked at increasing the robustness of CNN models against adversarial attacks [340]. However, some research also investigated common image corruptions. Geirhos et al. [341] investigated how CNNs and human vision respond to twelve different corruptions. Thereby, they found that a large ResNet-152 is more robust than GoogLeNet. Vasiljevic et al. [342], in particular, looked at blurry inputs to segmentation and classification CNNs and found that they make unreliable predictions in this case. To mitigate this, they demonstrated that retraining a few iterations with unsharp inputs boosts the robustness of common network topologies. Another frequently addressed topic is rain removal, as this is one of the most common perturbations that occur in real-world scenarios. Obviously, there are many ways to handle rain [343], but we will focus on methods that employ Deep Learning and target single images, similar to our approach. For example, the deep detail network (DDN) by Fu et al. [344], which improves a prior work of the same authors that proposes a CNN that learns a mapping between clean and rainy inputs. This network may then be used to restore the original input. Their work shows large improvements over other deraining techniques in terms of execution time and restoration quality. However, a major issue in such approaches is often unpredictable pattern of rain and especially rain streaks. To address this, Wang et al. [289] present a two-parted approach. First a method to build a comprehensive dataset that covers various rain aspects and second the spatial attentive network (SPANet), which aims to reduce the rain in such inputs. Their approach outperforms up till then state-of-the-

art deraining techniques. Today, modern methods often use Generative Adversarial Networks (GANs) for rain removal, like Semi-MoreGAN [288] by Shen et al. Their network reaches up to 58.9 G FLOPS and thus is quite fast. For evaluation, it is assessed using synthetic and real-world samples, and thereby outperforms state-of-the-art works.

With focus on aspects like compression or quantization, Wijayanto et al. [285] explored the impact of compression on CNN robustness through quantization. Thereby, they use adversarial training to mitigate corresponding attacks. Their algorithm achieves high compression rates for classification tasks, up to 90× for AlexNet, and also reduces the error rate of the model compared to other methods. Similarly, Khalid et al. [286] use quantization to increase the robustness of CNNs against adversarial attacks. The presented DNN is quantized both statically before inference and trainable, which iteratively is applied during the training phase. Therefore, they add additional quantization layers at the CNN input. For CIFAR-10 they can demonstrate up to 50 % better classification accuracy when the inputs are perturbed. Besides those works, Zhao et al. [287] increase the robustness of a DNN through outlier-aware quantization during training. In particular, they look at the non-maximum suppression stage of oriented object detection networks. On a MobileNet-SSD trained on the Airbus remote sensing dataset, they can demonstrate a better performance on outliers compared to post-training quantization methods.

6.4 Toolchain to find the Best Precision for Robust Mixed-Precision CNN Inference

Our approach enables quantized DNN inference on state-of-the-art heterogeneous platforms like AMD's Versal or the SoC that is developed by the European Processor Initiative (EPI), which usually come with a fixed DNN accelerator. With mixed-precision we aim to increase not only the energy efficiency of the otherwise energy-hungry DNN inference, but also to increase the robustness to unforeseen perturbations to the inputs. The latter is especially important in our envisaged domain of application, namely CV based environment perception that has to deal with all kinds of weather conditions, brightness changes and vision obstructions. However, identification of the right precision for each layer is not an easy task. Similar to the threshold based sparsification described in the previous chapter, a stronger quan-

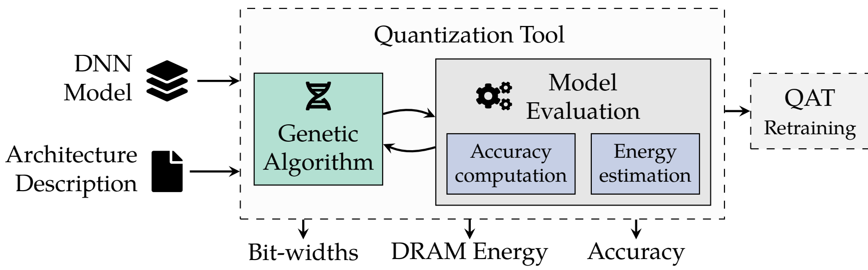


Figure 6.5: Overview of our tool to identify an optimal set of bit-widths for mixed-precision DNN inference. Based on a given DNN model and architecture design, the tool evaluates both the model accuracy and the required energy.

tization of a given layer also affects the result of the subsequent layers in the DNN. In addition, quantizing large layers have a higher impact on the energy needed for memory transfers, as more offloading happens. Hence, we have to look at the entire network and using a significant validation dataset. Only by running an inference and observe the impact of different layer precision, we can determine the accuracy.

As indicated before, the problem description is quite like the dynamic pruning approach addressed by Spex and Sparse-Blox. Also, the objective to reduce the energy needed for off-chip memory transfers is the same. Exactly this aspect is exploited by us while designing a tool to determine the right level of quantization for each layer in a large CNN. Many components of Spex can be reused for this quantization problem like the DSE abstraction layers, accuracy estimation, energy estimation using Timeloop and Accelergy and to some extent the altering of models in PyTorch. An overview of the tool is given in Figure 6.5. Obviously, it looks familiar to Spex. However, we added a considerable number of components to the exploration tool.

6.4.1 Design Space Exploration

Like Spex, we use an exploration algorithm that suggests a set of bit-width with which a base model is quantized and evaluated using a validation dataset. During inference, our DSE algorithm tries to optimize two objective functions, while complying with one inequality constraint. All of them are given in Equation 6.1. The first objective function $f_1(x)$ aims to maximize the model accuracy, which is directly generated by the evaluator. Accuracy is also checked by the first constraint $g_1(x)$ that dismisses all

solutions that do not meet a certain minimum accuracy that is set by the designer. Like Spex' objective functions, all objectives are negated as most optimization algorithms minimize objectives than maximizing them. Second design objective is to minimize the overall energy consumption. This is reported using a hardware model, which will be explained later. The actual degree of quantization is not considered as optimization objective since it is indirectly encoded in the inference energy.

$$
\begin{aligned}
f_1(x) &= -max(\text{accuracy}) \\
f_2(x) &= E_{inference} \\
g_1(x) &= accuracy_{min} - accuracy
\end{aligned}
\tag{6.1}
$$

Architectural characteristics of our number converter are not directly included in the set of objective and constraints. As we will see later, the number converter is designed to use only a small area and to add only little latency. Moreover, these aspects are in general independent of the quantization degree.

To make a comprehensive DSE feasible, the evaluation of individual bit-width combinations has to be fast, goal directed and constrained in the right way. Especially, since the design space for segmentation CNN workloads is typically much larger than for the workloads evaluated for pruning. For example, a modern CNNs for environment perception using semantic segmentation like YOLOP [86] features 102 convolutional layers. Considering individual bit-widths for weights and activations, it has 204 values that have to be determined. Hence, a quantized inference should only add very little to execution time. To shrink the design space from the beginning on we should be able to set a maximum and minimum precision for each operand.

So far, we have looked at compression to increase the energy efficiency during inference, but mixed-precision for increase robustness is not yet covered during DSE. Therefore, we add another variable to the DSE that sets the kind and severity of a perturbation to the input. When this variable changes, we have to rerun the optimization process to find a Pareto-optimal precision for each layer that maximizes off-chip memory transfer compression and model accuracy. The design variable that controls the intensity and kind of input perturbation has to be explored exhaustively as we have to see the impact of each. At the end, out tool returns quantized weights for each perturbation with different severity. Those can be used depending on the quality of the outputs. If for example, the accuracy degrades, we can load the corresponding set of weights.

However, it can happen that our exploration does not yield any solutions that meet the required accuracy constraint, for example, when the perturbations are too strong. For this case, we also added extra infrastructure that can perform Quantization Aware Training (QAT) after the exploration has finished. Since we only use Post-Training Quantization (PTQ), retraining a selection of the best Pareto-optimal solutions over a few epochs with the initial or an extended dataset helps to increase the accuracy, when it is not sufficient. But it is also conceivable to retrain all other found solutions and potentially reduce the precision even further to benefit from compression even more.

6.4.2 Implementation details

The goal is to integrate determination of the quantization level seamlessly into Spex without getting rid of any features needed to explore the possibilities of pruning. This should allow us to combine both pruning and quantization in a single tool later. As stated before, some components of Spex are left untouched, while others need to be adjusted for the search of quantization levels.

Primarily, we need support for quantization in DNN workloads. To reflect bfloat16 computation, we set the model to half precision in PyTorch, which tells the GPU used for infernce to use bfloat16 representation. Like Spex, we then alter a PyTorch model and replace all layers with counterparts that feature a forward function that enables quantization. In particular, we use uniform quantization as it is very simple to realize in hardware, together with fake quantization, which is very quick in terms of evaluation during inference, which allows us to test a quantization scheme with many validation samples. Fake quantization keeps all values in the original half precision format but splits the range of the floating-point numbers into bins according to the number of quantization steps, which are available with a certain precision. Corresponding scale factors and step widths are determined by a calibration process that has to happen in advance to determine the numeric range of all feature maps and weights. Since fake quantization represents a similar behavior as the actual quantization in hardware and can be implemented using two simple number conversions, it is easy to integrate into our tool. For that, we use the quantized layer implementations from the TensorRT library[13] by NVIDIA.

[13] https://github.com/NVIDIA/TensorRT/tree/release/9.1/tools/pytorch-quantization, accessed 2023-10-27

With this DNN model that support arbitrary precision on a layer-granularity, independent for weights and activations, we can start with the actual DSE. To allow for compatibility with Spex and with the exploration of sparsity, we kept the inputs to the tool the same. One has to provide the base DNN model together with a validation dataset on which the exploration happens. Next, a workload description file has to be given, which is also kept similar (see Listing 5.1 for reference), since most parameters like the settings of the exploration algorithm or the minimum accuracy are still used.

To reuse Spex for the layer precision search, we change the `problem_function` at the beginning of the workload description file. When this is set to `quantization_problem` instead of `sparsity_problem` the initialization of Spex works slightly different. The given input model is updated with fake quantization layers instead of counterparts that support pruning. As bit-widths for activations and weights have of course to be natural numbers the decision variable space has to be initialized with those numbers only to yield only natural input variable vectors for the evaluation function. As a result, we have to ensure that $\vec{x} \in \Omega = \{x \in \mathbb{N}\}$. To allow for more flexibility, we also add a bit weighting function that can for example shift the focus for quantization from weights to activations. This is especially worth considering when looking at CONV layers that have orders of magnitude larger feature maps than the number of weights. Hence, more compression can be achieved by putting more emphasis on the activations. This feature, as well as a range of bit-widths that constrains x into a certain interval can be controlled using the `extra_args` in the workload description file.

Since we want to look into the energy savings during inference and not solely at number of bits, we have to add another step before starting the exploration process. Reducing just the number of bits can result in misleading results, for example, when not all intermediate results have to be offloaded. Instead, a non-negligible share of values can stay in the local memory and be reused later. Hence, we have to figure out how many actual off-chip memory transactions are performed in a given layer and DNN workload. Therefore, we first create a mapping of the DNN workload onto the bfloat16 DNN accelerator using Timeloop [163]. Timeloop is an established tool that aims to find an ideal mapping. Since Timeloop can only evaluate a single layer, we run all layers in a row and collect the relevant figures, like the off-chip memory transfers and the corresponding energy consumption. Based on this it reports energy for all kinds of operations, including the needed off-chip memory transfers. Obviously, the used hardware model and libraries for Timeloop determine the accuracy of the energy estimation. To make the results comparable, we use the same model as for

Spex (see Subsection 5.4.3), with an energy estimation generated by Accelergy [169] coupled with CACTI [194]. With the energy, which is required for bfloat16 operation and memory transactions, we can estimate the energy for quantized inference. For example, when a layer can work with 6-bit weights and 8-bit activations, we can multiply the baseline energy at bfloat16 with the ratio of quantization. The resulting energy for all weight and activation transfers are accordingly $6/16 \cdot E^{layer}_{weight_move}$ and $8/16 \cdot E^{layer}_{offload}$.

Once we have the mapping result, we can proceed with the actual search for a set of bit-widths that optimizes accuracy and energy savings. Therefore, we use, just as for the sparsity threshold search described in the previous chapter, a GA. Most parts of the actual exploration are then quite similar. The algorithm starts with an initial population of individuals, each element in their genome represents a bit-width for activations or weights of a specific layer. Using goal-oriented recombination the GA runs over multiple generations and ideally settles at an optimal precision for each layer. Thereby, each proposed solution of the GA is evaluated using a significant fraction validation dataset to get the model accuracy. In addition, based on the mapping and the scaling, the required DRAM accesses for a single inference are computed.

When the exploration has terminated, we can evaluate the results using various visualizations like plotting the objective functions. Moreover, we can select a number of the best solutions and perform QAT to increase the model accuracy further. Here the quantized model is fine-tuned using the training dataset and a comparability small learning rate to not accidentally worsen the accuracy.

To account for various perturbations and to prove our claim that quantization also supports the model robustness, the exploration does not stop with a single evaluated and retrained model. On the contrary, we run the entire DSE process for various perturbations with different intensities. As a result, we do not get a single set of quantization levels, but multiple for each condition. The degree of quantization and the corresponding weights can be exchanged dynamically on the accelerator later. To get comparable results for various perturbations, our exploration GA always starts with a fixed starting population and applies the same crossover and mutation algorithms in all DSEs.

6.5 Hardware Support for Mixed-Precision Inference

The centerpiece that enables mixed-precision DNN inference for modern SoCs, is a hardware component for number conversion that alters the dataflow between the off-chip memory and the DNN accelerator. This number converter has to support a fast and low-latency conversion. Since it will be hosted on the reconfigurable fabric of a SoC it also has to be resource efficient to keep enough spare resources for other applications. To support off-chip memory movement from and to the local scratchpad memory, our component should consist of a quantizer to convert numbers into integers and a dequantizer that transforms them back into bfloat16 format. An external controller has to be able to dynamically and quickly adjust conversion formats, either in between layers or when the input is perturbed. The controller can trigger more precise computation, e.g., when external sensors detect a weather change. From an arithmetic viewpoint, inputs and weights have to undergo a conversion according to the following equations, in which x_q is the quantized and x_{float} the dequantized value. S and Z denote a scale and zero-offset factor.

$$x_q = \frac{x_{float}}{S} + Z \qquad\qquad x_{float} = (x_q - Z) * S$$

To make the computation in hardware much simpler, zero-centered uniform quantization is applied. Hence, the zero-offset factor is zero and can be neglected. Moreover, we can transform the division during quantization operation into another multiplication. As a result, the conversion operation narrows down to a multiplication with a scale factor. This can be realized in hardware using a single multiplier, which can be parallelized and pipelined in a straightforward manner to meet the aforementioned high throughput and low latency goal.

Details of our dataflow involving our number converter component along with all the necessary additional components are given in Figure 6.6 and Figure 6.7 for the dequantizer and the quantizer, respectively. All dashed boxes are placed onto the reconfigurable fabric. For an online conversion of data, we have direct access to the on-chip communication network, allowing us to redirect the data stream between the actual DNN accelerator and off-chip memory through our number converter architecture. Here, connection to the on-chip network is realized using the widely used AXI-Stream interface. Integer precision and scale factors, which were determined in advance by our DSE, can be updated during runtime via an AXI-Lite interface from an external control unit, noted as controller in the overview figure, e.g., a CPU.

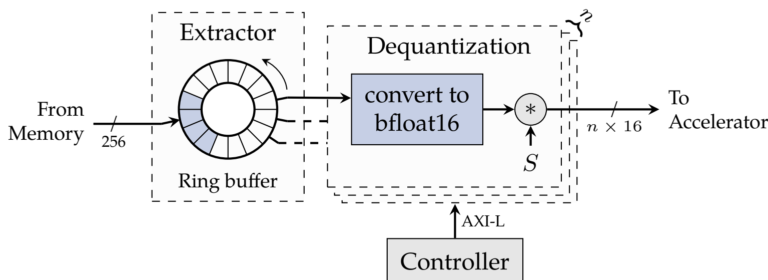


Figure 6.6: Overview of the data extractor and dequantization unit that enable conversion of an arbitrary precision input data stream into bfloat16 numbers.

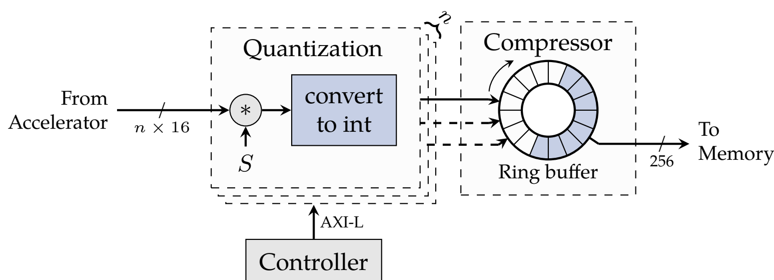


Figure 6.7: Overview of the quantization units, which convert bfloat16 numbers into arbitrary precision integer numbers, and the data compressor that packs data for transmission back to the main memory via AXI.

We will now walk you through an inference using our mixed-precision approach. For a given layer, quantized weights and inputs or intermediate results are fetched from the off-chip memory, e.g., a large DRAM memory. This data then passes through the dequantizer unit. Since data from the off-chip memory comes in large chunks of aligned data and at a nondeterministic time, incoming data is first buffered in a ring buffer inside the extractor. From this ring buffer, we can read smaller chunks of any given bit-length for dequantization. This bit-length is set to the corresponding quantization bit-width. In each cycle, multiple small chunks can be read from the ring buffer to feed multiple dequantization units simultaneously. Running the dequantization in parallel, allows us not only to mitigate backpressure effects, but also allows for clock-domain-crossing, which is necessary in reconfigurable fabrics that

usually run slower than the glue logic on-chip network. Each of the dequantization units first convert integer inputs into bfloat16 by aligning the number format and the exponent accordingly. This process follows a straightforward nine-stage approach, of which one checks the sign bit and the other eight shift the exponent. Since bfloat16 has eight exponent bits, we need exactly these nine stages, which can be realized in a pipelined structure. Then the scale factor is multiplied onto the converted number using a simple floating-point multiplier. Once the conversion has finished, we can pack multiple bfloat16 values and send them via the on-chip network to the DNN accelerator. The DNN accelerator finally receives its inputs and weight without noticing they underwent a conversion.

Pseudocode 6.1: Pseudocode representation of the quantization unit.

Data: Data from accelerator: I
Result: Data to memory: O

```
1 parallel-for i_d ∈ I do
2 |   r_float = i_d · 1/S                                   /* Apply scale factor */
  |   /* Convert bfloat16 to int                                                */
3 |   r_int = exponent_shift(r_float)                       /* adjust exponent */
4 |   if r_float < 0 then apply_twos_complement(r_int)      /* check sign bit */
5 |   place_in_ring_buffer(r_int)
6 endfor
```

When the DNN accelerator has finished parts of the computation and returns the first results, we can quantize them before transferring them to the off-chip memory. Some intermediate results and weights may be stored in a local memory that directly holds values in bfloat16 format. All remaining results, which have to be offloaded, are now passed through the second part of our architecture: the quantization part. First, the individual bfloat16 values are unpacked and the inverse scale factor is applied. Then, the conversion into the previous defined integer format is performed. This happens in multiple quantization units, which run in parallel to meet the bandwidth requirements. Their function is described by Pseudocode 6.1. Processed quantized values are then given into a compressor that packs converted values back into aligned packets for transmission to the off-chip memory. We realize this using another ring buffer, which can write multiple converted integer values of arbitrary length at the same time and give out data packets of a fixed length. This helps us to keep the data always aligned, implement correct handshaking for AXI communication, and to use all bits of the memory bus efficiently.

Obviously, our number converter units have to be able to change their parameters in between layers, which causes write accesses to registers. Therefore, the scale factor is not fixed, but stored in two registers, one that is currently used and another one, in which the controller can preload the next scale factor for the upcoming layer. This enables fast switching to new quantization and dequantization precision for all components.

Since we work on a reconfigurable fabric, there is still a more area efficient way of switching between different precision. It is conceivable to design the quantization and dequantization units partially reconfigurable. A design for each bit-width and scale factor can be synthesized and be loaded when it is needed. To enable a seamless switching between different precision levels in-between layers, we can instantiate the number converter architecture twice to operate in a ping-pong scheme. While one set of units is busy converting data, the other gets reconfigured and prepared for the next layer. Of course, fixing the scale factor to the synthesized design, enables us to highly optimize the multiplication and the extractor or compressor, as we can drop support for multiple bit-widths.

In the next section, we will explore the performance of the here described hardware architecture on a reconfigurable fabric and will see that it enables fast and flexible mixed-precision for increased energy efficiency and robustness.

6.6 Evaluation and Discussion

To prove our claim of increased robustness and energy efficiency during DNN inference with our mixed-precision approach, we performed comprehensive experiments. As workloads, we chose CNNs, which are typical for environment perception in embedded systems like semantic segmentation networks. For the evaluation, we first introduce our test setup, namely the picked CNNs and the hardware model for energy estimation. Then we looked at the energy savings mixed-precision can achieve and particularly how we can improve the robustness of the DNN. In the second part, we present our evaluation results on a modern SoC with a reconfigurable fabric to demonstrate how our method enables mixed-precision in such a platform. Finally, we discuss the results, also in comparison to other quantization approaches.

Network Backbone Dataset	DeepLabV3+ ResNet-101 CityScapes	DeepLabV3+ MobileNet-V2 CityScapes	UNet ResNet-18 DAVID	YOLOP BDD-100k
# Layers	113	60	35	102
# Parameters	58.75 million	5.23 million	19.46 million	7.94 million
Input size	2048 × 1024	2048 × 1024	608 × 800	640 × 480
Baseline accuracy	95.9 %	95.2 %	96.7 %	79.5 %
Inference energy	5520.00 mJ	4112.77 mJ	501.79 mJ	129.35 mJ
DRAM accesses	3292 million	2953 million	2570 million	1297 million
DRAM energy	658.49 mJ	590.69 mJ	51.40 mJ	27.94 mJ

Table 6.1: Workload details used for evaluation.

6.6.1 Selected Workloads and Test Setup

For our robustness and energy evaluation, we looked at four large image-segmentation CNNs. Each infers three distinct, well-established datasets, which are corrupted by different weather conditions for robustness evaluation. The goal of our evaluation is to find precision levels for each network's layer and image corruption type that maximizes the trade-off between fewer off-chip memory accesses through quantization and model accuracy. In doing that, we tried to maintain the baseline accuracy of the model. An overview of the workloads and some of their parameters are given in Table 6.1.

As evaluation datasets, we used CityScapes [72], DAVID[14] and BDD-100k [73]. For the first, a subset is used that features perturbed images with rain and fog [345]. This subset consists of 52 validation samples with three different severity levels and twelve different patterns each, which results in 1872 samples. And for the latter two, rain, fog, brightness and snow are added artificially through the image corruption tool by Michaelis et al. [346]. This tool adds those perturbations by mixing in noise that reflects the corresponding weather condition with a given intensity. Since we are using an established tool to corrupt the images, we can alter the entire CityScapes dataset

[14] https://mediatum.ub.tum.de/1596437

and build a large basis for evaluation. CityScapes is evaluated with DeepLabV3+ [96] for semantic segmentation. This network is moreover tested with two backbones, ResNet-101 and MobileNetV2. Both networks are pretrained[15] and achieve 95.9 % and 95.2 % pixel accuracy on the finely annotated CityScapes dataset, respectively. The DAVID autonomous driving dataset is fed into a U-Net topology. We trained the network from scratch to work with the DAVID dataset. It achieves a 96.7 % Intersection over Union (IoU) baseline accuracy. Finally, BDD-100k is tested with YOLOP [86] to segment drivable area, traffic objects and lanes. Here, we also used a pretrained model checkpoint[16]. Since the network returns three metrics at the same time, we used the average of all three for evaluation. The baseline accuracies for the network are 91.5 % for drivable area, 70.5 % for lane detection and 76.5 % mAP. All those datasets and models are well-established in image segmentation, which is crucial for, e.g., autonomous driving. Even though these four CNNs address the same challenge, they have distinct architectures and operate on different datasets, which allows us to take a deep look into how quantization impacts segmentation accuracy, energy efficiency and robustness. Besides them, our tool and architecture can also evaluate any PyTorch model to identify to what degree mixed-precision can increase the efficiency and robustness.

Finding an optimal quantization level for intermediate result and weights of each layer in each perturbed workload is crucial to benefit from our mixed-precision inference. As stated before, we find those levels using our tool that performs a DSE using a GA. Initially we started to shrink the design space to enable a faster exploration. For each layer's weight and activations, we limited the bit-width to the interval $x \in [2, 16]$ since more bits do not increase accuracy and fewer bits yield a too low accuracy. Moreover, we did not evaluate the entire dataset, but a significant subset of samples. For BDD-100k, CityScapes and DAVID we set the number of samples to 2048, 256 and 1024, respectively. To figure out the parameters for the GA, we performed sweeps over different GA parameters, like for the sparsity exploitation. We settled at 30 generations with a population size of 20. Thereby, we used a mutation factor and probability of 10 and 1.0, as well as crossover factor and probability of 5 and 1.0, respectively. With these parameters the GA has the objective to maximize the model accuracy and the off-chip memory compression through quantization. An accuracy constraint is added to dismiss solutions with an extremely low model accuracy. This constraint is obtained based on the baseline model accuracy, from which we allowed for a maximum of 3 %

[15] https://github.com/VainF/DeepLabV3Plus-Pytorch
[16] https://github.com/hustvl/YOLOP

accuracy degradation. To investigate the impact on robustness of mixed-precision, we performed multiple DSEs: each model and dataset combination is tested with all the above-mentioned weather conditions and intensities. In total, we evaluated over 4500 precision combinations.

Before starting to investigate the impact of quantization, we had a look into how many DRAM accesses each workload cause in its baseline configuration. As stated before, we used Timeloop to get an estimation of the actual memory transactions and how much traffic we save using our compression using mixed-precision inference. As basis for our architecture model, we chose the well-supported Eyeriss-v2 accelerator [60]. In its original configuration from 2016, however, Eyeriss offers only 12×14 processing elements with 128 kB local scratchpad memory. To match the performance of modern state-of-the-art embedded DNN accelerators, we made some changes to the architecture. First, we increased the number of PEs to 48x48, 2304 in total. The size of the local memory is set to 3 MB, inspired by architectures like Tesla's FSD. Local buffers for weights, inputs and accumulators are configured to a size of 192, 12 and 16 entries each, respectively. To estimate the energy consumption using Accelergy and Timeloop, we used a 45 nm technology for all components and the DRAM. The latter is connected with a 256-bit bus, which also reflects the AXI-Stream protocol that is used for the quantization and dequantization units in the reconfigurable fabric. With a mapping performed by Timeloop, we could derive the amount of required DRAM accesses and their energy consumption. To also account for the energy our quantization method adds, we reassembled a corresponding model in Accelergy that features the ring buffers and the quantization logic.

6.6.2 Evaluation of Mixed-Precision to Enhance Energy Efficiency and Robustness

First, we look at the energy each workload consumes for DRAM transfers when inferring a single input sample. For reference, the total energy required for an inference using bfloat16 is given in Table 6.1. We can already see that the numerous DRAM transfers can become challenging for energy constrained systems. To address this, we can now start our DSE tool to quantize the workloads and reduce the inference energy. The results for all four workloads are given in Figure 6.8. Each plot shows only the energy needed for offloading and the corresponding accuracy. Pareto optimal solutions are indicated in blue. For all three workloads, a clear trend can be observed:

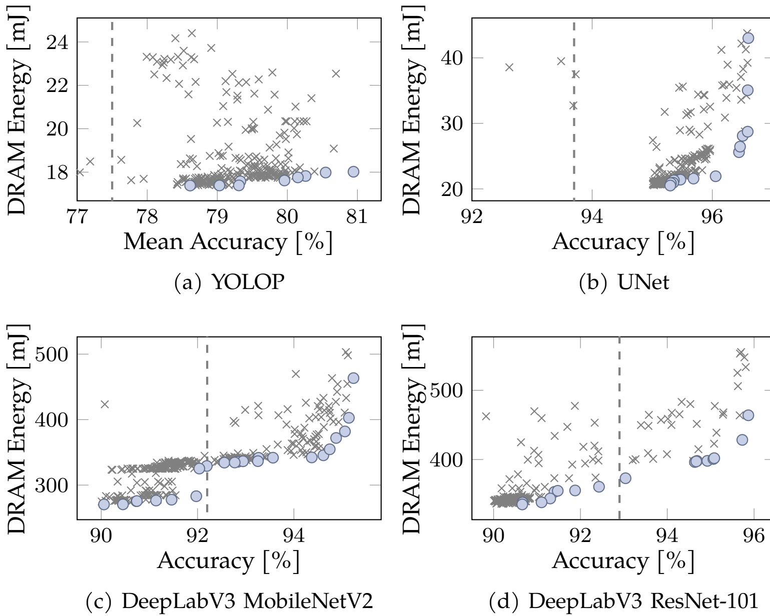


Figure 6.8: Quantization exploration for the most energy-efficient mixed-precision inference for the selected four workloads.

Obviously, a higher model accuracy also requires a more precise inference, resulting in more energy consumption due to offloading. If we look for example at Figure 6.8a, we can see that the Pareto optimal solutions can more than half the DRAM energy, from 27.94 mJ at bfloat16 execution down to 17.93 mJ using mixed-precision.

The figures above only indicate the energy needed for DRAM movements, as we only altered this part with our mixed-precision approach. Nonetheless, it is reasonable to look at the total energy consumption per inference and how we can improve it. Especially, to show if our number converter adds less energy to the total energy budget than we can save through compression. The total energy estimation reported by Timeloop and Accelergy of all Pareto optimal solutions for our four workloads with and without mixed-precision is shown in Figure 6.9. We break down the energy consumption into DRAM movements, buffers, registers, MAC units, and finally our converter unit that is added to the architecture. Our converter units only add at max 2 % energy to the overall design, which is little compared to the reduction of DRAM

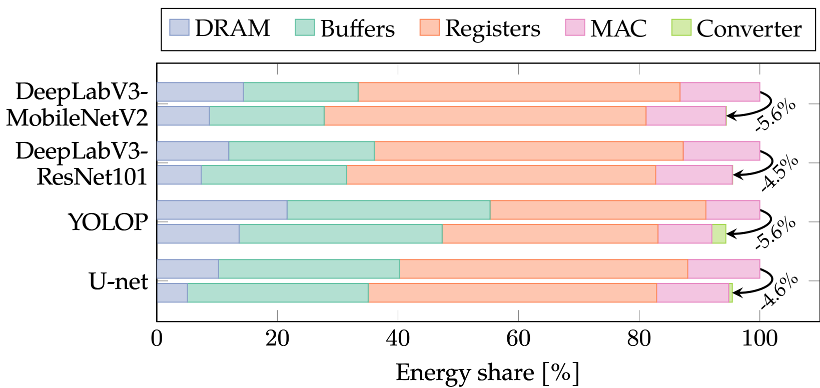


Figure 6.9: Total energy consumption of the three workloads with and without mixed-precision inference. DRAM accesses shown in blue, the DNN accelerator in teal and the conversion unit in brown.

access energy. Absolute energy figures range from about 3 mJ in the small YOLOP network to about 60 mJ in DeepLabV3 with a ResNet-101 backbone. In total, our approach could save up to 5.6 % of the total energy considering a DeepLabV3 network with a MobileNetV2 backbone or YOLOP, which operates on large input images and has many layers. Hereby, the accelerator itself ran in bfloat16 format, to achieve a high model accuracy by executing sensitive layers identified by our tool more precise, and it still allows for fine-tuning or retraining of the DNN. For DeepLabV3 with a ResNet-101 backbone and U-Net the total energy was reduced by 4.5 % and 4.6 %, respectively.

Our tool chooses the bit-width for each layer based on the impact it has on the model accuracy and the energy consumed for off-chip memory movements. The latter scales with the size of the input or output feature map of this layer, since layers whose intermediate results exceed the local memory, offloading to the DRAM happens. As a result of this evaluation metric, different layers show different degrees of quantization. To get an impression of the bit-width distribution, we show in Figure 6.10 the average precision for each layer of all Pareto optimal solutions, exemplarily for a DeepLabV3 with a MobileNetV2 backbone. Activation bits are shown in blue and weight bits in green. Error bars indicate one standard deviation. Looking at the distribution, there is no clear correlation between the precision of a layer and the dimensions or the weights for that given layer. To nevertheless identify sensitive layers, our tool

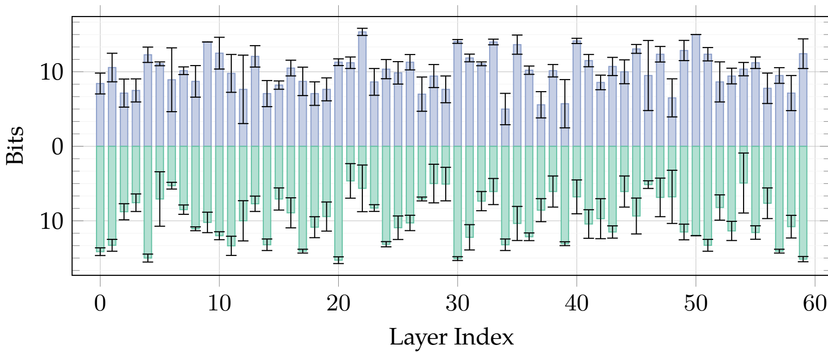
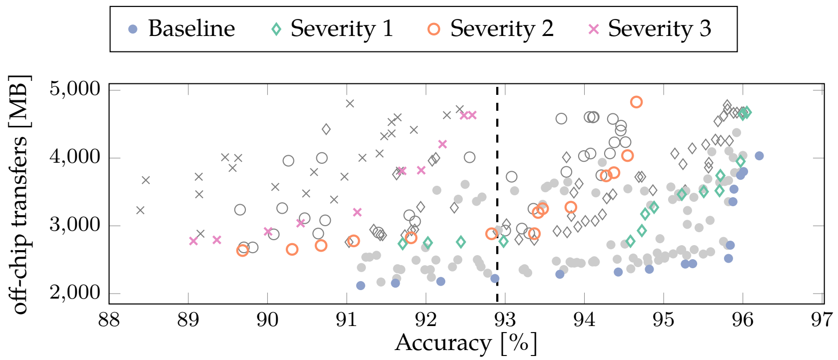


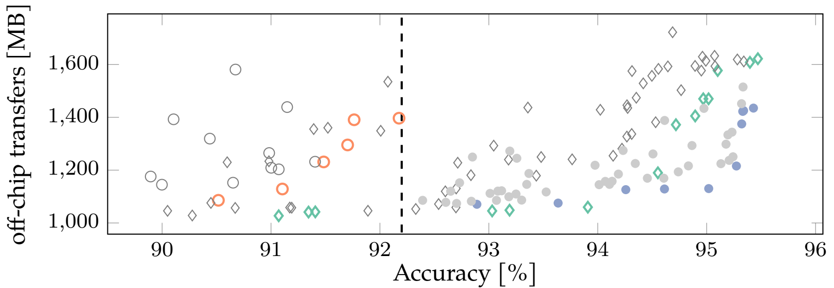
Figure 6.10: Bit-width distribution for Pareto optimal solutions in DeepLabV3 with a MobileNetV2 backbone. Blue shows activations, weights shown in green.

evaluates each bit-width combination with the validation dataset, since the minimum feasible precision regarding the model accuracy depends strongly on the model input. As it can be seen, some layers can benefit from a higher compression through a lower precision, while quantization sensitive layers might still be executed with higher precision. On average, we can quantize feature maps to 10.107 bit and weights to 9.837 bit.

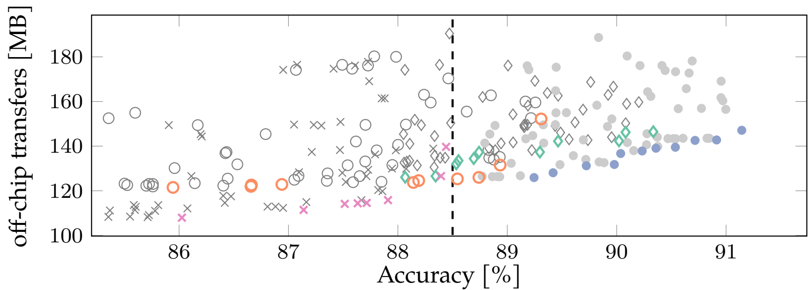
Now that we looked into the bare energy savings our mixed-precision approach brings and that it clearly covers the energy added by our number converters, we can analyze how it can support the robustness of CNN inference. Therefore, we performed, as mentioned before, multiple DSE with different perturbations on our three workloads. The results of our DSEs are depicted in Figure 6.11. Corresponding Pareto-optimal solutions are marked in color, while dominated solutions are shown in gray. Each plot shows three intensities as well as an exploration of the baseline dataset without perturbation. The tolerable minimum accuracy is indicated in the plot using a dashed line. Instead of giving the DRAM energy, we present the amount of data that has to be written and read from the off-chip memory, which directly correlates with the energy consumption. This number is independent of the technology node, as the 45 nm technology library that is shipped with Accelergy and CACTI is not state of the art anymore. In total, we evaluated various weather conditions with different intensities for different networks. Hence, we were able to only show a selection, which, however, presents the most significant findings and covers a large variety.



(a) DeepLabV3+ with ResNet101 backbone (Baseline accuracy 95.9 %, off-chip transfers without compression: 6279.8 MB), which is subject to corruption by rain.



(b) DeepLabV3+ with a MobileNetV2 backbone (Baseline accuracy 95.2 %, off-chip transfers without compression: 2068.9 MB). Data is subject to corruption by fog.



(c) YOLOP (Baseline accuracy 90.5 %, off-chip transfers without compression: 266.5 MB) with BDD-100k under the influence of brightness.

Figure 6.11: Model accuracy regarding off-chip memory movements required for a single inference of three different CNNs under various weather conditions. A dashed line shows a lower accuracy constraint.

Figure 6.11a shows the exploration results for a DeepLabV3+ model with a ResNet-101 backbone that is impacted by rainy conditions. The figure gives the model accuracy regarding the amount of off-chip memory transactions, with three severity levels of rain. Moreover, it presents the model exploration without any perturbation as baseline, marked by blue dots (●). DeepLabV3+ with a ResNet-101 backbone running with entirely bfloat16 number format creates in total 6279.88 MB of data offloading per inference. This alone accounts for 658.49 mJ energy for off-chip transactions per inference. As stated before, we can significantly reduce the number of off-chip memory accesses through mixed-precision. While maintaining baseline accuracy, we could lower them by 3760.57 MB, which equals to 59.8 % less compared to bfloat16 inference. If we now increase the perturbation by rain, we can leverage a higher computational precision and restore the model accuracy. The first rain severity level is marked by green diamonds (◊). When we select a solution of the DSE that can maintain the baseline accuracy, we could reduce the precision compared to fully bfloat16 inference. Here we saved, e.g., 1632.2 MB or 26 % of data moved to and fetched from the off-chip memory. If we allow for 1 % drop at this rain severity, we could even save up to 3006.8 MB or 47.9 %. In case of heavier rain, which is shown by orange circles (○), our approach could not completely restore the initial accuracy. However, we still stayed in the bounds of the configured accuracy constraint. For example, we could quantize the network to reach an accuracy of 94.66 % and still decrease the number of DRAM movements by 1452.58 MB or 23.1 %. When we look at very heavy rain, which is depicted with purple cross marks (×), we see that we were not able to meet the accuracy constraint regardless of how precise we perform the inference. As reference, we also tested this weather condition using 32-bit floating-point precision. Here, it only yielded 92.6 % pixel accuracy. Our approach could also achieve this accuracy, but with 1645.29 MB or 26.1 % less off-chip memory transfers.

The next interesting workload we looked at is DeepLabV3+ with a MobileNetV2 backbone under foggy conditions. Corresponding DSE results are shown in Figure 6.11b. Considering bfloat16 inference, this network loads and retrieves 2068.9 MB of data per inference from the off-chip memory. From this we could save 1215.75 MB or 41.2 % using mixed-precision when the weather conditions are clear. When we encounter light fog at severity level 1, our tool can leverage mixed-precision inference and reduced the number of off-chip memory transfers by 492.4 MB or 23.8 % compared to bfloat16 inference. At the same time, we maintained baseline model accuracy. With very severe fog at level 2 and 3, the model itself could not maintain the baseline accuracy. Here, we looked again at the performance the model can achieve under

this condition. With fog of level 2, it reached 91.9 % pixel accuracy. Targeting the same accuracy, with our mixed-precision, we could reduce off-chip data movement by 32.4 % (672.2 MB).

The third figure, Figure 6.11c, shows YOLOP, a comparably lightweight model, for drivable area detection, corrupted by increased brightness. Running at bfloat16 precision, the model requires 266.47 MB worth of off-chip memory transfers. With mixed-precision inference, we reduced this figure to 142.77 MB, which equals to a 46 % cut. At severity level 1, our tool found a solution that achieves a model accuracy of 90.3 %, which is only slightly less than an uncorrupted input. This solution also needed only 146.45 MB off-chip movements per inference, 45 % less compared to full bfloat16 inference. From level 2 the model accuracy started to degrade. Here we could reach 89.3 % model accuracy with 152.19 MB of data transfers, 42.9 % less. Finally, level 3 did not yield any solution that meet the accuracy constraint. For the sake of completeness, the best solution, that we were able to find, reaches 88.4 % accuracy, with 139.66 MB off-chip memory movements (47.5 % reduction).

6.6.3 Hardware Considerations for Mixed-Precision Inference

Now that we explored the design space of accuracy, robustness and reduced off-chip memory transfers, we want to evaluate hardware aspects of our mixed-precision inference. Therefore, we looked at various performance figures of our hardware architecture, which enables mixed-precision CNN inference on modern heterogeneous SoCs. As target platform, we chose AMD's Versal [131] platform, since it is a very recent high-performance SoC. It features besides other components like CPUs, also a reconfigurable fabric and a DNN accelerator. In particular, we mapped our architecture onto an AMD VCK190 evaluation kit, which is populated with an AMD Versal XCVC1902 device, to evaluate the performance and feasibility of our approach. On the SoC, the AI-Engines function as DNN accelerator and the quantization and dequantization units are hosted on the FPGA part, which is controlled via AXI-Lite from the ARM CPUs. As communication network, the built-in Network-on-Chip (NoC) is used. The connection of the quantization and dequantization units is done with Direct Memory Access (DMA) modules that interface with AXI-Stream.

Table 6.2 shows a breakdown of the required resources to host our architecture on the reconfigurable fabric. Our evaluation design had 16 quantization and 16 dequantization units, together with one extractor and compressor. The design ran

Component	LUTs	FFs	DSPs	BRAMs
Entire Design	37,366	25,995	32	9
AXI-DMA & Interfaces	8,007	12,754	0	9
Number converter	29,359	13,241	32	0
Extractor	4,835	2,352	0	0
1x Quantizer	686	257	1	0
1x Dequantizer	556	212	1	0
Compressor	2,138	3,120	0	0

Table 6.2: Resource utilization of our hardware prototype setup. The entire design features 16 quantization and dequantization units, with a 256-bit connection to the off-chip memory.

at 300 MHz clock speed. With this design we achieved a theoretical throughput of 4.8 billion conversions per second. From the table, we can see that our number converter consumes less than 1 % of the total available resources in the XCVC1902 device. The entire design, which comprises all necessary interfaces and peripherals for data movement, utilized just under 1 % of the total resources. This allows us to increase the number of quantization and dequantization units for more demanding designs and applications, while also keeping spare resources for other designs that need the reconfigurable fabric as well.

From a performance perspective, it is important to match the throughput of the DNN accelerator and to be able to compute modern CNN workloads within a short time window. As stated before, we designed our quantization and dequantization units in a highly parallelized and pipelined fashion. Based on a design with the aforementioned 16 units, we can report a latency of two and ten cycles for the extractor and quantization unit, respectively. Similarly, a dequantization unit and the compressor have a latency of ten and two cycles. We evaluated the latency for different integer precision. Like for the exploration, we fixed our architecture to support conversions from and to 2- to 16-bit integers. For all configurations the latency stays constant. As a result, we report a total latency of 12 clock cycles per conversion. When considering a DNN workload with 50 layers, this merely adds 2 µs at 300 MHz.

To put this into perspective, we can look at one YOLOP with 101 layers. Here, Timeloop reports 139 million memory accesses. The design with 16 conversion units for each direction was kept the same. Now, we have to consider the latency for adjusting the

configuration of the number converters between layers for weights and intermediate results. We measured this to take a total of 25 cycles, 12 to each flush and refill the pipeline plus an extra cycle for switching the register that holds the scale factor. This setup can convert the data required for 34.35 inferences on BDD-100k per second. For the larger DeepLabV3+ network with a ResNet-101 backbone, which requires 3.29 billion off-chip memory accesses, we can report a performance of 1.45 inferences per second. To achieve a similar inference rate for this workload, we can simply invest more resources and add more conversion units, which are running in parallel, since the design utilizes only a few resources. However, it has to be kept in mind that for such a large workload, we most likely have to scale up the DNN accelerator as well as the interface to the off-chip memory. The currently available bandwidth of 9.6 GB s^{-1} is probability not sufficient for DeepLabV3 inference with 30 FPS.

Finally, we have to look at the energy our number converter adds to the design. Regarding energy consumption, Xilinx Vivado reported 75 mW for our test setup with 16 converters running at 300 MHz. This equals 2.18 mJ of energy per YOLOP inference and 51.72 mJ for DeepLabV3+ with a ResNet-101 backbone, respectively.

6.6.4 Discussion

During our evaluation, we showed that an in-depth evaluation of the workload in advance is important for multiple reasons. First, it is crucial to identify sweet-spots between higher model accuracy and fewer off-chip memory accesses, since some layers contribute largely to the off-chip memory transactions and others may be very sensitive to quantization. Therefore, a designer has to carefully decide what degree of accuracy degradation is acceptable for the use case. In addition, they have to consider the area and energy our quantization extension adds to the overall architecture. For some applications, the extra hardware resources may not worth investing. Second, it is important to figure out the throughput and latency requirements at the beginning, to tune the converter architecture so that it matches the performance requirements. Finally, to increase the robustness of the CNN towards perturbed inputs, it is crucial to understand the workload, as selection of the right quantization degree for each layer and perturbation plays a significant role.

In terms of robustness, we can to restore the initial accuracy in most scenarios. However, the CNN model itself has not been optimized for weather conditions. This explains, why we were not able to find a solution that can mitigate very heavy rain

(severity level 3), even not with a higher computational precision. A combination of our approach with a network that is particularly designed and trained for robust inference, seems promising to work in even more situations. Moreover, it has to be noted, that we applied mixed-precision only as method to circumvent failures according to the intended functionality. We have to start from the premise that adversarial attacks or inputs outside the training dataset, which are essentially inputs the model is not trained to detect, can neither be detected nor mitigated by simply increasing the precision. For that further research in this direction has to be carried out. Hence, they are not covered in this contribution.

Considering our quantization architecture, we generated energy figures using Timeloop and a comparably large technology node. Hence, a lower total energy might be achieved if moving to more recent nodes. This is already indicated by our evaluation on the recent AMD Versal platform that is implemented in 7 nm.

6.7 Conclusion and Outlook

Applications like autonomous driving or robotics often rely on CNNs for environment perception. However, the embedded domain poses multiple challenges to the DNN accelerator that executes the CNN workload. For one thing there are harsh performance and energy constrains, since they are typically battery powered, then there are needs to make the network robust to perturbations like changes in weather conditions that obstruct the input. In the previous chapter, we already surveyed pruning and sparsification methods to increase the performance and energy efficiency of DNN inference. However, robustness was not yet discussed.

In this chapter, we presented a mixed-precision inference method, which is realized by an architecture that adds support for increased robustness to modern heterogeneous state-of-the-art SoCs. We have shown that mixed-precision inference cannot only increase the energy efficiency of CNN inference but can also support the model's robustness towards various weather conditions, while still saving energy through targeted quantization of less quantization sensitive layers. In contrast to traditional network quantization, mixed-precision achieves a higher model accuracy as quantization sensitive layers can be run with higher precision, and less sensitive layers are executed with less precision. When the inputs are subject to perturbation, we can increase the computational precision through mixed-precision inference and restore the initial accuracy. Therefore, we proposed a hardware architecture with low

resource requirements and latency, which takes care of dynamic number conversion of data going from the off-chip memory to the accelerator and back. Our design allows using an off-the-shelf DNN accelerator that uses bfloat16 number format and thus supports continual online learning for embedded High-Performance Computing (HPC) applications.

To test and evaluate our mixed-precision approach, we designed a tool to determine the best trade-off between quantization and model accuracy. Moreover, we mapped the hardware architecture on an AMD Versal platform that represents a modern HPC platform. With this setup we performed a wide range of experiments and DSEs. Thereby, we gained insightful results on how the model accuracy behaves under various weather conditions and degrees of quantization. We demonstrated, for example, that under rainy conditions we can maintain the model accuracy compared to clear weather conditions, by increasing the precision. At the same time, we could reduce the number of off-chip memory transfers by 45 %, which also directly reflects into saved inference energy.

The state of the art around quantization is steadily growing and here research has by far not finished. Hence, some paths for future research directions are worth exploring. For example, evaluating models that are especially trained to be robust to perturbations seems like a promising approach, e.g., to increase the accuracy even under heavy weather conditions. Also, a combination of our approach with the deraining methods can deliver improved results. To increase the efficiency in terms of energy consumption further, it is also conceivable to use a finer quantization granularity than per-layer. With such an approach a higher compression can be achieved and robustness sensitive sections of layers can run with a high precision. Although a similar method was investigated by Song et al. [219], one has to carefully consider the time it takes to change the bit-width. When this has to happen too often, the returns may quickly diminish. In addition, one should consider looking into other quantization schemes that leverage a lower quantization noise like non-uniform quantization. Another interesting approach may be to combine this quantization method with pruning methods like the one we introduced in the previous chapter. The foundations for this are already laid in the tool and a straightforward combination of the hardware structures is conceivable.

Chapter 7

Summary, Conclusion and Outlook

Deep Neural Networks (DNNs) have become ubiquitous in many domains starting from consumer products like smartphones to highly constrained domains like automotive. Over the course of this thesis, we have seen that there are multiple challenges that are yet searching for a solution. Especially, when we want to deploy Deep Neural Network (DNN) algorithms to embedded systems. One of the most predominant challenges is the rising complexity of the DNN workloads, which is necessary to solve more and more complex tasks. At the same time, constraints like an energy consumption or chip area limits, make it hard to put more powerful devices into those embedded systems. Over the course of the last ten years, many efforts have been put into making DNN inference more efficient in terms of energy, area, performance and latency. Starting with the large rise of dedicated hardware accelerators that are tailored to the unique dataflow of DNNs. Then multiple optimization strategies were explored by a vast body of research: From dedicated hardware-aware Neural Architecture Search (NAS) that aims to co-design the network topology in accordance with the hardware accelerator, over pruning and quantization that effectively reduces or simplifies the vast number of computations in DNNs, to tools that enable highly-efficient mapping and tiling of the complex workloads onto the accelerators.

7.1 Concluding Remarks

In this thesis, we explored ways to address the energy efficiency constraints to performance requirements gap that in particular emerges when Convolutional Neural Networks (CNNs) are deployed in embedded systems. Especially, CNNs are essential to vision based environment perception systems and at the same time have huge

memory requirements for the intermediate results. Since those intermediate results of large DNN workloads cannot be stored locally, they have to be offloaded to external memories. This necessary process does not only add latency to the inference but also requires a large chunk of the available energy budget. As a result, CNNs with their large feature maps present an exceptional challenge regarding energy efficiency.

To tackle the energy demand of CNNs in embedded systems we propose three main contributions in this thesis. First, in Chapter 4 looked at the interplay of DNN model and hardware accelerator that is later executing the workload. Therefore, we presented FLECSim a cycle-accurate simulation framework to model and evaluate all kinds of dataflows that may occur in DNNs. FLECSim brings along necessary infrastructure like Direct Memory Accesses (DMAs), memories and Central Processing Units (CPUs) to quickly sketch a System-on-Chip (SoC) design for fast evaluation. In combination with established tools like Accelergy and CACTI, FLECSim is able to assess the efficiency and performance of workload running on the specified SoC. Using our cycle-accurate simulator, we carried out experiments that gave viable design insights. Moreover, we conducted a case-study in which, we used FLECSim to design a DNN accelerator that is hosted on an embedded Field Programmable Gate Array (eFPGA) for the European Processor Initiative (EPI) project. This design is area and energy constrained and has many degrees of freedom, as the eFPGA architecture, i.e., the number and arrangement of building blocks, can be adjusted. The found eFPGA accelerator can infer a pruned and quantized SqueezeNet topology for face recognition in about 150 ms, while using a mere 4.8 mW power.

Cycle-accurate simulation has the advantage of being precise, but it typically takes a lot of time to simulate every single data movement and operations. To address this, we moreover introduced our analytical approach that does not rely on this kind of detailed simulation, but uses a model to estimate the total number of memory access and computations directly. It uses the widely established Roofline model, which essentially groups computational workloads in memory or compute bound problems. Our model, now analyzes the accelerator and DNN workload parameters to determine whether they can be considered as compute or memory bound. Since the tool is much faster than cycle-accurate simulation, we can even use it to explore various accelerator designs and to find an ideal configuration for a given DNN workload. To show the performance of our model we evaluated it with some benchmark workloads and performed a case study showing its feasibility in Design Space Exploration (DSE). In our experiments, we demonstrated a precision that is only 1 % off compared to a cycle-accurate simulation and at the same time up to 100× faster comparing to state-

of-the-art works. The full power of our two accelerator and algorithm co-design tools unfolds when they are combined. With the analytical model, we can coarsely search the design space and then evaluate the remaining design options using cycle-accurate simulation in FLECSim.

In addition, to the co-design approaches that enable efficient design of accelerators and CNN workloads, we also looked in particular into the energy expensive off-chip memory transactions. In Chapter 5 we presented Spex and Sparse-Blox, our methodology to increase and prune regular activation sparsity in DNNs. Regular sparsity in the form of blocks that match the underlying hardware accelerator, was proven beneficial to achieve a straightforward reduction in Multiply-Accumulate (MAC) operations and compression of off-chip memory transactions. In hardware, this pruning mechanism is realized by our low-overhead hardware extension Sparse-Blox, which detects sparse blocks by comparing the sum of a block with a threshold. The threshold upon which a block in a given layer is pruned, is determined in advance by our exploration tool Spex. It systematically searches the design space of different per-layer thresholds and the corresponding inference accuracy, as higher thresholds and thus more pruning can worsen the model accuracy. With Spex and Sparse-Blox we analyzed their performance effects on the state-of-the-art benchmark CNNs ResNet-50 and YOLOv5s. We could demonstrate that we are able to prune up to 22.73 % of activations considering a block size of 8×8. At the same time, we reduced the number of off-chip memory transactions by up to 18.9 %. Thereby, our hardware extension consumes 2.5× to 5× less area than state-of-the-art sparse accelerators that operate on irregular sparsity.

Apart from pruning, quantization is also a very popular optimization technique to make DNN inference more efficient. In Chapter 6 we introduced a novel mixed-precision dataflow that is not only able to compress off-chip memory transactions but can also help to increase the robustness of CNNs to perturbed inputs. Therefore, we proposed a hardware module that is put in between the DNN accelerator and the off-chip memory to enable dynamic quantization and dequantization of weights and activations. As a result, the accelerator can operate on bfloat16 number format to keep the possibility of online learning, and we are still able to benefit from compression through quantization. To find beneficial degrees of quantization for each layer's weights and activations we presented an exploration tool to identify quantization sensitive layers and those that can benefit from higher quantization. The tool can also analyze to what extent higher inference precision can restore the accuracy of networks that are under influence of image corruptions, like various weather conditions or

other environmental factors. Evaluation of three highly-demanding CNNs for image segmentation showed that we could achieve a large reduction in off-chip memory transfers through mixed-percision. Even when the inputs are subject to, e.g., rainy conditions we were able to restore the initial precision without rain using a higher precision. In such a weather perturbation scenario, we could, in particular, save up to 45 % off-chip memory transfers and thus energy. Our quantization module, hosted on the reconfigurable fabric, of a modern SoC thereby only required a small energy and area budget.

7.2 Outlook and Future Work

Obviously, the landscape of optimizations that can be applied to make DNNs more efficient is vast. As always, a designer has to decide between flexibility to support various different dataflows versus the efficiency one can achieve by specifically designing an accelerator for one particular DNN workload. This problem is well-known from hardware / software co-design and will likely never be resolved entirely. However, we saw in this thesis that there are many ways to unify both at least to some extent. Therefore, we introduced co-design approaches along with optimization strategies that leverage the probabilistic behavior of DNNs. Although our proposed methods demonstrated good results to make DNNs more efficient, there is still loads of room for further extension of our work.

Starting with our co-design methods. As mentioned before, a combination of both the cycle-accurate and the analytical tool seems very beneficial for large-scale parameter exploration. The potential of this has not been fully studied yet. Especially considering the currently popular Transformer models, which have yet other requirements than CNNs. Besides that, it is conceivable to run an actual HW-NAS using, e.g., our analytical model to co-design accelerator and model at the same time. Here some preliminary work has already shown promising results, however, they are often rendered infeasible due to the long exploration time. Finally, an evaluation on an actual hardware platform, involving a synthesis tool for Application-Specific Integrated Circuit (ASIC) flows, is worth looking into. This will help to verify and underpin the Performance, Power and Area (PPA) figures reported by our simulation tools.

Looking at the optimization strategies, we investigated with Sparse-Blox and our mixed-precision methodology, some further research options are imaginable. First, it is worth looking at DNNs aside from CNNs. For example, sparsification may also

support Long-Short Term Memories (LSTMs) or quantization can increase the robustness of Transformer networks. Beyond simulation and evaluation using mapping tools like Timeloop or Accelergy, it is insightful to perform the same experiments with ASIC synthesis tools to obtain more real-world accurate PPA results. We can achieve this, e.g., with a modification of the systolic array builder Gemmini, since it already features a configurable systolic array and an ASIC flow. One promising approach is moreover the consideration of other optimization algorithms for both determining the precision and the sparsity thresholds. Recently, super networks demonstrated great capabilities to reduce the design space for NAS. Besides the methods mentioned so far, integration of our methods into the training process can also be envisaged. Most likely training towards networks that already yield high degrees of sparsity and are inherently robust, might even increase the benefits of our methodologies.

Another opportunity for further research is the combination of our sparsification and quantization method. This will likely increase the performance of the DNN accelerator further. The advantage of our approaches comes also here to play: the accelerator architecture itself is left unchanged, and only Sparse-Blox and the external number converter have to be added. Finally, it is also conceivable to use our architecture simulation and modelling tools. With this, we can lay out an efficient accelerator that works with quantized and pruned networks, before adding Sparse-Blox and the number converter. Therefore, sparsification and mixed-precision has to be added to FLECSim or our analytical model.

Bibliography

[1] Olga **Russakovsky** et al. *ImageNet Large Scale Visual Recognition Challenge*. en. arXiv:1409.0575 [cs]. Jan. 2015. http://arxiv.org/abs/1409.0575

[2] Xiaohua **Zhai**, Alexander **Kolesnikov**, Neil **Houlsby**, and Lucas **Beyer**. *Scaling Vision Transformers*. arXiv:2106.04560 [cs]. June 2022. doi:10.48550/arXiv.2106.04560. http://arxiv.org/abs/2106.04560

[3] Christian **Szegedy**, Wei **Liu**, Yangqing **Jia**, Pierre **Sermanet**, Scott **Reed**, Dragomir **Anguelov**, Dumitru **Erhan**, Vincent **Vanhoucke**, and Andrew **Rabinovich**. "Going Deeper with Convolutions." en. In: *arXiv:1409.4842 [cs]*, Sept. 2014. arXiv: 1409.4842. http://arxiv.org/abs/1409.4842

[4] Radosvet **Desislavov**, Fernando **Martínez-Plumed**, and José **Hernández-Orallo**. *Compute and Energy Consumption Trends in Deep Learning Inference*. en. arXiv:2109.05472 [cs]. Sept. 2021. http://arxiv.org/abs/2109.05472

[5] Jaime **Sevilla**, Lennart **Heim**, Anson **Ho**, Tamay **Besiroglu**, Marius **Hobbhahn**, and Pablo **Villalobos**. "Compute Trends Across Three Eras of Machine Learning." en. In: *2022 International Joint Conference on Neural Networks (IJCNN)*. arXiv:2202.05924 [cs]. July 2022, pp. 1–8. doi:10.1109/IJCNN55064.2022.9891914. http://arxiv.org/abs/2202.05924

[6] Victor **Hristov**. *A16 Bionic explained: what's new in Apple's Pro-grade mobile chip?* en. Sept. 2022. https://www.phonearena.com/news/A16-Bionic-explained-whats-new_id142438

[7] Norman P. **Jouppi** et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit." en. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Toronto ON Canada: ACM, June 2017, pp. 1–12. isbn:978-1-4503-4892-8. doi:10.1145/3079856.3080246. https://dl.acm.org/doi/10.1145/3079856.3080246

[8] Frank **Rosenblatt**. "Principles of Neurodynamics. Perceptrons and the Theory of Brain Mechanisms." In: *Archives of General Psychiatry* 7.3, Mar. 1961, pp. 218–219. ISSN: 0003-990X. doi:10.1001/archpsyc.1962.01720030064010. https://doi.org/10.1001/archpsyc.1962.01720030064010

[9] A L **Samuel**. "Some Studies in Machine Learning Using the Game of Checkers." en. In: 1959

[10] David E. **Rumelhart**, Geoffrey E. **Hinton**, and Ronald J. **Williams**. "Learning representations by back-propagating errors." en. In: *Nature* 323.6088, Oct. 1986. Number: 6088 Publisher: Nature Publishing Group, pp. 533–536. ISSN: 1476-4687. doi:10.1038/323533a0. https://www.nature.com/articles/323533a0

[11] D. **Crevier**. *Ai: The Tumultuous History Of The Search For Artificial Intelligence*. Basic Books, 1993. isbn:978-0-465-02997-6. https://books.google.de/books?id=QJNQAAAAMAAJ

[12] Yann **LeCun**, Yoshua **Bengio**, and Geoffrey **Hinton**. "Deep learning." en. In: *Nature* 521.7553, May 2015. Number: 7553 Publisher: Nature Publishing Group, pp. 436–444. ISSN: 1476-4687. doi:10.1038/nature14539. https://www.nature.com/articles/nature14539

[13] Yunzhu **Li**, Andre **Esteva**, Brett **Kuprel**, Rob **Novoa**, Justin **Ko**, and Sebastian **Thrun**. "Skin Cancer Detection and Tracking using Data Synthesis and Deep Learning." In: *arXiv:1612.01074 [cs]*, Dec. 2016. arXiv: 1612.01074. http://arxiv.org/abs/1612.01074

[14] S. **Baehr**, F. **Kempf**, and J. **Becker**. "Data Reduction and Readout Triggering in Particle Physics Experiments Using Neural Networks on FPGAs." In: *2018 IEEE 18th International Conference on Nanotechnology (IEEE-NANO)*. ISSN: 1944-9380. July 2018, pp. 1–4. doi:10.1109/NANO.2018.8626239

[15] M. I. **Jordan** and T. M. **Mitchell**. "Machine learning: Trends, perspectives, and prospects." In: *Science* 349.6245, July 2015. Publisher: American Association for the Advancement of Science, pp. 255–260. doi:10.1126/science.aaa8415. https://www.science.org/doi/10.1126/science.aaa8415

[16] S.J. **Russell**, P. **Norvig**, and J.F. **Canny**. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall/Pearson Education, 2003. isbn:978-0-13-790395-5. https://books.google.de/books?id=KI2WQgAACAAJ

[17] S. **Lloyd**. "Least squares quantization in PCM." in: *IEEE Transactions on Information Theory* 28.2, Mar. 1982. Conference Name: IEEE Transactions on Information Theory, pp. 129–137. issn: 1557-9654. doi:10.1109/TIT.1982.1056489

[18] Martin **Ester**, Hans-Peter **Kriegel**, Jörg **Sander**, and Xiaowei **Xu**. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, Aug. 1996, pp. 226–231

[19] Ian T. **Jolliffe** and Jorge **Cadima**. "Principal component analysis: a review and recent developments." In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374.2065, Apr. 2016. Publisher: Royal Society, p. 20150202. doi:10.1098/rsta.2015.0202. https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0202

[20] Sindhu **Padakandla**. "A Survey of Reinforcement Learning Algorithms for Dynamically Varying Environments." In: *ACM Computing Surveys* 54.6, July 2021. arXiv:2005.10619 [cs, stat], pp. 1–25. issn: 0360-0300, 1557-7341. doi:10.1145/3459991. http://arxiv.org/abs/2005.10619

[21] Tushar **Krishna**, Hyoukjun **Kwon**, Angshuman **Parashar**, Michael **Pellauer**, and Ananda **Samajdar**. "Data Orchestration in Deep Learning Accelerators." en. In: *Synthesis Lectures on Computer Architecture* 15.3, Aug. 2020, pp. 1–164. issn: 1935-3235, 1935-3243. doi:10.2200/S01015ED1V01Y202005CAC052. https://www.morganclaypool.com/doi/10.2200/S01015ED1V01Y202005CAC052

[22] Warren S. **McCulloch** and Walter **Pitts**. "A logical calculus of the ideas immanent in nervous activity." en. In: *The bulletin of mathematical biophysics* 5.4, Dec. 1943, pp. 115–133. issn: 1522-9602. doi:10.1007/BF02478259. https://doi.org/10.1007/BF02478259

[23] Ian **Goodfellow**, Yoshua **Bengio**, and Aaron **Courville**. *Deep Learning*. MIT Press, 2016

[24] José **Mira**, Francisco **Sandoval**, Gerhard **Goos**, Juris **Hartmanis**, and Jan van **Leeuwen**, eds. *From Natural to Artificial Neural Computation: International Workshop on Artificial Neural Networks Malaga-Torremolinos, Spain, June 7–9, 1995 Proceedings*. en. Vol. 930. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1995. isbn:978-3-540-59497-0. doi:10.1007/3-540-59497-3. http://link.springer.com/10.1007/3-540-59497-3

[25] Kunihiko **Fukushima**. "Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements." In: *IEEE Transactions on Systems Science and Cybernetics* 5.4, Oct. 1969. Conference Name: IEEE Transactions on Systems Science and Cybernetics, pp. 322–333. issn: 2168-2887. doi:10.1109/TSSC.1969.300225

[26] Prajit **Ramachandran**, Barret **Zoph**, and Quoc V. **Le**. *Searching for Activation Functions*. arXiv:1710.05941 [cs]. Oct. 2017. doi:10.48550/arXiv.1710.05941. http://arxiv.org/abs/1710.05941

[27] Xavier **Glorot**, Antoine **Bordes**, and Y. **Bengio**. "Deep Sparse Rectifier Neural Networks." In: vol. 15. Jan. 2010

[28] Vivienne **Sze**, Yu-Hsin **Chen**, Tien-Ju **Yang**, and Joel S. **Emer**. *Efficient Processing of Deep Neural Networks*. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2020. isbn:978-3-031-00638-8978-3-031-01766-7. doi:10.1007/978-3-031-01766-7. https://link.springer.com/10.1007/978-3-031-01766-7

[29] Y. **LeCun**, B. **Boser**, J. S. **Denker**, D. **Henderson**, R. E. **Howard**, W. **Hubbard**, and L. D. **Jackel**. "Backpropagation Applied to Handwritten Zip Code Recognition." In: *Neural Computation* 1.4, Dec. 1989. Conference Name: Neural Computation, pp. 541–551. issn: 0899-7667. doi:10.1162/neco.1989.1.4.541

[30] Kyoung-Su **Oh** and Keechul **Jung**. "GPU implementation of neural networks." en. In: *Pattern Recognition* 37.6, June 2004, pp. 1311–1314. issn: 0031-3203. doi:10.1016/j.patcog.2004.01.013. https://www.sciencedirect.com/science/article/pii/S0031320304000524

[31] Diederik P. **Kingma** and Jimmy **Ba**. *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980 [cs]. Jan. 2017. doi:10.48550/arXiv.1412.6980. http://arxiv.org/abs/1412.6980

[32] Ilya **Sutskever**, James **Martens**, George **Dahl**, and Geoffrey **Hinton**. "On the importance of initialization and momentum in deep learning." In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML'13. Atlanta, GA, USA: JMLR.org, June 2013, pp. III–1139–III–1147

[33] Leslie N. **Smith**. *Cyclical Learning Rates for Training Neural Networks*. arXiv:1506.01186 [cs]. Apr. 2017. doi:10.48550/arXiv.1506.01186. http://arxiv.org/abs/1506.01186

[34] Yuan **Yao**, Lorenzo **Rosasco**, and Andrea **Caponnetto**. "On Early Stopping in Gradient Descent Learning." en. In: *Constructive Approximation* 26.2, Aug. 2007, pp. 289–315. ISSN: 1432-0940. doi:10.1007/s00365-006-0663-2. https://doi.org/10.1007/s00365-006-0663-2

[35] Sergey **Ioffe** and Christian **Szegedy**. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv:1502.03167 [cs]. Mar. 2015. doi:10.48550/arXiv.1502.03167. http://arxiv.org/abs/1502.03167

[36] Kunihiko **Fukushima**. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position." en. In: *Biological Cybernetics* 36.4, Apr. 1980, pp. 193–202. ISSN: 1432-0770. doi:10.1007/BF00344251. https://doi.org/10.1007/BF00344251

[37] Y. **Lecun**, L. **Bottou**, Y. **Bengio**, and P. **Haffner**. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11, Nov. 1998. Conference Name: Proceedings of the IEEE, pp. 2278–2324. ISSN: 1558-2256. doi:10.1109/5.726791

[38] Kumar **Chellapilla**, Sidd **Puri**, and Patrice **Simard**. "High Performance Convolutional Neural Networks for Document Processing." en. In: Suvisoft, Oct. 2006. https://hal.inria.fr/inria-00112631

[39] Alex **Krizhevsky**, Ilya **Sutskever**, and Geoffrey E **Hinton**. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems*. Vol. 25. Curran Associates, Inc., 2012. https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[40] Kaiming **He**, Xiangyu **Zhang**, Shaoqing **Ren**, and Jian **Sun**. "Deep Residual Learning for Image Recognition." en. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, June 2016, pp. 770–778. isbn:978-1-4673-8851-1. doi:10.1109/CVPR.2016.90. http://ieeexplore.ieee.org/document/7780459/

[41] Forrest N. **Iandola**, Song **Han**, Matthew W. **Moskewicz**, Khalid **Ashraf**, William J. **Dally**, and Kurt **Keutzer**. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size." en. In: *arXiv:1602.07360 [cs]*, Nov. 2016. arXiv: 1602.07360. http://arxiv.org/abs/1602.07360

[42] Mingxing **Tan** and Quoc V. **Le**. "EfficientNetV2: Smaller Models and Faster Training." en. In: *arXiv:2104.00298 [cs]*, Apr. 2021. arXiv: 2104.00298. http://arxiv.org/abs/2104.00298

[43] Joseph **Redmon**, Santosh **Divvala**, Ross **Girshick**, and Ali **Farhadi**. *You Only Look Once: Unified, Real-Time Object Detection*. arXiv:1506.02640 [cs]. May 2016. doi:10.48550/arXiv.1506.02640. http://arxiv.org/abs/1506.02640

[44] Mengmeng **Zhang**, Wei **Li**, and Qian **Du**. "Diverse Region-Based CNN for Hyperspectral Image Classification." In: *IEEE Transactions on Image Processing* 27.6, June 2018. Conference Name: IEEE Transactions on Image Processing, pp. 2623–2634. ISSN: 1941-0042. doi:10.1109/TIP.2018.2809606

[45] Daniel **Bolya**, Chong **Zhou**, Fanyi **Xiao**, and Yong Jae **Lee**. *YOLACT: Real-time Instance Segmentation*. arXiv:1904.02689 [cs]. Oct. 2019. doi:10.48550/arXiv.1904.02689. http://arxiv.org/abs/1904.02689

[46] Hengshuang **Zhao**, Jianping **Shi**, Xiaojuan **Qi**, Xiaogang **Wang**, and Jiaya **Jia**. *Pyramid Scene Parsing Network*. arXiv:1612.01105 [cs]. Apr. 2017. doi:10.48550/arXiv.1612.01105. http://arxiv.org/abs/1612.01105

[47] Olaf **Ronneberger**, Philipp **Fischer**, and Thomas **Brox**. "U-Net: Convolutional Networks for Biomedical Image Segmentation." en. In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. by Nassir **Navab**, Joachim **Hornegger**, William M. **Wells**, and Alejandro F. **Frangi**. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 234–241. isbn:978-3-319-24574-4. doi:10.1007/978-3-319-24574-4_28

[48] Liang-Chieh **Chen**, George **Papandreou**, Iasonas **Kokkinos**, Kevin **Murphy**, and Alan L. **Yuille**. *DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs*. en. arXiv:1606.00915 [cs]. May 2017. http://arxiv.org/abs/1606.00915

[49] Ran **Cheng**, Ryan **Razani**, Yuan **Ren**, and Liu **Bingbing**. "S3Net: 3D LiDAR Sparse Semantic Segmentation Network." In: *2021 IEEE International Conference on Robotics and Automation* (*ICRA*). ISSN: 2577-087X. May 2021, pp. 14040–14046. doi:10.1109/ICRA48506.2021.9561305

[50] Qishen **Ha**, Kohei **Watanabe**, Takumi **Karasawa**, Yoshitaka **Ushiku**, and Tatsuya **Harada**. "MFNet: Towards real-time semantic segmentation for autonomous vehicles with multi-spectral scenes." In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems* (*IROS*). ISSN: 2153-0866. Sept. 2017, pp. 5108–5115. doi:10.1109/IROS.2017.8206396

[51] Qingxue **Zhang**, Dian **Zhou**, and Xuan **Zeng**. "HeartID: A Multiresolution Convolutional Neural Network for ECG-Based Biometric Human Identification in Smart Health Applications." In: *IEEE Access* 5, 2017, pp. 11805–11816. ISSN: 2169-3536. doi:10.1109/ACCESS.2017.2707460. http://ieeexplore.ieee.org/document/7933065/

[52] Yuxi **Dong**, Yuchao **Pan**, Jun **Zhang**, and Wei **Xu**. "Learning to Read Chest X-Ray Images from 16000+ Examples Using CNN." in: *2017 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies* (*CHASE*). July 2017, pp. 51–57. doi:10.1109/CHASE.2017.59

[53] Titus J. **Brinker** et al. "Deep learning outperformed 136 of 157 dermatologists in a head-to-head dermoscopic melanoma image classification task." en. In: *European Journal of Cancer* 113, May 2019, pp. 47–54. ISSN: 0959-8049. doi:10.1016/j.ejca.2019.04.001. http://www.sciencedirect.com/science/article/pii/S0959804919302217

[54] Jorge **Beltrán**, Carlos **Guindel**, Francisco Miguel **Moreno**, Daniel **Cruzado**, Fernando **García**, and Arturo **De La Escalera**. "BirdNet: A 3D Object Detection Framework from LiDAR Information." In: *2018 21st International Conference on Intelligent Transportation Systems* (*ITSC*). ISSN: 2153-0017. Nov. 2018, pp. 3517–3523. doi:10.1109/ITSC.2018.8569311

[55] Wenbo **Lan**, Jianwu **Dang**, Yangping **Wang**, and Song **Wang**. "Pedestrian Detection Based on YOLO Network Model." In: *2018 IEEE International Conference on Mechatronics and Automation* (*ICMA*). ISSN: 2152-744X. Aug. 2018, pp. 1547–1551. doi:10.1109/ICMA.2018.8484698

[56] Martin **Velas**, Michal **Spanel**, Michal **Hradis**, and Adam **Herout**. *CNN for Very Fast Ground Segmentation in Velodyne LiDAR Data*. arXiv:1709.02128 [cs]. Sept. 2017. doi:10.48550/arXiv.1709.02128. http://arxiv.org/abs/1709.02128

[57] Florian **Schroff**, Dmitry **Kalenichenko**, and James **Philbin**. "FaceNet: A Unified Embedding for Face Recognition and Clustering." en. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), June 2015. arXiv: 1503.03832, pp. 815–823. doi:10.1109/CVPR.2015.7298682. http://arxiv.org/abs/1503.03832

[58] Yaniv **Taigman**, Ming **Yang**, Marc'Aurelio **Ranzato**, and Lior **Wolf**. "DeepFace: Closing the Gap to Human-Level Performance in Face Verification." In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. ISSN: 1063-6919. June 2014, pp. 1701–1708. doi:10.1109/CVPR.2014.220

[59] Yi **Huang**, Shang-Hong **Lai**, and Shao-Heng **Tai**. "Human Action Recognition Based on Temporal Pose CNN and Multi-dimensional Fusion." en. In: *Computer Vision – ECCV 2018 Workshops*. Ed. by Laura **Leal-Taixé** and Stefan **Roth**. Vol. 11130. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 426–440. isbn:978-3-030-11011-6. doi:10.1007/978-3-030-11012-3_33. http://link.springer.com/10.1007/978-3-030-11012-3_33

[60] Yu-Hsin **Chen**, Tushar **Krishna**, Joel S. **Emer**, and Vivienne **Sze**. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks." In: *IEEE Journal of Solid-State Circuits* 52.1, Jan. 2017. Conference Name: IEEE Journal of Solid-State Circuits, pp. 127–138. ISSN: 1558-173X. doi:10.1109/JSSC.2016.2616357

[61] Jiantao **Qiu** et al. "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network." en. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-*

Programmable Gate Arrays. Monterey California USA: ACM, Feb. 2016, pp. 26–35. isbn:978-1-4503-3856-1. doi:10.1145/2847263.2847265. https://dl.acm.org/doi/10.1145/2847263.2847265

[62] Vivienne **Sze**, Yu-Hsin **Chen**, Tien-Ju **Yang**, and Joel S. **Emer**. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey." In: *Proceedings of the IEEE* 105.12, Dec. 2017. Conference Name: Proceedings of the IEEE, pp. 2295–2329. ISSN: 1558-2256. doi:10.1109/JPROC.2017.2761740

[63] Adam **Paszke** et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. arXiv:1912.01703 [cs, stat]. Dec. 2019. doi:10.48550/arXiv.1912.01703. http://arxiv.org/abs/1912.01703

[64] Daniel S. **Kermany** et al. "Identifying Medical Diagnoses and Treatable Diseases by Image-Based Deep Learning." English. In: *Cell* 172.5, Feb. 2018. Publisher: Elsevier, 1122–1131.e9. ISSN: 0092-8674, 1097-4172. doi:10.1016/j.cell.2018.02.010. https://www.cell.com/cell/abstract/S0092-8674(18)30154-5

[65] Aditya **Khosla**, Nityananda **Jayadevaprakash**, Bangpeng **Yao**, and Li **Fei-Fei**. "Novel Dataset for Fine-Grained Image Categorization." In: *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition*. Colorado Springs, CO, June 2011

[66] Jia **Deng**, Wei **Dong**, Richard **Socher**, Li-Jia **Li**, Kai **Li**, and Li **Fei-Fei**. "ImageNet: A large-scale hierarchical image database." In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. ISSN: 1063-6919. June 2009, pp. 248–255. doi:10.1109/CVPR.2009.5206848

[67] Alex **Krizhevsky**. "Learning Multiple Layers of Features from Tiny Images." en. In: 2009

[68] Andrei **Barbu**, David **Mayo**, Julian **Alverio**, William **Luo**, Christopher **Wang**, Dan **Gutfreund**, Josh **Tenenbaum**, and Boris **Katz**. "ObjectNet: A large-scale bias-controlled dataset for pushing the limits of object recognition models." In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. https://papers.nips.cc/paper_files/paper/2019/hash/97af07a14cacba681feacf3012730892-Abstract.html

[69] Tsung-Yi **Lin**, Michael **Maire**, Serge **Belongie**, Lubomir **Bourdev**, Ross **Girshick**, James **Hays**, Pietro **Perona**, Deva **Ramanan**, C. Lawrence **Zitnick**, and Piotr **Dollár**. *Microsoft COCO: Common Objects in Context*. Tech. rep. arXiv:1405.0312. arXiv:1405.0312 [cs] type: article. arXiv, Feb. 2015. doi:10.48550/arXiv.1405.0312. http://arxiv.org/abs/1405.0312

[70] Mark **Everingham**, Luc **Van Gool**, Christopher K. I. **Williams**, John **Winn**, and Andrew **Zisserman**. "The Pascal Visual Object Classes (VOC) Challenge." en. In: *International Journal of Computer Vision* 88.2, June 2010, pp. 303–338. ISSN: 0920-5691, 1573-1405. doi:10.1007/s11263-009-0275-4. http://link.springer.com/10.1007/s11263-009-0275-4

[71] Gui-Song **Xia**, Xiang **Bai**, Jian **Ding**, Zhen **Zhu**, Serge **Belongie**, Jiebo **Luo**, Mihai **Datcu**, Marcello **Pelillo**, and Liangpei **Zhang**. "DOTA: A Large-Scale Dataset for Object Detection in Aerial Images." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018

[72] Marius **Cordts**, Mohamed **Omran**, Sebastian **Ramos**, Timo **Rehfeld**, Markus **Enzweiler**, Rodrigo **Benenson**, Uwe **Franke**, Stefan **Roth**, and Bernt **Schiele**. "The Cityscapes Dataset for Semantic Urban Scene Understanding." en. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, June 2016, pp. 3213–3223. isbn:978-1-4673-8851-1. doi:10.1109/CVPR.2016.350. http://ieeexplore.ieee.org/document/7780719/

[73] Fisher **Yu**, Haofeng **Chen**, Xin **Wang**, Wenqi **Xian**, Yingying **Chen**, Fangchen **Liu**, Vashisht **Madhavan**, and Trevor **Darrell**. *BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning*. arXiv:1805.04687 [cs]. Apr. 2020. doi:10.48550/arXiv.1805.04687. http://arxiv.org/abs/1805.04687

[74] Andreas **Geiger**, Philip **Lenz**, Christoph **Stiller**, and Raquel **Urtasun**. "Vision meets Robotics: The KITTI Dataset." In: *International Journal of Robotics Research (IJRR)*, 2013

[75] Christopher B. **Kuhn**, Markus **Hofbauer**, Ziqin **Xu**, Goran **Petrovic**, and Eckehard **Steinbach**. "Pixel-Wise Failure Prediction For Semantic Video Segmentation." In: *2021 IEEE International Conference on Image Processing (ICIP)*. ISSN: 2381-8549. Sept. 2021, pp. 614–618. doi:10.1109/ICIP42928.2021.9506552

[76] Moritz **Menze** and Andreas **Geiger**. "Object Scene Flow for Autonomous Vehicles." In: *Conference on Computer Vision and Pattern Recognition* (*CVPR*). 2015

[77] Karen **Simonyan** and Andrew **Zisserman**. "Very Deep Convolutional Networks for Large-Scale Image Recognition." en. In: *arXiv:1409.1556 [cs]*, Apr. 2015. arXiv: 1409.1556. http://arxiv.org/abs/1409.1556

[78] Gao **Huang**, Zhuang **Liu**, Laurens van der **Maaten**, and Kilian Q. **Weinberger**. "Densely Connected Convolutional Networks." In: *arXiv:1608.06993 [cs]*, Jan. 2018. arXiv: 1608.06993. http://arxiv.org/abs/1608.06993

[79] Andrew G. **Howard**, Menglong **Zhu**, Bo **Chen**, Dmitry **Kalenichenko**, Weijun **Wang**, Tobias **Weyand**, Marco **Andreetto**, and Hartwig **Adam**. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." en. In: *arXiv:1704.04861 [cs]*, Apr. 2017. arXiv: 1704.04861. http://arxiv.org/abs/1704.04861

[80] Ningning **Ma**, Xiangyu **Zhang**, Hai-Tao **Zheng**, and Jian **Sun**. "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design." In: *arXiv:1807.11164 [cs]*, July 2018. arXiv: 1807.11164. http://arxiv.org/abs/1807.11164

[81] Wei **Liu**, Dragomir **Anguelov**, Dumitru **Erhan**, Christian **Szegedy**, Scott **Reed**, Cheng-Yang **Fu**, and Alexander C. **Berg**. "SSD: Single Shot MultiBox Detector." In: vol. 9905. arXiv:1512.02325 [cs]. 2016, pp. 21–37. doi:10.1007/978-3-319-46448-0_2. http://arxiv.org/abs/1512.02325

[82] Shaoqing **Ren**, Kaiming **He**, Ross **Girshick**, and Jian **Sun**. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. arXiv:1506.01497 [cs]. Jan. 2016. doi:10.48550/arXiv.1506.01497. http://arxiv.org/abs/1506.01497

[83] Tsung-Yi **Lin**, Priya **Goyal**, Ross **Girshick**, Kaiming **He**, and Piotr **Dollár**. *Focal Loss for Dense Object Detection*. arXiv:1708.02002 [cs]. Feb. 2018. doi:10.48550/arXiv.1708.02002. http://arxiv.org/abs/1708.02002

[84] Glenn **Jocher**. *YOLOv5 by Ultralytics*. May 2020. doi:10.5281/zenodo.3908559. https://github.com/ultralytics/yolov5

[85] Chien-Yao **Wang**, Alexey **Bochkovskiy**, and Hong-Yuan Mark **Liao**. *YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors*. en. arXiv:2207.02696 [cs]. July 2022. http://arxiv.org/abs/2207.02696

[86] Dong **Wu**, Manwen **Liao**, Weitian **Zhang**, Xinggang **Wang**, Xiang **Bai**, Wenqing **Cheng**, and Wenyu **Liu**. "YOLOP: You Only Look Once for Panoptic Driving Perception." en. In: *Machine Intelligence Research* 19.6, Dec. 2022. arXiv:2108.11250 [cs], pp. 550–562. ISSN: 2731-538X, 2731-5398. doi:10.1007/s11633-022-1339-y. http://arxiv.org/abs/2108.11250

[87] Mateusz **Buda**, Ashirbani **Saha**, and Maciej A. **Mazurowski**. "Association of genomic subtypes of lower-grade gliomas with shape features automatically extracted by a deep learning algorithm." In: *Computers in Biology and Medicine* 109, June 2019. arXiv:1906.03720 [cs, eess], pp. 218–225. ISSN: 00104825. doi:10.1016/j.compbiomed.2019.05.002. http://arxiv.org/abs/1906.03720

[88] Evan **Shelhamer**, Jonathan **Long**, and Trevor **Darrell**. "Fully Convolutional Networks for Semantic Segmentation." en. In: *arXiv:1605.06211 [cs]*, May 2016. arXiv: 1605.06211. http://arxiv.org/abs/1605.06211

[89] Xiangyu **Zhang**, Xinyu **Zhou**, Mengxiao **Lin**, and Jian **Sun**. "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices." In: *arXiv:1707.01083 [cs]*, Dec. 2017. arXiv: 1707.01083. http://arxiv.org/abs/1707.01083

[90] Ross **Girshick**, Jeff **Donahue**, Trevor **Darrell**, and Jitendra **Malik**. "Rich feature hierarchies for accurate object detection and semantic segmentation." In: *arXiv:1311.2524 [cs]*, Oct. 2014. arXiv: 1311.2524. http://arxiv.org/abs/1311.2524

[91] Tsung-Yi **Lin**, Piotr **Dollár**, Ross **Girshick**, Kaiming **He**, Bharath **Hariharan**, and Serge **Belongie**. *Feature Pyramid Networks for Object Detection*. arXiv:1612.03144 [cs]. Apr. 2017. doi:10.48550/arXiv.1612.03144. http://arxiv.org/abs/1612.03144

[92] Chuyi **Li** et al. *YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications*. en. arXiv:2209.02976 [cs]. Sept. 2022. http://arxiv.org/abs/2209.02976

[93] Alexey **Bochkovskiy**, Chien-Yao **Wang**, and Hong-Yuan Mark **Liao**. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. en. arXiv:2004.10934 [cs, eess]. Apr. 2020. http://arxiv.org/abs/2004.10934

[94] Zhengxin **Zhang**, Qingjie **Liu**, and Yunhong **Wang**. "Road Extraction by Deep Residual U-Net." In: *IEEE Geoscience and Remote Sensing Letters* 15.5, May 2018. Conference Name: IEEE Geoscience and Remote Sensing Letters, pp. 749–753. ISSN: 1558-0571. doi:10.1109/LGRS.2018.2802944

[95] Liang-Chieh **Chen**, George **Papandreou**, Florian **Schroff**, and Hartwig **Adam**. *Rethinking Atrous Convolution for Semantic Image Segmentation*. en. arXiv:1706.05587 [cs]. Dec. 2017. http://arxiv.org/abs/1706.05587

[96] Liang-Chieh **Chen**, Yukun **Zhu**, George **Papandreou**, Florian **Schroff**, and Hartwig **Adam**. *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation*. arXiv:1802.02611 [cs]. Aug. 2018. doi:10.48550/arXiv.1802.02611. http://arxiv.org/abs/1802.02611

[97] J.C. **Logue**. "From vacuum tubes to very large scale integration: a personal memoir." In: *IEEE Annals of the History of Computing* 20.3, July 1998. Conference Name: IEEE Annals of the History of Computing, pp. 55–68. ISSN: 1934-1547. doi:10.1109/85.707575

[98] *Apple unveils M3, M3 Pro, and M3 Max, the most advanced chips for a personal computer*. en-US. https://www.apple.com/newsroom/2023/10/apple-unveils-m3-m3-pro-and-m3-max-the-most-advanced-chips-for-a-personal-computer/

[99] R.H. **Dennard**, F.H. **Gaensslen**, Hwa-Nien **Yu**, V.L. **Rideout**, E. **Bassous**, and A.R. **LeBlanc**. "Design of ion-implanted MOSFET's with very small physical dimensions." In: *IEEE Journal of Solid-State Circuits* 9.5, Oct. 1974. Conference Name: IEEE Journal of Solid-State Circuits, pp. 256–268. ISSN: 1558-173X. doi:10.1109/JSSC.1974.1050511

[100] Karl **Rupp**. *42 Years of Microprocessor Trend Data | Karl Rupp*. en-US. Feb. 2018. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

[101] "IEEE Standard VHDL Language Reference Manual." In: *IEEE Std 1076-1987*, Mar. 1988. Conference Name: IEEE Std 1076-1987, pp. 1–218. doi:10.1109/IEEESTD.1988.122645

[102] "IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language." In: *IEEE Std 1364-1995*, Oct. 1996. Conference Name: IEEE Std 1364-1995, pp. 1–688. doi:10.1109/IEEESTD.1996.81542

[103] Jonathan **Bachrach**, Huy **Vo**, Brian **Richards**, Yunsup **Lee**, Andrew **Waterman**, Rimas **Avižienis**, John **Wawrzynek**, and Krste **Asanović**. "Chisel: constructing hardware in a Scala embedded language." In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. New York, NY, USA: Association for Computing Machinery, June 2012, pp. 1216–1225. isbn:978-1-4503-1199-1. doi:10.1145/2228360.2228584. https://dl.acm.org/doi/10.1145/2228360.2228584

[104] William **Stallings** and Pearson Education **Limited**. *Computer organization and architecture : eleventh edition, global edition*. New York, NY :: Pearson, 2022. https://elibrary.pearson.de/book/99.150005/9781292420080

[105] John L. **Hennessy**, David A. **Patterson**, and Krste **Asanović**. *Computer architecture: a quantitative approach*. 5th ed. OCLC: ocn755102367. Waltham, MA: Morgan Kaufmann/Elsevier, 2012. isbn:978-0-12-383872-8

[106] G. **Conte**, S. **Tommesani**, and F. **Zanichelli**. "The long and winding road to high-performance image processing with MMX/SSE." in: *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*. Sept. 2000, pp. 302–310. doi:10.1109/CAMP.2000.875989

[107] Michael J. **Flynn**. "Some Computer Organizations and Their Effectiveness." en. In: *IEEE Transactions on Computers* C-21.9, Sept. 1972, pp. 948–960. ISSN: 0018-9340. doi:10.1109/TC.1972.5009071. http://ieeexplore.ieee.org/document/5009071/

[108] Shuai **Che**, Michael **Boyer**, Jiayuan **Meng**, David **Tarjan**, Jeremy W. **Sheaffer**, and Kevin **Skadron**. "A performance study of general-purpose applications on graphics processors using CUDA." en. In: *Journal of Parallel and Distributed Computing*. General-Purpose Processing using Graphics Processing Units 68.10, Oct. 2008, pp. 1370–1380. ISSN: 0743-7315. doi:10.1016/j.jpdc.2008.05.014. https://www.sciencedirect.com/science/article/pii/S0743731508000932

[109] Naga K. **Govindaraju**, Brandon **Lloyd**, Wei **Wang**, Ming **Lin**, and Dinesh **Manocha**. "Fast computation of database operations using graphics processors." In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. SIGMOD '04. New York, NY, USA: Association for Computing Machinery, June 2004, pp. 215–226. isbn:978-1-58113-859-7. doi:10.1145/1007568.1007594. https://dl.acm.org/doi/10.1145/1007568.1007594

[110] Steve **Kilts**. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. en. John Wiley & Sons, June 2007. isbn:978-0-470-12788-9

[111] Giovanni **DeMicheli** and M. G. **Sami**. *Hardware/Software Co-Design*. en. Google-Books-ID: BuCoCAAAQBAJ. Springer Science & Business Media, Nov. 2013. isbn:978-94-009-0187-2

[112] Shihua **Huang**, Luc **Waeijen**, and Henk **Corporaal**. "How Flexible is Your Computing System?" In: *ACM Transactions on Embedded Computing Systems* 21.4, Aug. 2022, 37:1–37:41. issn: 1539-9087. doi:10.1145/3524861. https://dl.acm.org/doi/10.1145/3524861

[113] Weijie **Liu**, Peng **Zhou**, Zhe **Zhao**, Zhiruo **Wang**, Haotang **Deng**, and Qi **Ju**. *FastBERT: a Self-distilling BERT with Adaptive Inference Time*. en. Number: arXiv:2004.02178 arXiv:2004.02178 [cs]. Apr. 2020. http://arxiv.org/abs/2004.02178

[114] Samuel **Williams**, Andrew **Waterman**, and David **Patterson**. "Roofline: an insightful visual performance model for multicore architectures." In: *Communications of the ACM* 52.4, Apr. 2009, pp. 65–76. issn: 0001-0782. doi:10.1145/1498765.1498785. https://dl.acm.org/doi/10.1145/1498765.1498785

[115] Mark **Horowitz**. "1.1 Computing's energy problem (and what we can do about it)." In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. ISSN: 2376-8606. Feb. 2014, pp. 10–14. doi:10.1109/ISSCC.2014.6757323

[116] Lei **Deng**, Guoqi **Li**, Song **Han**, Luping **Shi**, and Yuan **Xie**. "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey." In: *Proceedings of the IEEE* 108.4, Apr. 2020. Conference Name: Proceedings of the IEEE, pp. 485–532. issn: 1558-2256. doi:10.1109/JPROC.2020.2976475

[117] *Zynq-7000 SoC Product Selection Guide*. 2019. https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/zynq-7000-product-selection-guide.pdf

[118] J. J. **Dongarra**, Jeremy **Du Croz**, Sven **Hammarling**, and I. S. **Duff**. "A set of level 3 basic linear algebra subprograms." In: *ACM Transactions on Mathematical Software* 16.1, Mar. 1990, pp. 1–17. issn: 0098-3500. doi:10.1145/77626.79170. https://dl.acm.org/doi/10.1145/77626.79170

[119] Kazushige **Goto** and Robert **Van De Geijn**. "High-performance implementation of the level-3 BLAS." en. In: *ACM Transactions on Mathematical Software* 35.1, July 2008, pp. 1–14. issn: 0098-3500, 1557-7295. doi:10.1145/1377603.1377607. https://dl.acm.org/doi/10.1145/1377603.1377607

[120] Yifan **Sun**, Nicolas Bohm **Agostini**, Shi **Dong**, and David **Kaeli**. "Summarizing CPU and GPU Design Trends with Product Data." en. In: *arXiv:1911.11313 [cs]*, July 2020. arXiv: 1911.11313. http://arxiv.org/abs/1911.11313

[121] *NVIDIA A100 Tensor Core GPU Architecture*. Whitepaper. NVIDIA, 2020

[122] **Kung**. "Why systolic architectures?" In: *Computer* 15.1, Jan. 1982. Conference Name: Computer, pp. 37–46. issn: 1558-0814. doi:10.1109/MC.1982.1653825

[123] H. T. **Kung** and C.E. **Leiserson**. *Algorithms for VLSI processor arrays*. Vol. Introduction to VLSI Systems. 1979

[124] Yu-Hsin **Chen**. *DNN Accelerator Architectures*. 2019

[125] *NVIDIA Deep Learning Accelerator*. http://nvdla.org/

[126] Yu-Hsin **Chen**, Tien-Ju **Yang**, Joel **Emer**, and Vivienne **Sze**. "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices." en. In: *arXiv:1807.07928 [cs]*, May 2019. arXiv: 1807.07928. http://arxiv.org/abs/1807.07928

[127] Amir **Gholami**, Sehoon **Kim**, Zhen **Dong**, Zhewei **Yao**, Michael W. **Mahoney**, and Kurt **Keutzer**. *A Survey of Quantization Methods for Efficient Neural Network Inference*. en. arXiv:2103.13630 [cs]. June 2021. http://arxiv.org/abs/2103.13630

[128] Albert **Reuther**, Peter **Michaleas**, Michael **Jones**, Vijay **Gadepally**, Siddharth **Samsi**, and Jeremy **Kepner**. "AI and ML Accelerator Survey and Trends." en. In: *2022 IEEE High*

Performance Extreme Computing Conference (HPEC). arXiv:2210.04055 [cs]. Sept. 2022, pp. 1–10. doi:10.1109/HPEC55821.2022.9926331. http://arxiv.org/abs/2210.04055

[129] Kaiyuan **Guo**, Wenshuo **Li**, Kai **Zhong**, Zhenhua **Zhu**, Shulin **Zeng**, Song **Han**, Yuan **Xie**, Peter **Debacker**, Marian **Verhelst**, and Yu **Wang**. *Neural Network Accelerator Comparison*. https://nicsefc.ee.tsinghua.edu.cn/projects/neural-network-accelerator/

[130] *NVIDIA H100 Tensor Core GPU Datasheet*. en. July 2023. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet

[131] "Versal Architecture and Product Data Sheet: Overview (DS950)." en. In: 2023

[132] Mario **Kovač**, Philippe **Notton**, Daniel **Hofman**, and Josip **Knezović**. "How Europe Is Preparing Its Core Solution for Exascale Machines and a Global, Sovereign, Advanced Computing Platform." en. In: *Mathematical and Computational Applications* 25.3, Sept. 2020. Number: 3 Publisher: Multidisciplinary Digital Publishing Institute, p. 46. doi:10.3390/mca25030046. https://www.mdpi.com/2297-8747/25/3/46

[133] Zhe **Jia**, Blake **Tillman**, Marco **Maggioni**, and Daniele Paolo **Scarpazza**. *Dissecting the Graphcore IPU Architecture via Microbenchmarking*. arXiv:1912.03413 [cs]. Dec. 2019. doi:10.48550/arXiv.1912.03413. http://arxiv.org/abs/1912.03413

[134] Rebecca **Lewington**. *An AI Chip With Unprecedented Performance To Do the Unimaginable*. en. Aug. 2021. https://www.cerebras.net/blog/an-ai-chip-with-unprecedented-performance-to-do-the-unimaginable/

[135] Pete **Bannon**, Ganesh **Venkataramanan**, Debjit **Das Sarma**, and Emil **Talpes**. "Computer and Redundancy Solution for the Full Self-Driving Computer." In: Aug. 2019, pp. 1–22. doi:10.1109/HOTCHIPS.2019.8875645

[136] Emil **Talpes** et al. "Compute Solution for Tesla's Full Self-Driving Computer." In: *IEEE Micro* 40.2, Mar. 2020. Conference Name: IEEE Micro, pp. 25–35. ISSN: 1937-4143. doi:10.1109/MM.2020.2975764

[137] *Jetson Modules, Support, Ecosystem, and Lineup*. en-US. Oct. 2020. https://developer.nvidia.com/embedded/jetson-modules

[138] *Snapdragon 8 Gen 2 Product Brief*. Tech. rep. 2022

[139] Tianshi **Chen**, Zidong **Du**, Ninghui **Sun**, Jia **Wang**, Chengyong **Wu**, Yunji **Chen**, and Olivier **Temam**. "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning." In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, Feb. 2014, pp. 269–284. isbn:978-1-4503-2305-5. doi:10.1145/2541940.2541967. https://dl.acm.org/doi/10.1145/2541940.2541967

[140] Yunji **Chen** et al. "DaDianNao: A Machine-Learning Supercomputer." In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. ISSN: 2379-3155. Dec. 2014, pp. 609–622. doi:10.1109/MICRO.2014.58

[141] Zidong **Du**, Robert **Fasthuber**, Tianshi **Chen**, Paolo **Ienne**, Ling **Li**, Tao **Luo**, Xiaobing **Feng**, Yunji **Chen**, and Olivier **Temam**. "ShiDianNao: shifting vision processing closer to the sensor." In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. New York, NY, USA: Association for Computing Machinery, June 2015, pp. 92–104. isbn:978-1-4503-3402-0. doi:10.1145/2749469.2750389. https://doi.org/10.1145/2749469.2750389

[142] Daofu **Liu**, Tianshi **Chen**, Shaoli **Liu**, Jinhong **Zhou**, Shengyuan **Zhou**, Olivier **Teman**, Xiaobing **Feng**, Xuehai **Zhou**, and Yunji **Chen**. "PuDianNao: A Polyvalent Machine Learning Accelerator." In: *ACM SIGPLAN Notices* 50.4, Mar. 2015, pp. 369–381. ISSN: 0362-1340. doi:10.1145/2775054.2694358. https://dl.acm.org/doi/10.1145/2775054.2694358

[143] Yunji **Chen**, Tianshi **Chen**, Zhiwei **Xu**, Ninghui **Sun**, and Olivier **Temam**. "DianNao family: energy-efficient hardware accelerators for machine learning." en. In: *Communications of the ACM* 59.11, Oct. 2016, pp. 105–112. ISSN: 0001-0782, 1557-7317. doi:10.1145/2996864. https://dl.acm.org/doi/10.1145/2996864

[144] Suyog **Gupta**, Ankur **Agrawal**, Kailash **Gopalakrishnan**, and Pritish **Narayanan**. "Deep Learning with Limited Numerical Precision." en. In: *International Conference on Machine*

Learning. ISSN: 1938-7228. PMLR, June 2015, pp. 1737–1746. http://proceedings.mlr.press/v37/gupta15.html

[145] Bert **Moons** and Marian **Verhelst**. “A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets.” In: *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*. June 2016, pp. 1–2. doi:10.1109/VLSIC.2016.7573525

[146] Anon74277754 **Says**. *NVDLA Deep Learning Inference Compiler is Now Open Source*. en-US. Sept. 2019. https://developer.nvidia.com/blog/nvdla/

[147] Lukas **Cavigelli**, David **Gschwend**, Christoph **Mayer**, Samuel **Willi**, Beat **Muheim**, and Luca **Benini**. “Origami: A Convolutional Network Accelerator.” In: *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. GLSVLSI ’15. New York, NY, USA: Association for Computing Machinery, May 2015, pp. 199–204. isbn:978-1-4503-3474-7. doi:10.1145/2742060.2743766. https://doi.org/10.1145/2742060.2743766

[148] Yakun Sophia **Shao** et al. “Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture.” In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 14–27. isbn:978-1-4503-6938-1. doi:10.1145/3352460.3358302. https://doi.org/10.1145/3352460.3358302

[149] Hasan **Genc** et al. “Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration.” In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. ISSN: 0738-100X. Dec. 2021, pp. 769–774. doi:10.1109/DAC18074.2021.9586216

[150] Alon **Amid** et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs.” In: *IEEE Micro* 40.4, July 2020. Conference Name: IEEE Micro, pp. 10–21. issn: 1937-4143. doi:10.1109/MM.2020.2996616

[151] Angshuman **Parashar**, Minsoo **Rhu**, Anurag **Mukkara**, Antonio **Puglielli**, Rangharajan **Venkatesan**, Brucek **Khailany**, Joel **Emer**, Stephen W. **Keckler**, and William J. **Dally**. “SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks.” In: *ACM SIGARCH Computer Architecture News* 45.2, June 2017, pp. 27–40. issn: 0163-5964. doi:10.1145/3140659.3080254. https://doi.org/10.1145/3140659.3080254

[152] Wenyan **Lu**, Guihai **Yan**, Jiajun **Li**, Shijun **Gong**, Yinhe **Han**, and Xiaowei **Li**. “FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks.” In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). ISSN: 2378-203X. Feb. 2017, pp. 553–564. doi:10.1109/HPCA.2017.29

[153] Hyoukjun **Kwon**, Ananda **Samajdar**, and Tushar **Krishna**. “MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects.” In: *ACM SIGPLAN Notices* 53.2, Mar. 2018, pp. 461–475. issn: 0362-1340. doi:10.1145/3296957.3173176. https://dl.acm.org/doi/10.1145/3296957.3173176

[154] Shimeng **Yu**, Hongwu **Jiang**, Shanshi **Huang**, Xiaochen **Peng**, and Anni **Lu**. “Compute-in-Memory Chips for Deep Learning: Recent Trends and Prospects.” In: *IEEE Circuits and Systems Magazine* 21.3, 2021. Conference Name: IEEE Circuits and Systems Magazine, pp. 31–56. issn: 1558-0830. doi:10.1109/MCAS.2021.3092533

[155] Tom **Glint**, Chandan Kumar **Jha**, Manu **Awasthi**, and Joycee **Mekie**. “Analysis of Conventional, Near-Memory, and In-Memory DNN Accelerators.” In: *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Apr. 2023, pp. 349–351. doi:10.1109/ISPASS57527.2023.00049

[156] Ping-Chun **Wu** et al. “A 28nm 1Mb Time-Domain Computing-in-Memory 6T-SRAM Macro with a 6.6ns Latency, 1241GOPS and 37.01TOPS/W for 8b-MAC Operations for Edge-AI Devices.” In: *2022 IEEE International Solid- State Circuits Conference (ISSCC)*. vol. 65. ISSN: 2376-8606. Feb. 2022, pp. 1–3. doi:10.1109/ISSCC42614.2022.9731681

[157] Laura **Fick**, Skylar **Skrzyniarz**, Malav **Parikh**, Michael B. **Henry**, and David **Fick**. “Analog Matrix Processor for Edge AI Real-Time Video Analytics.” In: *2022 IEEE International Solid-*

State Circuits Conference (ISSCC). vol. 65. ISSN: 2376-8606. Feb. 2022, pp. 260–262. doi:10.1109/ISSCC42614.2022.9731773

[158] Yexin **Yan**, David **Kappel**, Felix **Neumaerker**, Johannes **Partzsch**, Bernhard **Vogginger**, Sebastian **Hoeppner**, Steve **Furber**, Wolfgang **Maass**, Robert **Legenstein**, and Christian **Mayr**. "Efficient Reward-Based Structural Plasticity on a SpiNNaker 2 Prototype." In: *IEEE Transactions on Biomedical Circuits and Systems* 13.3, June 2019, pp. 579–591. ISSN: 1932-4545, 1940-9990. doi:10.1109/TBCAS.2019.2906401. arXiv: 1903.08500[cs]. http://arxiv.org/abs/1903.08500

[159] Ben Varkey **Benjamin**, Peiran **Gao**, Emmett **McQuinn**, Swadesh **Choudhary**, Anand R. **Chandrasekaran**, Jean-Marie **Bussat**, Rodrigo **Alvarez-Icaza**, John V. **Arthur**, Paul A. **Merolla**, and Kwabena **Boahen**. "Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations." In: *Proceedings of the IEEE* 102.5, May 2014. Conference Name: Proceedings of the IEEE, pp. 699–716. ISSN: 1558-2256. doi:10.1109/JPROC.2014.2313565

[160] Aboozar **Taherkhani**, Ammar **Belatreche**, Yuhua **Li**, Georgina **Cosma**, Liam P. **Maguire**, and T.M. **McGinnity**. "A review of learning in biologically plausible spiking neural networks." In: *Neural Networks* 122, Feb. 2020, pp. 253–272. ISSN: 08936080. doi:10.1016/j.neunet.2019.09.036. https://linkinghub.elsevier.com/retrieve/pii/S0893608019303181

[161] Torsten **Hoefler**, Dan **Alistarh**, Tal **Ben-Nun**, Nikoli **Dryden**, and Alexandra **Peste**. *Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks*. en. arXiv:2102.00554 [cs]. Jan. 2021. http://arxiv.org/abs/2102.00554

[162] Yu-Hsin **Chen**, Joel **Emer**, and Vivienne **Sze**. "Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators." In: *IEEE Micro* 37.3, 2017. Conference Name: IEEE Micro, pp. 12–21. ISSN: 1937-4143. doi:10.1109/MM.2017.54

[163] A. **Parashar**, P. **Raina**, Y. S. **Shao**, Y.-H. **Chen**, V. A. **Ying**, A. **Mukkara**, R. **Venkatesan**, B. **Khailany**, S. W. **Keckler**, and J. **Emer**. "Timeloop: A Systematic Approach to DNN Accelerator Evaluation." In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2019, pp. 304–315. doi:10.1109/ISPASS.2019.00042

[164] Mingzhen **Li**, Yi **Liu**, Xiaoyan **Liu**, Qingxiao **Sun**, Xin **You**, Hailong **Yang**, Zhongzhi **Luan**, Lin **Gan**, Guangwen **Yang**, and Depei **Qian**. "The Deep Learning Compiler: A Comprehensive Survey." In: *IEEE Transactions on Parallel and Distributed Systems* 32.3, Mar. 2021, pp. 708–727. ISSN: 1045-9219, 1558-2183, 2161-9883. doi:10.1109/TPDS.2020.3030548. https://ieeexplore.ieee.org/document/9222299/

[165] Tianqi **Chen** et al. *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*. arXiv:1802.04799 [cs]. Oct. 2018. doi:10.48550/arXiv.1802.04799. http://arxiv.org/abs/1802.04799

[166] Richard **Wei**, Lane **Schwartz**, and Vikram **Adve**. *DLVM: A modern compiler infrastructure for deep learning systems*. arXiv:1711.03016 [cs]. Feb. 2018. doi:10.48550/arXiv.1711.03016. http://arxiv.org/abs/1711.03016

[167] Chris **Lattner**, Mehdi **Amini**, Uday **Bondhugula**, Albert **Cohen**, Andy **Davis**, Jacques **Pienaar**, River **Riddle**, Tatiana **Shpeisman**, Nicolas **Vasilache**, and Oleksandr **Zinenko**. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation." en. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Seoul, Korea (South): IEEE, Feb. 2021, pp. 2–14. isbn:978-1-72818-613-9. doi:10.1109/CGO51591.2021.9370308. https://ieeexplore.ieee.org/document/9370308/

[168] *onnx/onnx · Discussions*. GitHub. https://github.com/onnx/onnx/discussions

[169] Yannan Nellie **Wu**, Joel S. **Emer**, and Vivienne **Sze**. "Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs." en. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Westminster, CO, USA: IEEE, Nov. 2019, pp. 1–8. isbn:978-1-72812-350-9. doi:10.1109/ICCAD45719.2019.8942149. https://ieeexplore.ieee.org/document/8942149/

[170] John **Nickolls** and William J. **Dally**. "The GPU Computing Era." In: *IEEE Micro* 30.2, Mar. 2010. Conference Name: IEEE Micro, pp. 56–69. ISSN: 1937-4143. doi:10.1109/MM.2010.41. https://ieeexplore.ieee.org/document/5446251

[171] Michael **Pellauer**, Yakun Sophia **Shao**, Jason **Clemons**, Neal **Crago**, Kartik **Hegde**, Rangharajan **Ventakesan**, Stephen W **Keckler**, Christopher W **Fletcher**, and Joel **Emer**. "Buffets: An

Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration." en. In: 2019, p. 15
[172] Jeremy **Fowers** et al. "A Configurable Cloud-Scale DNN Processor for Real-Time AI." in: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. ISSN: 2575-713X. June 2018, pp. 1–14. doi:10.1109/ISCA.2018.00012. https://ieeexplore.ieee.org/document/8416814
[173] Pengzhen **Ren**, Yun **Xiao**, Xiaojun **Chang**, Po-Yao **Huang**, Zhihui **Li**, Xiaojiang **Chen**, and Xin **Wang**. *A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions*. Mar. 2, 2021. arXiv: 2006.02903[cs,stat]. http://arxiv.org/abs/2006.02903
[174] Barret **Zoph** and Quoc V. **Le**. *Neural Architecture Search with Reinforcement Learning*. Feb. 15, 2017. arXiv: 1611.01578[cs]. http://arxiv.org/abs/1611.01578
[175] Hanxiao **Liu**, Karen **Simonyan**, and Yiming **Yang**. *DARTS: Differentiable Architecture Search*. Apr. 23, 2019. doi:10.48550/arXiv.1806.09055. arXiv: 1806.09055[cs,stat]. http://arxiv.org/abs/1806.09055
[176] Chenxi **Liu**, Liang-Chieh **Chen**, Florian **Schroff**, Hartwig **Adam**, Wei **Hua**, Alan **Yuille**, and Li **Fei-Fei**. *Auto-DeepLab: Hierarchical Neural Architecture Search for Semantic Image Segmentation*. Number: arXiv:1901.02985 arXiv:1901.02985 [cs]. Apr. 2019. doi:10.48550/arXiv.1901.02985. http://arxiv.org/abs/1901.02985
[177] Albert **Shaw**, Daniel **Hunter**, Forrest **Iandola**, and Sammy **Sidhu**. *SqueezeNAS: Fast neural architecture search for faster semantic segmentation*. en. Number: arXiv:1908.01748 arXiv:1908.01748 [cs]. Aug. 2019. http://arxiv.org/abs/1908.01748
[178] Sheng-Chun **Kao**, Geonhwa **Jeong**, and Tushar **Krishna**. "ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning." en. In: *arXiv:2009.02010 [cs, eess]*, Sept. 2020. arXiv: 2009.02010. http://arxiv.org/abs/2009.02010
[179] D. E. **Goldberg**. "Genetic Algorithms in Search, Optimization & Machine Learning." In: 1989. https://www.semanticscholar.org/paper/Genetic-Algorithms-in-Search%2C-Optimization-%26-Goldberg/ccf73e723d8308f28a98cb435dfb585581f59e2c
[180] Nicolas **Vasilache**, Oleksandr **Zinenko**, Theodoros **Theodoridis**, Priya **Goyal**, Zachary **DeVito**, William S. **Moses**, Sven **Verdoolaege**, Andrew **Adams**, and Albert **Cohen**. *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*. arXiv:1802.04730 [cs]. June 2018. doi:10.48550/arXiv.1802.04730. http://arxiv.org/abs/1802.04730
[181] Krishna Teja **Chitty-Venkata** and Arun K. **Somani**. "Neural Architecture Search Survey: A Hardware Perspective." In: *ACM Computing Surveys* 55.4, Apr. 30, 2023, pp. 1–36. ISSN: 0360-0300, 1557-7341. doi:10.1145/3524500. https://dl.acm.org/doi/10.1145/3524500
[182] Han **Cai**, Chuang **Gan**, Tianzhe **Wang**, Zhekai **Zhang**, and Song **Han**. *Once-for-All: Train One Network and Specialize it for Efficient Deployment*. Apr. 29, 2020. arXiv: 1908.09791[cs,stat]. http://arxiv.org/abs/1908.09791
[183] Chi-Hung **Hsu**, Shu-Huan **Chang**, Jhao-Hong **Liang**, Hsin-Ping **Chou**, Chun-Hao **Liu**, Shih-Chieh **Chang**, Jia-Yu **Pan**, Yu-Ting **Chen**, Wei **Wei**, and Da-Cheng **Juan**. *MONAS: Multi-Objective Neural Architecture Search using Reinforcement Learning*. Dec. 3, 2018. doi:10.48550/arXiv.1806.10332. arXiv: 1806.10332[cs,stat]. http://arxiv.org/abs/1806.10332
[184] Mingxing **Tan**, Bo **Chen**, Ruoming **Pang**, Vijay **Vasudevan**, Mark **Sandler**, Andrew **Howard**, and Quoc V. **Le**. *MnasNet: Platform-Aware Neural Architecture Search for Mobile*. arXiv:1807.11626 [cs]. May 2019. doi:10.48550/arXiv.1807.11626. http://arxiv.org/abs/1807.11626
[185] Bichen **Wu**, Xiaoliang **Dai**, Peizhao **Zhang**, Yanghan **Wang**, Fei **Sun**, Yiming **Wu**, Yuandong **Tian**, Peter **Vajda**, Yangqing **Jia**, and Kurt **Keutzer**. *FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search*. May 24, 2019. doi:10.48550/arXiv.1812.03443. arXiv: 1812.03443[cs]. http://arxiv.org/abs/1812.03443
[186] Suyog **Gupta** and Berkin **Akin**. *Accelerator-aware Neural Network Design using AutoML*. Mar. 5, 2020. doi:10.48550/arXiv.2003.02838. arXiv: 2003.02838[cs,eess,stat]. http://arxiv.org/abs/2003.02838
[187] Edgar **Liberis**, Łukasz **Dudziak**, and Nicholas D. **Lane**. "μNAS: Constrained Neural Architecture Search for Microcontrollers." In: *Proceedings of the 1st Workshop on Machine Learning and Systems*. EuroSys '21: Sixteenth European Conference on Computer Systems. Online

United Kingdom: ACM, Apr. 26, 2021, pp. 70–79. isbn:978-1-4503-8298-4. doi:10.1145/3437984.3458836. https://dl.acm.org/doi/10.1145/3437984.3458836

[188] Sheng **Li**, Mingxing **Tan**, Ruoming **Pang**, Andrew **Li**, Liqun **Cheng**, Quoc **Le**, and Norman P. **Jouppi**. *Searching for Fast Model Families on Datacenter Accelerators*. Feb. 10, 2021. doi:10.48550/arXiv.2102.05610. arXiv: 2102.05610[cs,eess]. http://arxiv.org/abs/2102.05610

[189] Hongxiang **Fan**, Martin **Ferianc**, Shuanglong **Liu**, Zhiqiang **Que**, Xinyu **Niu**, and Wayne **Luk**. "Optimizing FPGA-Based CNN Accelerator Using Differentiable Neural Architecture Search." In: *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 2020 IEEE 38th International Conference on Computer Design (ICCD). ISSN: 2576-6996. Oct. 2020, pp. 465–468. doi:10.1109/ICCD50377.2020.00085

[190] Tianqi **Chen**, Lianmin **Zheng**, Eddie **Yan**, Ziheng **Jiang**, Thierry **Moreau**, Luis **Ceze**, Carlos **Guestrin**, and Arvind **Krishnamurthy**. *Learning to Optimize Tensor Programs*. en. arXiv:1805.08166 [cs, stat]. Jan. 2019. http://arxiv.org/abs/1805.08166

[191] Hyoukjun **Kwon**, Prasanth **Chatarasi**, Michael **Pellauer**, Angshuman **Parashar**, Vivek **Sarkar**, and Tushar **Krishna**. "Understanding Reuse, Performance, and Hardware Cost of DNN Dataflows: A Data-Centric Approach Using MAESTRO." en. In: *arXiv:1805.02566 [cs]*, May 2020. arXiv: 1805.02566. http://arxiv.org/abs/1805.02566

[192] Tien-Ju **Yang**, Yu-Hsin **Chen**, Joel **Emer**, and Vivienne **Sze**. "A method to estimate the energy consumption of deep neural networks." en. In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. Pacific Grove, CA, USA: IEEE, Oct. 2017, pp. 1916–1920. isbn:978-1-5386-1823-3. doi:10.1109/ACSSC.2017.8335698. http://ieeexplore.ieee.org/document/8335698/

[193] Yakun Sophia **Shao**, Brandon **Reagen**, Gu-Yeon **Wei**, and David **Brooks**. "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures." In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. ISSN: 1063-6897. June 2014, pp. 97–108. doi:10.1109/ISCA.2014.6853196

[194] Shyamkumar **Thoziyoor**, Norman P. **Jouppi**, Naveen **Muralimanohar**, and Jung Ho **Ahn**. *CACTI 5.1*. Tech. rep. HPL-2008-20. HP Laboratories, Palo Alto, Apr. 2008. https://www.hpl.hp.com/techreports/2008/HPL-2008-20.pdf

[195] Jean-Michel **Muller**, Nicolas **Brisebarre**, Florent de **Dinechin**, Claude-Pierre **Jeannerod**, Vincent **Lefèvre**, Guillaume **Melquiond**, Nathalie **Revol**, Damien **Stehlé**, and Serge **Torres**. *Handbook of Floating-Point Arithmetic*. en. Google-Books-ID: baFvrIOPvncC. Springer Science & Business Media, Nov. 2009. isbn:978-0-8176-4705-6

[196] "IEEE Standard for Floating-Point Arithmetic." In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, July 2019. Conference Name: IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1–84. doi:10.1109/IEEESTD.2019.8766229

[197] Dipankar **Das** et al. *Mixed Precision Training of Convolutional Neural Networks using Integer Operations*. en. arXiv:1802.00930 [cs]. Feb. 2018. http://arxiv.org/abs/1802.00930

[198] **Intel**. *bfloat16 - Hardware Numerics Definition*. en. Whitepaper 338302-001US. Intel, Nov. 2018

[199] Jeffrey **Dean** et al. "Large Scale Distributed Deep Networks." In: *Advances in Neural Information Processing Systems*. Vol. 25. Curran Associates, Inc., 2012. https://papers.nips.cc/paper_files/paper/2012/hash/6aca97005c68f1206823815f66102863-Abstract.html

[200] Vincent **Vanhoucke**, Andrew **Senior**, and Mark Z. **Mao**. "Improving the speed of neural networks on CPUs." In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*. 2011

[201] John L. **Gustafson**. *Posit Arithmetic*. Oct. 2017

[202] **Gustafson** and **Yonemoto**. "Beating Floating Point at its Own Game: Posit Arithmetic." In: *Supercomputing Frontiers and Innovations: an International Journal* 4.2, June 2017, pp. 71–86. issn: 2409-6008. doi:10.14529/jsfi170206. https://doi.org/10.14529/jsfi170206

[203] Zachariah **Carmichael**, Hamed F. **Langroudi**, Char **Khazanov**, Jeffrey **Lillie**, John L. **Gustafson**, and Dhireesha **Kudithipudi**. *Deep Positron: A Deep Neural Network Using the Posit Number System*. en. arXiv:1812.01762 [cs]. Jan. 2019. http://arxiv.org/abs/1812.01762

[204] Gonçalo **Raposo**, Pedro **Tomás**, and Nuno **Roma**. "Positnn: Training Deep Neural Networks with Mixed Low-Precision Posit." In: *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. ISSN: 2379-190X. June 2021, pp. 7908–7912. doi:10.1109/ICASSP39728.2021.9413919

[205] Jeff **Johnson**. *Rethinking floating point for deep learning*. en. arXiv:1811.01721 [cs]. Nov. 2018. http://arxiv.org/abs/1811.01721

[206] Misha **Denil**, Babak **Shakibi**, Laurent **Dinh**, Marc'Aurelio **Ranzato**, and Nando de **Freitas**. *Predicting Parameters in Deep Learning*. Oct. 27, 2014. doi:10.48550/arXiv.1306.0543. arXiv: 1306.0543[cs,stat]. http://arxiv.org/abs/1306.0543

[207] Hao **Wu**, Patrick **Judd**, Xiaojie **Zhang**, Mikhail **Isaev**, and Paulius **Micikevicius**. *Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation*. en. arXiv:2004.09602 [cs, stat]. Apr. 2020. http://arxiv.org/abs/2004.09602

[208] Benoit **Jacob**, Skirmantas **Kligys**, Bo **Chen**, Menglong **Zhu**, Matthew **Tang**, Andrew **Howard**, Hartwig **Adam**, and Dmitry **Kalenichenko**. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. arXiv:1712.05877 [cs, stat]. Dec. 2017. doi:10.48550/arXiv.1712.05877. http://arxiv.org/abs/1712.05877

[209] Daisuke **Miyashita**, Edward H. **Lee**, and Boris **Murmann**. *Convolutional Neural Networks using Logarithmic Data Representation*. arXiv:1603.01025 [cs]. Mar. 2016. doi:10.48550/arXiv.1603.01025. http://arxiv.org/abs/1603.01025

[210] Denis A. **Gudovskiy** and Luca **Rigazio**. *ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks*. arXiv:1706.02393 [cs]. June 2017. doi:10.48550/arXiv.1706.02393. http://arxiv.org/abs/1706.02393

[211] Zhaohui **Yang**, Yunhe **Wang**, Kai **Han**, Chunjing **Xu**, Chao **Xu**, Dacheng **Tao**, and Chang **Xu**. *Searching for Low-Bit Weights in Quantized Neural Networks*. arXiv:2009.08695 [cs]. Sept. 2020. doi:10.48550/arXiv.2009.08695. http://arxiv.org/abs/2009.08695

[212] Dongqing **Zhang**, Jiaolong **Yang**, Dongqiangzi **Ye**, and Gang **Hua**. *LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks*. arXiv:1807.10029 [cs]. July 2018. doi:10.48550/arXiv.1807.10029. http://arxiv.org/abs/1807.10029

[213] Sangil **Jung**, Changyong **Son**, Seohyung **Lee**, Jinwoo **Son**, Youngjun **Kwak**, Jae-Joon **Han**, Sung Ju **Hwang**, and Changkyu **Choi**. *Learning to Quantize Deep Networks by Optimizing Quantization Intervals with Task Loss*. arXiv:1808.05779 [cs]. Nov. 2018. doi:10.48550/arXiv.1808.05779. http://arxiv.org/abs/1808.05779

[214] Kohei **Yamamoto**. "Learnable Companding Quantization for Accurate Low-bit Neural Networks." In: *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. ISSN: 2575-7075. June 2021, pp. 5027–5036. doi:10.1109/CVPR46437.2021.00499

[215] Philipp **Gysel**, Jon **Pimentel**, Mohammad **Motamedi**, and Soheil **Ghiasi**. "Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks." In: *IEEE Transactions on Neural Networks and Learning Systems* 29.11, Nov. 2018. Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 5784–5789. ISSN: 2162-2388. doi:10.1109/TNNLS.2018.2808319

[216] Ron **Banner**, Yury **Nahshan**, Elad **Hoffer**, and Daniel **Soudry**. *Post-training 4-bit quantization of convolution networks for rapid-deployment*. en. arXiv:1810.05723 [cs]. May 2019. http://arxiv.org/abs/1810.05723

[217] Xiaotian **Zhu**, Wengang **Zhou**, and Houqiang **Li**. "Adaptive Layerwise Quantization for Deep Neural Network Compression." In: *2018 IEEE International Conference on Multimedia and Expo (ICME)*. ISSN: 1945-788X. July 2018, pp. 1–6. doi:10.1109/ICME.2018.8486500

[218] Moran **Shkolnik**, Brian **Chmiel**, Ron **Banner**, Gil **Shomron**, Yury **Nahshan**, Alex **Bronstein**, and Uri **Weiser**. *Robust Quantization: One Model to Rule Them All*. arXiv:2002.07686 [cs, stat]. Oct. 2020. doi:10.48550/arXiv.2002.07686. http://arxiv.org/abs/2002.07686

[219] Zhuoran **Song**, Bangqi **Fu**, Feiyang **Wu**, Zhaoming **Jiang**, Li **Jiang**, Naifeng **Jing**, and Xiaoyao **Liang**. "DRQ: Dynamic Region-based Quantization for Deep Neural Network Acceleration."

In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. May 2020, pp. 1010–1021. doi:10.1109/ISCA45697.2020.00086

[220] Stefan **Uhlich**, Lukas **Mauch**, Fabien **Cardinaux**, Kazuki **Yoshiyama**, Javier Alonso **García**, Stephen **Tiedemann**, Thomas **Kemp**, and Akira **Nakamura**. "MIXED PRECISION DNNS: ALL YOU NEED IS A GOOD PARAMETRIZATION." en. In: 2020

[221] Jungwook **Choi**, Zhuo **Wang**, Swagath **Venkataramani**, Pierce I.-Jen **Chuang**, Vijayalakshmi **Srinivasan**, and Kailash **Gopalakrishnan**. *PACT: Parameterized Clipping Activation for Quantized Neural Networks*. en. arXiv:1805.06085 [cs]. July 2018. http://arxiv.org/abs/1805.06085

[222] Bert **Moons**, Bert **De Brabandere**, Luc **Van Gool**, and Marian **Verhelst**. "Energy-Efficient ConvNets Through Approximate Computing." en. In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*. arXiv:1603.06777 [cs]. Mar. 2016, pp. 1–8. doi:10.1109/WACV.2016.7477614. http://arxiv.org/abs/1603.06777

[223] Patrick **Judd**, Jorge **Albericio**, Tayler **Hetherington**, Tor M. **Aamodt**, and Andreas **Moshovos**. "Stripes: Bit-serial deep neural network computing." In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12. doi:10.1109/MICRO.2016.7783722

[224] Sayeh **Sharify**, Alberto Delmas **Lascorz**, Kevin **Siu**, Patrick **Judd**, and Andreas **Moshovos**. "Loom: Exploiting Weight and Activation Precisions to Accelerate Convolutional Neural Networks." In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. June 2018, pp. 1–6. doi:10.1109/DAC.2018.8465915

[225] Jinmook **Lee**, Changhyeon **Kim**, Sanghoon **Kang**, Dongjoo **Shin**, Sangyeob **Kim**, and Hoi-Jun **Yoo**. "UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision." In: *IEEE Journal of Solid-State Circuits* 54.1, Jan. 2019. Conference Name: IEEE Journal of Solid-State Circuits, pp. 173–185. ISSN: 1558-173X. doi:10.1109/JSSC.2018.2865489

[226] Yaman **Umuroglu**, Lahiru **Rasnayake**, and Magnus **Sjalander**. *BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing*. en. arXiv:1806.08862 [cs]. June 2018. http://arxiv.org/abs/1806.08862

[227] Shuchang **Zhou**, Yuxin **Wu**, Zekun **Ni**, Xinyu **Zhou**, He **Wen**, and Yuheng **Zou**. *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*. arXiv:1606.06160 [cs]. Feb. 2018. doi:10.48550/arXiv.1606.06160. http://arxiv.org/abs/1606.06160

[228] Yash **Bhalgat**, Jinwon **Lee**, Markus **Nagel**, Tijmen **Blankevoort**, and Nojun **Kwak**. *LSQ+: Improving low-bit quantization through learnable offsets and better initialization*. arXiv:2004.09576 [cs, stat]. Apr. 2020. doi:10.48550/arXiv.2004.09576. http://arxiv.org/abs/2004.09576

[229] Chunyu **Yuan** and Sos S. **Agaian**. "A comprehensive review of Binary Neural Network." In: *Artificial Intelligence Review*, Mar. 2023. arXiv:2110.06804 [cs]. ISSN: 0269-2821, 1573-7462. doi:10.1007/s10462-023-10464-w. http://arxiv.org/abs/2110.06804

[230] Matthieu **Courbariaux**, Itay **Hubara**, Daniel **Soudry**, Ran **El-Yaniv**, and Yoshua **Bengio**. "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1." en. In: *arXiv:1602.02830 [cs]*, Mar. 2016. arXiv: 1602.02830. http://arxiv.org/abs/1602.02830

[231] Mohammad **Rastegari**, Vicente **Ordonez**, Joseph **Redmon**, and Ali **Farhadi**. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*. arXiv:1603.05279 [cs]. Aug. 2016. doi:10.48550/arXiv.1603.05279. http://arxiv.org/abs/1603.05279

[232] Haotong **Qin**, Ruihao **Gong**, Xianglong **Liu**, Mingzhu **Shen**, Ziran **Wei**, Fengwei **Yu**, and Jingkuan **Song**. *Forward and Backward Information Retention for Accurate Binary Neural Networks*. arXiv:1909.10788 [cs]. Mar. 2020. doi:10.48550/arXiv.1909.10788. http://arxiv.org/abs/1909.10788

[233] Mingbao **Lin**, Rongrong **Ji**, Zihan **Xu**, Baochang **Zhang**, Yan **Wang**, Yongjian **Wu**, Feiyue **Huang**, and Chia-Wen **Lin**. *Rotated Binary Neural Network*. arXiv:2009.13055 [cs]. Oct. 2020. doi:10.48550/arXiv.2009.13055. http://arxiv.org/abs/2009.13055

[234] Taylor **Simons** and Dah-Jye **Lee**. "A Review of Binarized Neural Networks." en. In: *Electronics* 8.6, June 2019. Number: 6 Publisher: Multidisciplinary Digital Publishing Institute, p. 661. ISSN: 2079-9292. doi:10.3390/electronics8060661. https://www.mdpi.com/2079-9292/8/6/661

[235] Yaman **Umuroglu**, Nicholas J. **Fraser**, Giulio **Gambardella**, Michaela **Blott**, Philip **Leong**, Magnus **Jahre**, and Kees **Vissers**. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference." In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. New York, NY, USA: Association for Computing Machinery, Feb. 2017, pp. 65–74. isbn:978-1-4503-4354-1. doi:10.1145/3020078.3021744. https://doi.org/10.1145/3020078.3021744

[236] Tailin **Liang**, John **Glossner**, Lei **Wang**, Shaobo **Shi**, and Xiaotong **Zhang**. "Pruning and Quantization for Deep Neural Network Acceleration: A Survey." en. In: *arXiv:2101.09671 [cs]*, June 2021. arXiv: 2101.09671. http://arxiv.org/abs/2101.09671

[237] Sangkug **Lym**, Esha **Choukse**, Siavash **Zangeneh**, Wei **Wen**, Sujay **Sanghavi**, and Mattan **Erez**. "PruneTrain: Fast Neural Network Training by Dynamic Sparse Model Reconfiguration." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. arXiv:1901.09290 [cs, stat]. Nov. 2019, pp. 1–13. doi:10.1145/3295500.3356156. http://arxiv.org/abs/1901.09290

[238] Song **Han**, Xingyu **Liu**, Huizi **Mao**, Jing **Pu**, Ardavan **Pedram**, Mark A. **Horowitz**, and William J. **Dally**. *EIE: Efficient Inference Engine on Compressed Deep Neural Network*. arXiv:1602.01528 [cs]. May 2016. doi:10.48550/arXiv.1602.01528. http://arxiv.org/abs/1602.01528

[239] Song **Han**, Huizi **Mao**, and W. **Dally**. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding." In: *arXiv: Computer Vision and Pattern Recognition*, Oct. 2015. https://www.semanticscholar.org/paper/Deep-Compression%3A-Compressing-Deep-Neural-Network-Han-Mao/642d0f49b7826adcf986616f4af77e736229990f

[240] Hao **Li**, Hanan **Samet**, Asim **Kadav**, Igor **Durdanovic**, and Hans Peter **Graf**. "PRUNING FILTERS FOR EFFICIENT CONVNETS." en. In: 2017

[241] Sajid **Anwar**, Kyuyeon **Hwang**, and Wonyong **Sung**. "Structured Pruning of Deep Convolutional Neural Networks." In: *ACM Journal on Emerging Technologies in Computing Systems* 13.3, Feb. 2017, 32:1–32:18. issn: 1550-4832. doi:10.1145/3005348. https://dl.acm.org/doi/10.1145/3005348

[242] Trevor **Gale**, Erich **Elsen**, and Sara **Hooker**. *The State of Sparsity in Deep Neural Networks*. arXiv:1902.09574 [cs, stat]. Feb. 2019. doi:10.48550/arXiv.1902.09574. http://arxiv.org/abs/1902.09574

[243] Maxwell D. **Collins** and Pushmeet **Kohli**. *Memory Bounded Deep Convolutional Networks*. arXiv:1412.1442 [cs]. Dec. 2014. doi:10.48550/arXiv.1412.1442. http://arxiv.org/abs/1412.1442

[244] Jian-Hao **Luo**, Jianxin **Wu**, and Weiyao **Lin**. *ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression*. arXiv:1707.06342 [cs]. July 2017. doi:10.48550/arXiv.1707.06342. http://arxiv.org/abs/1707.06342

[245] Ji **Lin**, Yongming **Rao**, Jiwen **Lu**, and Jie **Zhou**. "Runtime Neural Pruning." In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. https://papers.nips.cc/paper_files/paper/2017/hash/a51fb975227d6640e4fe47854476d133-Abstract.html

[246] Jianda **Chen**, Shangyu **Chen**, and Sinno Jialin **Pan**. "Storage Efficient and Dynamic Flexible Runtime Channel Pruning via Deep Reinforcement Learning." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 14747–14758. https://proceedings.neurips.cc/paper/2020/hash/a914ecef9c12ffdb9bede64bb703d877-Abstract.html

[247] Hong-Gui **Han** and Jun-Fei **Qiao**. "A structure optimisation algorithm for feedforward neural network construction." en. In: *Neurocomputing* 99, Jan. 2013, pp. 347–357. issn: 0925-2312. doi:10.1016/j.neucom.2012.07.023. https://www.sciencedirect.com/science/article/pii/S0925231212005929

[248] Yi **Sun**, Xiaogang **Wang**, and Xiaoou **Tang**. *Sparsifying Neural Network Connections for Face Recognition*. arXiv:1512.01891 [cs]. Dec. 2015. doi:10.48550/arXiv.1512.01891. http://arxiv.org/abs/1512.01891

[249] Davis **Blalock**, Jose Javier Gonzalez **Ortiz**, Jonathan **Frankle**, and John **Guttag**. *What is the State of Neural Network Pruning?* arXiv:2003.03033 [cs, stat]. Mar. 2020. doi:10.48550/arXiv.2003.03033. http://arxiv.org/abs/2003.03033

[250] Song **Han**, Jeff **Pool**, John **Tran**, and William J. **Dally**. "Learning both Weights and Connections for Efficient Neural Networks." en. In: *arXiv:1506.02626 [cs]*, Oct. 2015. arXiv: 1506.02626. http://arxiv.org/abs/1506.02626

[251] Tien-Ju **Yang**, Yu-Hsin **Chen**, and Vivienne **Sze**. *Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning*. arXiv:1611.05128 [cs]. Apr. 2017. doi:10.48550/arXiv.1611.05128. http://arxiv.org/abs/1611.05128

[252] Zhuohan **Li**, Eric **Wallace**, Sheng **Shen**, Kevin **Lin**, Kurt **Keutzer**, Dan **Klein**, and Joseph E. **Gonzalez**. *Train Large, Then Compress: Rethinking Model Size for Efficient Training and Inference of Transformers*. arXiv:2002.11794 [cs]. June 2020. doi:10.48550/arXiv.2002.11794. http://arxiv.org/abs/2002.11794

[253] Victor **Sanh**, Thomas **Wolf**, and Alexander M. **Rush**. *Movement Pruning: Adaptive Sparsity by Fine-Tuning*. arXiv:2005.07683 [cs]. Oct. 2020. doi:10.48550/arXiv.2005.07683. http://arxiv.org/abs/2005.07683

[254] Jiecao **Yu**, Andrew **Lukefahr**, David **Palframan**, Ganesh **Dasika**, Reetuparna **Das**, and Scott **Mahlke**. "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism." In: *ACM SIGARCH Computer Architecture News* 45.2, June 2017, pp. 548–560. issn: 0163-5964. doi:10.1145/3140659.3080215. https://doi.org/10.1145/3140659.3080215

[255] B. **Reagen**, P. **Whatmough**, R. **Adolf**, S. **Rama**, H. **Lee**, S. K. **Lee**, J. M. **Hernández-Lobato**, G. **Wei**, and D. **Brooks**. "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators." In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. ISSN: 1063-6897. June 2016, pp. 267–278. doi:10.1109/ISCA.2016.32

[256] Manoj **Alwani**, Han **Chen**, Michael **Ferdman**, and Peter **Milder**. "Fused-layer CNN accelerators." In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12. doi:10.1109/MICRO.2016.7783725

[257] Eric **Qin**, Ananda **Samajdar**, Hyoukjun **Kwon**, Vineet **Nadella**, Sudarshan **Srinivasan**, Dipankar **Das**, Bharat **Kaul**, and Tushar **Krishna**. "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training." In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. ISSN: 2378-203X. Feb. 2020, pp. 58–70. doi:10.1109/HPCA47549.2020.00015

[258] Kartik **Hegde**, Hadi **Asghari-Moghaddam**, Michael **Pellauer**, Neal **Crago**, Aamer **Jaleel**, Edgar **Solomonik**, Joel **Emer**, and Christopher W. **Fletcher**. "ExTensor: An Accelerator for Sparse Tensor Algebra." In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 319–333. isbn:978-1-4503-6938-1. doi:10.1145/3352460.3358275. https://dl.acm.org/doi/10.1145/3352460.3358275

[259] Guowei **Zhang**, Nithya **Attaluri**, Joel S. **Emer**, and Daniel **Sanchez**. "Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication." en. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Virtual USA: ACM, Apr. 2021, pp. 687–701. isbn:978-1-4503-8317-2. doi:10.1145/3445814.3446702. https://dl.acm.org/doi/10.1145/3445814.3446702

[260] Dongyoung **Kim**, Junwhan **Ahn**, and Sungjoo **Yoo**. "ZeNA: Zero-Aware Neural Network Accelerator." In: *IEEE Design Test* 35.1, Feb. 2018. Conference Name: IEEE Design Test, pp. 39–46. issn: 2168-2364. doi:10.1109/MDAT.2017.2741463

[261] Jorge **Albericio**, Patrick **Judd**, Tayler **Hetherington**, Tor **Aamodt**, Natalie Enright **Jerger**, and Andreas **Moshovos**. "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing." In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. ISSN: 1063-6897. June 2016, pp. 1–13. doi:10.1109/ISCA.2016.11

[262] Shijin **Zhang**, Zidong **Du**, Lei **Zhang**, Huiying **Lan**, Shaoli **Liu**, Ling **Li**, Qi **Guo**, Tianshi **Chen**, and Yunji **Chen**. "Cambricon-X: An accelerator for sparse neural networks." In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12. doi:10.1109/MICRO.2016.7783723

[263] Xuda **Zhou**, Zidong **Du**, Qi **Guo**, Shaoli **Liu**, Chengsi **Liu**, Chao **Wang**, Xuehai **Zhou**, Ling **Li**, Tianshi **Chen**, and Yunji **Chen**. "Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach." In: *2018 51st Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO). Oct. 2018, pp. 15–28. doi:10.1109/MICRO.2018.00011

[264] Francisco **Muñoz-Martínez**, Raveesh **Garg**, Michael **Pellauer**, José L. **Abellán**, Manuel E. **Acacio**, and Tushar **Krishna**. "Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 252–265. isbn:978-1-4503-9918-0. doi:10.1145/3582016.3582069. https://dl.acm.org/doi/10.1145/3582016.3582069

[265] Yue **Niu**, Rajgopal **Kannan**, Ajitesh **Srivastava**, and Viktor **Prasanna**. "Reuse Kernels or Activations? A Flexible Dataflow for Low-latency Spectral CNN Acceleration." In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 266–276. isbn:978-1-4503-7099-8. doi:10.1145/3373087.3375302. https://dl.acm.org/doi/10.1145/3373087.3375302

[266] Dingqing **Yang**, Amin **Ghasemazar**, Xiaowei **Ren**, Maximilian **Golub**, Guy **Lemieux**, and Mieszko **Lis**. *Procrustes: a Dataflow and Accelerator for Sparse Deep Neural Network Training*. arXiv:2009.10976 [cs]. Sept. 2020. doi:10.48550/arXiv.2009.10976. http://arxiv.org/abs/2009.10976

[267] Chunyun **Chen**, Zhe **Wang**, Xiaowei **Chen**, Jie **Lin**, and Mohamed M. Sabry **Aly**. "Efficient Tunstall Decoder for Deep Neural Network Compression." In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. ISSN: 0738-100X. Dec. 2021, pp. 1021–1026. doi:10.1109/DAC18074.2021.9586173

[268] Frederick **Tung** and Greg **Mori**. "Deep Neural Network Compression by In-Parallel Pruning-Quantization." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.3, Mar. 2020. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 568–579. issn: 1939-3539. doi:10.1109/TPAMI.2018.2886192

[269] Haichuan **Yang**, Shupeng **Gui**, Yuhao **Zhu**, and Ji **Liu**. "Automatic Neural Network Compression by Sparsity-Quantization Joint Learning: A Constrained Optimization-Based Approach." In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). ISSN: 2575-7075. June 2020, pp. 2175–2185. doi:10.1109/CVPR42600.2020.00225

[270] YingBo **Fan**, Wei **Pang**, and ShengLi **Lu**. "HFPQ: deep neural network compression by hardware-friendly pruning-quantization." en. In: *Applied Intelligence* 51.10, Oct. 2021, pp. 7016–7028. issn: 1573-7497. doi:10.1007/s10489-020-01968-x. https://doi.org/10.1007/s10489-020-01968-x

[271] Torben **Krieger**, Bernhard **Klein**, and Holger **Fröning**. *Towards Hardware-Specific Automatic Compression of Neural Networks*. en. arXiv:2212.07818 [cs, stat]. Dec. 2022. http://arxiv.org/abs/2212.07818

[272] Rangharajan **Venkatesan** et al. "MAGNet: A Modular Accelerator Generator for Neural Networks." en. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Westminster, CO, USA: IEEE, Nov. 2019, pp. 1–8. isbn:978-1-72812-350-9. doi:10.1109/ICCAD45719.2019.8942127. https://ieeexplore.ieee.org/document/8942127/

[273] Sam Likun **Xi**, Yuan **Yao**, Kshitij **Bhardwaj**, Paul **Whatmough**, Gu-Yeon **Wei**, and David **Brooks**. "SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads." en. In: *arXiv:1912.04481 [cs]*, Dec. 2019. arXiv: 1912.04481. http://arxiv.org/abs/1912.04481

[274] Francisco **Muñoz-Martínez**, José L. **Abellán**, Manuel E. **Acacio**, and Tushar **Krishna**. "STONNE: A Detailed Architectural Simulator for Flexible Neural Network Accelerators." In: *arXiv:2006.07137 [cs, eess]*, June 2020. arXiv: 2006.07137. http://arxiv.org/abs/2006.07137

[275] Mohammad **Motamedi**, Philipp **Gysel**, Venkatesh **Akella**, and Soheil **Ghiasi**. "Design space exploration of FPGA-based Deep Convolutional Neural Networks." In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. ISSN: 2153-697X. Jan. 2016, pp. 575–580. doi:10.1109/ASPDAC.2016.7428073

[276] Mohamed S. **Abdelfattah**, Łukasz **Dudziak**, Thomas **Chau**, Royson **Lee**, Hyeji **Kim**, and Nicholas D. **Lane**. "Best of Both Worlds: AutoML Codesign of a CNN and its Hardware

Accelerator.” en. In: *arXiv:2002.05022 [cs, eess]*, Mar. 2020. arXiv: 2002.05022. http://arxiv.org/abs/2002.05022

[277] Ananda **Samajdar**, Yuhao **Zhu**, Paul **Whatmough**, Matthew **Mattina**, and Tushar **Krishna**. “SCALE-Sim: Systolic CNN Accelerator Simulator.” In: *arXiv:1811.02883 [cs]*, Feb. 2019. arXiv: 1811.02883. http://arxiv.org/abs/1811.02883

[278] Linyan **Mei**, Pouya **Houshmand**, Vikram **Jain**, Sebastian **Giraldo**, and Marian **Verhelst**. “ZigZag: A Memory-Centric Rapid DNN Accelerator Design Space Exploration Framework.” en. In: *arXiv:2007.11360 [cs]*, Aug. 2020. arXiv: 2007.11360. http://arxiv.org/abs/2007.11360

[279] Zhe **Yuan**, Yongpan **Liu**, Jinshan **Yue**, Yixiong **Yang**, Jingyu **Wang**, Xiaoyu **Feng**, Jian **Zhao**, Xueqing **Li**, and Huazhong **Yang**. “STICKER: An Energy-Efficient Multi-Sparsity Compatible Accelerator for Convolutional Neural Networks in 65-nm CMOS.” in: *IEEE Journal of Solid-State Circuits* 55.2, Feb. 2020. Conference Name: IEEE Journal of Solid-State Circuits, pp. 465–477. ISSN: 1558-173X. doi:10.1109/JSSC.2019.2946771

[280] Jie-Fang **Zhang**, Ching-En **Lee**, Chester **Liu**, Yakun Sophia **Shao**, Stephen W. **Keckler**, and Zhengya **Zhang**. “SNAP: An Efficient Sparse Neural Acceleration Processor for Unstructured Sparse Deep Neural Network Inference.” In: *IEEE Journal of Solid-State Circuits* 56.2, Feb. 2021. Conference Name: IEEE Journal of Solid-State Circuits, pp. 636–647. ISSN: 1558-173X. doi:10.1109/JSSC.2020.3043870

[281] Kuan **Wang**, Zhijian **Liu**, Yujun **Lin**, Ji **Lin**, and Song **Han**. “HAQ: Hardware-Aware Automated Quantization With Mixed Precision.” en. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, June 2019, pp. 8604–8612. isbn:978-1-72813-293-8. doi:10.1109/CVPR.2019.00881. https://ieeexplore.ieee.org/document/8954415/

[282] Nael **Fasfous** et al. “AnaCoNGA: Analytical HW-CNN Co-Design Using Nested Genetic Algorithms.” In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. ISSN: 1558-1101. Mar. 2022, pp. 238–243. doi:10.23919/DATE54114.2022.9774574

[283] Sungju **Ryu**, Hyungjun **Kim**, Wooseok **Yi**, Eunhwan **Kim**, Yulhwa **Kim**, Taesu **Kim**, and Jae-Joon **Kim**. “BitBlade: Energy-Efficient Variable Bit-Precision Hardware Accelerator for Quantized Neural Networks.” In: *IEEE Journal of Solid-State Circuits* 57.6, June 2022. Conference Name: IEEE Journal of Solid-State Circuits, pp. 1924–1935. ISSN: 1558-173X. doi:10.1109/JSSC.2022.3141050

[284] Kai **Huang**, Bowen **Li**, Dongliang **Xiong**, Haitian **Jiang**, Xiaowen **Jiang**, Xiaolang **Yan**, Luc **Claesen**, Dehong **Liu**, Junjian **Chen**, and Zhili **Liu**. “Structured Dynamic Precision for Deep Neural Networks Quantization.” en. In: *ACM Transactions on Design Automation of Electronic Systems* 28.1, Jan. 2023, pp. 1–24. ISSN: 1084-4309, 1557-7309. doi:10.1145/3549535. https://dl.acm.org/doi/10.1145/3549535

[285] Arie Wahyu **Wijayanto**, Jun Jin **Choong**, Kaushalya **Madhawa**, and Tsuyoshi **Murata**. “Towards Robust Compressed Convolutional Neural Networks.” In: *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*. ISSN: 2375-9356. Feb. 2019, pp. 1–8. doi:10.1109/BIGCOMP.2019.8679132

[286] Faiq **Khalid**, Hassan **Ali**, Hammad **Tariq**, Muhammad Abdullah **Hanif**, Semeen **Rehman**, Rehan **Ahmed**, and Muhammad **Shafique**. “QuSecNets: Quantization-based Defense Mechanism for Securing Deep Neural Network against Adversarial Attacks.” In: *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. ISSN: 1942-9401. July 2019, pp. 182–187. doi:10.1109/IOLTS.2019.8854377

[287] Mingxin **Zhao**, Ke **Ning**, Shuangming **Yu**, Liyuan **Liu**, and Nanjian **Wu**. “Quantizing Oriented Object Detection Network via Outlier-Aware Quantization and IoU Approximation.” In: *IEEE Signal Processing Letters* 27, 2020. Conference Name: IEEE Signal Processing Letters, pp. 1914–1918. ISSN: 1558-2361. doi:10.1109/LSP.2020.3031490

[288] Yiyang **Shen**, Yongzhen **Wang**, Mingqiang **Wei**, Honghua **Chen**, Haoran **Xie**, Gary **Cheng**, and Fu Lee **Wang**. “Semi-MoreGAN: A New Semi-supervised Generative Adversarial Network for Mixture of Rain Removal.” en. In: *Computer Graphics Forum* 41.7, Oct. 2022. arXiv:2204.13420 [cs], pp. 443–454. ISSN: 0167-7055, 1467-8659. doi:10.1111/cgf.14690. http://arxiv.org/abs/2204.13420

[289] Tianyu **Wang**, Xin **Yang**, Ke **Xu**, Shaozhe **Chen**, Qiang **Zhang**, and Rynson W.H. **Lau**. “Spatial Attentive Single-Image Deraining With a High Quality Real Rain Dataset.” In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. Conference Name: 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) ISBN: 9781728132938 Place: Long Beach, CA, USA Publisher: IEEE, pp. 12262–12271. doi:10.1109/CVPR.2019.01255. https://ieeexplore.ieee.org/document/8953571/

[290] Lei **Yang**, Weiwen **Jiang**, Weichen **Liu**, Edwin H. M. **Sha**, Yiyu **Shi**, and Jingtong **Hu**. “Co-Exploring Neural Architecture and Network-on-Chip Design for Real-Time Artificial Intelligence.” In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. ISSN: 2153-697X. Jan. 2020, pp. 85–90. doi:10.1109/ASP-DAC47756.2020.9045595

[291] Lei **Yang**, Zheyu **Yan**, Meng **Li**, Hyoukjun **Kwon**, Liangzhen **Lai**, Tushar **Krishna**, Vikas **Chandra**, Weiwen **Jiang**, and Yiyu **Shi**. “Co-Exploration of Neural Architectures and Heterogeneous ASIC Accelerator Designs Targeting Multiple Tasks.” en. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. San Francisco, CA, USA: IEEE, July 2020, pp. 1–6. isbn:978-1-72811-085-1. doi:10.1109/DAC18072.2020.9218676. https://ieeexplore.ieee.org/document/9218676/

[292] Xuefei **Ning**, Changcheng **Tang**, Wenshuo **Li**, Songyi **Yang**, Tianchen **Zhao**, Niansong **Zhang**, Tianyi **Lu**, Shuang **Liang**, Huazhong **Yang**, and Yu **Wang**. “aw_nas: A Modularized and Extensible NAS framework.” en. In: *arXiv:2012.10388 [cs]*, Nov. 2020. arXiv: 2012.10388. http://arxiv.org/abs/2012.10388

[293] Jason **Cong**, Zhenman **Fang**, Michael **Gill**, and Glenn **Reinman**. “PARADE: A cycle-accurate full-system simulation Platform for Accelerator-Rich Architectural Design and Exploration.” In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2015, pp. 380–387. doi:10.1109/ICCAD.2015.7372595. https://ieeexplore.ieee.org/document/7372595

[294] Mijing **Sun**, Li **Xu**, Zhenmin **Li**, Wei **Ni**, Gaoming **Du**, Xiaolei **Wang**, and Yongsheng **Yin**. “Simulators for Deep Neural Network Accelerator Design and Analysis: A Brief Review.” In: *2021 IEEE 15th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. ISSN: 2163-5056. Oct. 2021, pp. 36–42. doi:10.1109/ASID52932.2021.9651675. https://ieeexplore.ieee.org/document/9651675

[295] Nathan **Binkert** et al. “The gem5 simulator.” In: *ACM SIGARCH Computer Architecture News* 39.2, Aug. 2011, pp. 1–7. issn: 0163-5964. doi:10.1145/2024716.2024718. https://dl.acm.org/doi/10.1145/2024716.2024718

[296] Y. **Chen**, J. **Emer**, and V. **Sze**. “Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks.” In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. ISSN: 1063-6897. June 2016, pp. 367–379. doi:10.1109/ISCA.2016.40

[297] Xuan **Yang** et al. “Interstellar: Using Halide’s Scheduling Language to Analyze DNN Accelerators.” en. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. arXiv:1809.04070 [cs]. Mar. 2020, pp. 369–383. doi:10.1145/3373376.3378514. http://arxiv.org/abs/1809.04070

[298] Cong **Hao**, Yao **Chen**, Xiaofan **Zhang**, Yuhong **Li**, Jinjun **Xiong**, Wen-mei **Hwu**, and Deming **Chen**. “Effective Algorithm-Accelerator Co-design for AI Solutions on Edge Devices.” en. In: *arXiv:2010.07185 [cs]*, Oct. 2020. arXiv: 2010.07185. http://arxiv.org/abs/2010.07185

[299] Zhen **Dong**, Yizhao **Gao**, Qijing **Huang**, John **Wawrzynek**, Hayden K.H. **So**, and Kurt **Keutzer**. “HAO: Hardware-aware Neural Architecture Optimization for Efficient Inference.” en. In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Orlando, FL, USA: IEEE, May 2021, pp. 50–59. isbn:978-1-66543-555-0. doi:10.1109/FCCM51124.2021.00014. https://ieeexplore.ieee.org/document/9444059/

[300] *CHaiDNN: HLS based Deep Neural Network Accelerator Library for Xilinx Ultrascale+ MPSoCs*. https://github.com/Xilinx/CHaiDNN. Accessed: 2023-10-11

[301] *Imperas OVPsim*. https://ovpworld.org. Accessed: 2021-04-05

[302] Joseph **Redmon**. *Darknet: Open Source Neural Networks in C*. http://pjreddie.com/darknet/. 2013

[303] *OSCI TLM-2.0 standard*. http://systemc.org. Accessed: 2021-04-07

[304] Wilson **Snyder**. "Verilator and Systemperl." In: *North American SystemC Users' Group Design Automation Conference*, 2004

[305] J. **Blank** and K. **Deb**. "pymoo: Multi-Objective Optimization in Python." In: *IEEE Access* 8, 2020, pp. 89497–89509

[306] Krste **Asanović** et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[307] *Tiny Darknet*. https://pjreddie.com/darknet/tiny-darknet/. Accessed: 2021-04-12

[308] K. **Deb**, A. **Pratap**, S. **Agarwal**, and T. **Meyarivan**. "A fast and elitist multiobjective genetic algorithm: NSGA-II." in: *IEEE Transactions on Evolutionary Computation* 6.2, Apr. 2002. Conference Name: IEEE Transactions on Evolutionary Computation, pp. 182–197. issn: 1941-0026. doi:10.1109/4235.996017

[309] Fahimeh **Yazdanpanah**, Carlos **Álvarez**, Daniel **Jiménez-González**, Rosa M. **Badia**, and Mateo **Valero**. "Picos: A hardware runtime architecture support for OmpSs." In: *Future Generation Computer Systems* 53, Dec. 2015, pp. 130–139. issn: 0167-739X. doi:10.1016/j.future.2014.12.010. https://www.sciencedirect.com/science/article/pii/S0167739X14002702

[310] Ting-Wu **Chin**, Pierce I.-Jen **Chuang**, Vikas **Chandra**, and Diana **Marculescu**. "One Weight Bitwidth to Rule Them All." en. In: *arXiv:2008.09916 [cs, eess]*, Aug. 2020. arXiv: 2008.09916. http://arxiv.org/abs/2008.09916

[311] Dong **Yi**, Zhen **Lei**, Shengcai **Liao**, and Stan Z. **Li**. "Learning Face Representation from Scratch." In: *arXiv:1411.7923 [cs]*, Nov. 2014. arXiv: 1411.7923. http://arxiv.org/abs/1411.7923

[312] Nanfei **Jiang**, Xu **Zhao**, Chaoyang **Zhao**, Yongqi **An**, Ming **Tang**, and Jinqiao **Wang**. *Pruning-aware Sparse Regularization for Network Pruning*. Tech. rep. arXiv:2201.06776. arXiv:2201.06776 [cs] type: article. arXiv, Jan. 2022. doi:10.48550/arXiv.2201.06776. http://arxiv.org/abs/2201.06776

[313] Mark **Kurtz** et al. "Inducing and exploiting activation sparsity for fast neural network inference." In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119. ICML'20. JMLR.org, July 2020, pp. 5533–5543

[314] Maohua **Zhu** and Yuan **Xie**. "Taming Unstructured Sparsity on GPUs via Latency-Aware Optimization." In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. ISSN: 0738-100X. July 2020, pp. 1–6. doi:10.1109/DAC18072.2020.9218644

[315] Sharan **Narang**, Eric **Undersander**, and Gregory **Diamos**. "Block-Sparse Recurrent Neural Networks." In: *arXiv:1711.02782 [cs, stat]*, Nov. 2017. arXiv: 1711.02782. http://arxiv.org/abs/1711.02782

[316] Baoyuan **Liu**, Min **Wang**, Hassan **Foroosh**, Marshall **Tappen**, and Marianna **Penksy**. "Sparse Convolutional Neural Networks." In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. ISSN: 1063-6919. June 2015, pp. 806–814. doi:10.1109/CVPR.2015.7298681

[317] Mingbao **Lin**, Yuxin **Zhang**, Yuchao **Li**, Bohong **Chen**, Fei **Chao**, Mengdi **Wang**, Shen **Li**, Yonghong **Tian**, and Rongrong **Ji**. *1xN Pattern for Pruning Convolutional Neural Networks*. arXiv:2105.14713 [cs]. Oct. 2022. doi:10.48550/arXiv.2105.14713. http://arxiv.org/abs/2105.14713

[318] Aojun **Zhou**, Yukun **Ma**, Junnan **Zhu**, Jianbo **Liu**, Zhijie **Zhang**, Kun **Yuan**, Wenxiu **Sun**, and Hongsheng **Li**. *Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch*. Tech. rep. arXiv:2102.04010. arXiv:2102.04010 [cs] type: article. arXiv, Apr. 2021. doi:10.48550/arXiv.2102.04010. http://arxiv.org/abs/2102.04010

[319] Yu **Ji**, Ling **Liang**, Lei **Deng**, Youyang **Zhang**, Youhui **Zhang**, and Yuan **Xie**. "TETRIS: TilE-matching the TRemendous Irregular Sparsity." In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc., 2018. https://papers.nips.cc/paper_files/paper/2018/hash/89885ff2c83a10305ee08bd507c1049c-Abstract.html

[320] Stephen P. **Boyd** and Lieven **Vandenberghe**. *Convex optimization*. en. Cambridge, UK ; New York: Cambridge University Press, 2004. isbn:978-0-521-83378-3

[321] John Tan Chong **Min** and Mehul **Motani**. *DropNet: Reducing Neural Network Complexity via Iterative Pruning*. en. arXiv:2207.06646 [cs]. July 2022. doi:10.5555/3524938.3525805. http://arxiv.org/abs/2207.06646

[322] A. **Avizienis**, J.-C. **Laprie**, B. **Randell**, and C. **Landwehr**. "Basic concepts and taxonomy of dependable and secure computing." en. In: *IEEE Transactions on Dependable and Secure Computing* 1.1, Jan. 2004, pp. 11–33. ISSN: 1545-5971. doi:10.1109/TDSC.2004.2. http://ieeexplore.ieee.org/document/1335465/

[323] *ISO 26262-1:2018(en), Road vehicles — Functional safety*. International Standard 26262. 2018

[324] *Road vehicles — Safety of the intended functionality*. International Standard 21448. 2022

[325] Joel **Janai**, Fatma **Güney**, Aseem **Behl**, and Andreas **Geiger**. *Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art*. arXiv:1704.05519 [cs]. Mar. 2021. doi:10.48550/arXiv.1704.05519. http://arxiv.org/abs/1704.05519

[326] Christoph **Kamann** and Carsten **Rother**. "Benchmarking the Robustness of Semantic Segmentation Models with Respect to Common Corruptions." en. In: *International Journal of Computer Vision* 129.2, Feb. 2021, pp. 462–483. ISSN: 1573-1405. doi:10.1007/s11263-020-01383-2. https://doi.org/10.1007/s11263-020-01383-2

[327] Dan **Hendrycks** and Thomas **Dietterich**. *Benchmarking Neural Network Robustness to Common Corruptions and Perturbations*. en. arXiv:1903.12261 [cs, stat]. Mar. 2019. http://arxiv.org/abs/1903.12261

[328] Kaiyuan **Guo**, Shulin **Zeng**, Jincheng **Yu**, Yu **Wang**, and Huazhong **Yang**. "A Survey of FPGA-Based Neural Network Accelerator." In: *arXiv:1712.08934 [cs]*, Dec. 2018. arXiv: 1712.08934. http://arxiv.org/abs/1712.08934

[329] *Versal™ Prime Series Product Selection Guide*. https://www.xilinx.com/support/documents/selection-guides/versal-prime-product-selection-guide.pdf. Accessed: 2023-10-30

[330] Hyunjoon **Kim**, Qian **Chen**, Taegeun **Yoo**, Tony Tae-Hyoung **Kim**, and Bongjin **Kim**. "A 1-16b Precision Reconfigurable Digital In-Memory Computing Macro Featuring Column-MAC Architecture and Bit-Serial Computation." In: *ESSCIRC 2019 - IEEE 45th European Solid State Circuits Conference (ESSCIRC)*. ISSN: 2643-1319. Sept. 2019, pp. 345–348. doi:10.1109/ESSCIRC.2019.8902824

[331] Alok **Parmar**, Kailash **Prasad**, Nanditha **Rao**, and Joycee **Mekie**. "An Automated Approach to Compare Bit Serial and Bit Parallel In-Memory Computing for DNNs." In: *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. ISSN: 2158-1525. May 2022, pp. 2948–2952. doi:10.1109/ISCAS48785.2022.9937586

[332] Bert **Moons**, Roel **Uytterhoeven**, Wim **Dehaene**, and Marian **Verhelst**. "14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI." in: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. ISSN: 2376-8606. Feb. 2017, pp. 246–247. doi:10.1109/ISSCC.2017.7870353

[333] Hardik **Sharma**, Jongse **Park**, Naveen **Suda**, Liangzhen **Lai**, Benson **Chau**, Joon Kyung **Kim**, Vikas **Chandra**, and Hadi **Esmaeilzadeh**. "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network." en. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. Los Angeles, CA, USA: IEEE, June 2018, pp. 764–775. isbn:978-1-5386-5984-7. doi:10.1109/ISCA.2018.00069. https://ieeexplore.ieee.org/document/8416871/

[334] Ahmed T. **Elthakeb**, Prannoy **Pilligundla**, FatemehSadat **Mireshghallah**, Amir **Yazdanbakhsh**, and Hadi **Esmaeilzadeh**. *ReLeQ: A Reinforcement Learning Approach for Deep Quantization of Neural Networks*. arXiv:1811.01704 [cs, stat]. Apr. 2020. doi:10.48550/arXiv.1811.01704. http://arxiv.org/abs/1811.01704

[335] Bichen **Wu**, Yanghan **Wang**, Peizhao **Zhang**, Yuandong **Tian**, Peter **Vajda**, and Kurt **Keutzer**. *Mixed Precision Quantization of ConvNets via Differentiable Neural Architecture Search*. arXiv:1812.00090 [cs]. Nov. 2018. doi:10.48550/arXiv.1812.00090. http://arxiv.org/abs/1812.00090

[336] Weihan **Chen**, Peisong **Wang**, and Jian **Cheng**. *Towards Mixed-Precision Quantization of Neural Networks via Constrained Optimization*. en. arXiv:2110.06554 [cs]. Oct. 2021. http://arxiv.org/abs/2110.06554

[337] Mengshu **Sun**, Zhengang **Li**, Alec **Lu**, Yanyu **Li**, Sung-En **Chang**, Xiaolong **Ma**, Xue **Lin**, and Zhenman **Fang**. "FILM-QNN: Efficient FPGA Acceleration of Deep Neural Networks

with Intra-Layer, Mixed-Precision Quantization." en. In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Virtual Event USA: ACM, Feb. 2022, pp. 134–145. isbn:978-1-4503-9149-8. doi:10.1145/3490422.3502364. https://dl.acm.org/doi/10.1145/3490422.3502364

[338] Iraj **Moghaddasi**, Saeid **Gorgin**, and Jeong-A. **Lee**. "Dependable DNN Accelerator for Safety-critical Systems: A Review on the Aging Perspective." In: *IEEE Access*, 2023. Conference Name: IEEE Access, pp. 1–1. issn: 2169-3536. doi:10.1109/ACCESS.2023.3300376

[339] Oliver **Zendel**, Markus **Murschitz**, Martin **Humenberger**, and Wolfgang **Herzner**. "How Good Is My Test Data? Introducing Safety Analysis for Computer Vision." en. In: *International Journal of Computer Vision* 125.1, Dec. 2017, pp. 95–109. issn: 1573-1405. doi:10.1007/s11263-017-1020-z. https://doi.org/10.1007/s11263-017-1020-z

[340] Xiaowei **Huang**, Marta **Kwiatkowska**, Sen **Wang**, and Min **Wu**. *Safety Verification of Deep Neural Networks*. arXiv:1610.06940 [cs, stat]. May 2017. doi:10.48550/arXiv.1610.06940. http://arxiv.org/abs/1610.06940

[341] Robert **Geirhos**, Carlos R. Medina **Temme**, Jonas **Rauber**, Heiko H. **Schütt**, Matthias **Bethge**, and Felix A. **Wichmann**. *Generalisation in humans and deep neural networks*. en. arXiv:1808.08750 [cs, q-bio, stat]. Oct. 2020. http://arxiv.org/abs/1808.08750

[342] Igor **Vasiljevic**, Ayan **Chakrabarti**, and Gregory **Shakhnarovich**. *Examining the Impact of Blur on Recognition by Convolutional Networks*. arXiv:1611.05760 [cs]. May 2017. doi:10.48550/arXiv.1611.05760. http://arxiv.org/abs/1611.05760

[343] Hong **Wang**, Yichen **Wu**, Minghan **Li**, Qian **Zhao**, and Deyu **Meng**. "A Survey on Rain Removal from Video and Single Image." en. In: *Science China Information Sciences* 65.1, Jan. 2022. arXiv:1909.08326 [cs, eess], p. 111101. issn: 1674-733X, 1869-1919. doi:10.1007/s11432-020-3225-9. http://arxiv.org/abs/1909.08326

[344] Xueyang **Fu**, Jiabin **Huang**, Delu **Zeng**, Yue **Huang**, Xinghao **Ding**, and John **Paisley**. "Removing Rain from Single Images via a Deep Detail Network." en. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, July 2017, pp. 1715–1723. isbn:978-1-5386-0457-1. doi:10.1109/CVPR.2017.186. http://ieeexplore.ieee.org/document/8099669/

[345] Xiaowei **Hu**, Chi-Wing **Fu**, Lei **Zhu**, and Pheng-Ann **Heng**. "Depth-Attentional Features for Single-Image Rain Removal." In: 2019, pp. 8022–8031. https://openaccess.thecvf.com/content_CVPR_2019/html/Hu_Depth-Attentional_Features_for_Single-Image_Rain_Removal_CVPR_2019_paper.html

[346] Claudio **Michaelis**, Benjamin **Mitzkus**, Robert **Geirhos**, Evgenia **Rusak**, Oliver **Bringmann**, Alexander S. **Ecker**, Matthias **Bethge**, and Wieland **Brendel**. "Benchmarking Robustness in Object Detection: Autonomous Driving when Winter is Coming." In: *arXiv preprint arXiv:1907.07484*, 2019

Publications

[Hoe+24] Julian **Hoefer**, Fabian **Kreß**, Tim **Hotfilter**, Matthias **Stammler**, Tanja **Harbaum**, and Jürgen **Becker**. "BayWatch: Leveraging Bayesian Neural Networks for Hardware Fault Tolerance and Monitoring." In: *37th International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. In press. 2024

[Kre+24a] Fabian **Kreß**, El Mahdi **El Annabi**, Tim **Hotfilter**, Julian **Hoefer**, Tanja **Harbaum**, and Jürgen **Becker**. "Automated Deep Neural Network Inference Partitioning for Distributed Embedded Systems." In: *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. In press. 2024, pp. 1–6

[Kre+24b] Fabian **Kreß**, Alexey **Serdyuk**, Denis **Kobsar**, Tim **Hotfilter**, Julian **Hoefer**, Tanja **Harbaum**, and Jürgen **Becker**. "LOTTA: An FPGA-based Low-Power Temporal Convolutional Network Hardware Accelerator." In: *2024 IEEE 37th International System-on-Chip Conference (SOCC)*. In press. 2024, pp. 1–6

[Chu+23] Anqi **Chu** et al. "LETSCOPE: Lifecycle Extensions Through Software-Defined Predictive Control of Power Electronics." In: *IEEE EUROCON 2023 - 20th International Conference on Smart Technologies*. July 2023, pp. 665–670. doi:10.1109/EUROCON56442.2023.10199076

[Hoe+23a] Julian **Hoefer**, Tim **Hotfilter**, Fabian **Kreß**, Chen **Qiu**, Tanja **Harbaum**, and Juergen **Becker**. "A Hardware-Aware Sampling Parameter Search for Efficient Probabilistic Object Detection." en. In: *Computer Vision Systems*. Ed. by Henrik I. **Christensen**, Peter **Corke**, Renaud **Detry**, Jean-Baptiste **Weibel**, and Markus **Vincze**. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 299–309. isbn:978-3-031-44137-0. doi:10.1007/978-3-031-44137-0_25

[Hoe+23b] Julian **Hoefer**, Fabian **Kempf**, Tim **Hotfilter**, Fabian **Kreß**, Tanja **Harbaum**, and Jürgen **Becker**. "SiFI-AI: A Fast and Flexible RTL Fault Simulation Framework Tailored for AI Models and Accelerators." In: *Proceedings of the Great Lakes Symposium on VLSI 2023*. GLSVLSI '23. New York, NY, USA: Association for Computing Machinery, May 2023, pp. 287–292. doi:10.1145/3583781.3590226. https://dl.acm.org/doi/10.1145/3583781.3590226

[Hof+23] D. **Hofman**, M. **Brcic**, M. **Kovac**, T. **Hotfilter**, J. **Becker**, D. **Reinhardt**, S. M. **Grigorescu**, R. **Stevens**, and T. T. **Vo**. "European Processor Initiative Demonstration of Integrated Semi-Autonomous Driving System." In: *2023 IEEE 36th International System-on-Chip Conference (SOCC)*. ISSN: 2164-1706. Sept. 2023, pp. 1–6. doi:10.1109/SOCC58585.2023.10257105. https://ieeexplore.ieee.org/document/10257105

[Hot+23a] Tim **Hotfilter**, Julian **Hoefer**, Fabian **Kreß**, Fabian **Kempf**, Leonhard **Kraft**, Tanja **Harbaum**, and Jürgen **Becker**. "A Hardware-Centric Approach to Increase and Prune Regular Activation Sparsity in CNNs." In: *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. ISSN: 2834-9857. June 2023, pp. 1–5. doi:10.1109/AICAS57966.2023.10168566

[Hot+23b] Tim **Hotfilter**, Julian **Hoefer**, Philipp **Merz**, Fabian **Kreß**, Fabian **Kempf**, Tanja **Harbaum**, and Jürgen **Becker**. "Leveraging Mixed-Precision CNN Inference for Increased Robustness and Energy Efficiency." In: *2023 IEEE 36th International System-on-Chip Conference (SOCC)*. ISSN: 2164-1706. Sept. 2023, pp. 1–6. doi:10.1109/SOCC58585.2023.10256738. https://ieeexplore.ieee.org/document/10256738

[Hot+23c] Tim **Hotfilter**, Patrick **Schmidt**, Julian **Höfer**, Fabian **Kreß**, Tanja **Harbaum**, and Juergen **Becker**. "An Analytical Model of Configurable Systolic Arrays to find the Best-Fitting Accelerator for a given DNN Workload." In: *Proceedings of the DroneSE and RAPIDO:*

System Engineering for constrained embedded systems. RAPIDO '23. New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 73–78. doi:10.1145/3579170.3579258. https://dl.acm.org/doi/10.1145/3579170.3579258

[Kem+23] Fabian **Kempf**, Julian **Hoefer**, Tim **Hotfilter**, and Juergen **Becker**. "A Low-Stall Methodology for an Interleaved Processor State Replication." In: *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. ISSN: 2771-3075. Dec. 2023, pp. 276–283. doi:10.1109/MCSoC60832.2023.00048. https://ieeexplore.ieee.org/document/10387795

[Kre+23a] Fabian **Kreß**, Julian **Hoefer**, Tim **Hotfilter**, Iris **Walter**, El Mahdi **El Annabi**, Tanja **Harbaum**, and Jürgen **Becker**. "Automated Search for Deep Neural Network Inference Partitioning on Embedded FPGA." en. In: *Machine Learning and Principles and Practice of Knowledge Discovery in Databases*. Ed. by Irena **Koprinska** et al. Communications in Computer and Information Science. Cham: Springer Nature Switzerland, 2023, pp. 557–568. isbn:978-3-031-23618-1. doi:10.1007/978-3-031-23618-1_37

[Kre+23b] Fabian **Kreß**, Vladimir **Sidorenko**, Patrick **Schmidt**, Julian **Hoefer**, Tim **Hotfilter**, Iris **Walter**, Tanja **Harbaum**, and Jürgen **Becker**. "CNNParted: An open source framework for efficient Convolutional Neural Network inference partitioning in embedded systems." en. In: *Computer Networks* 229, May 2023, p. 109759. ISSN: 1389-1286. doi:10.1016/j.comnet.2023.109759. https://www.sciencedirect.com/science/article/pii/S1389128623002049

[Kre+23c] Fabian **Kreß** et al. "ATLAS: An Approximate Time-Series LSTM Accelerator for Low-Power IoT Applications." In: *2023 26th Euromicro Conference on Digital System Design (DSD)*. ISSN: 2771-2508. Sept. 2023, pp. 569–576. doi:10.1109/DSD60849.2023.00084. https://ieeexplore.ieee.org/document/10456796

[Sta+23] Matthias **Stammler**, Julian **Hoefer**, David **Kraus**, Patrick **Schmidt**, Tim **Hotfilter**, Tanja **Harbaum**, and Jürgen **Becker**. "EFFECT: An End-to-End Framework for Evaluating Strategies for Parallel AI Anomaly Detection." In: *Procedia Computer Science*. International Neural Network Society Workshop on Deep Learning Innovations and Applications (INNS DLIA 2023) 222, Jan. 2023, pp. 499–508. ISSN: 1877-0509. doi:10.1016/j.procs.2023.08.188. https://www.sciencedirect.com/science/article/pii/S1877050923009535

[Hot+22a] Tim **Hotfilter**, Fabian **Kreß**, Fabian **Kempf**, Jürgen **Becker**, and Imen **Baili**. "Data Movement Reduction for DNN Accelerators: Enabling Dynamic Quantization Through an eFPGA." in: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. ISSN: 2159-3477. June 2022, pp. 371–372. doi:10.1109/ISVLSI54635.2022.00082

[Hot+22b] Tim **Hotfilter**, Fabian **Kreß**, Fabian **Kempf**, Jürgen **Becker**, Juan Miguel **De Haro**, Daniel **Jiménez-González**, Miquel **Moretó**, Carlos **Álvarez**, Jesús **Labarta**, and Imen **Baili**. "Towards Reconfigurable Accelerators in HPC: Designing a Multipurpose eFPGA Tile for Heterogeneous SoCs." In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. ISSN: 1558-1101. Feb. 2022, pp. 628–631. doi:10.23919/DATE54114.2022.9774716

[Kem+22] Fabian **Kempf**, Julian **Hoefer**, Fabian **Kreß**, Tim **Hotfilter**, Tanja **Harbaum**, and Juergen **Becker**. "Runtime Adaptive Cache Checkpointing for RISC Multi-Core Processors." In: *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. ISSN: 2164-1706. Aug. 2022, pp. 1–6. doi:10.1109/SOCC56010.2022.9908110

[Kre+22a] Fabian **Kreß**, Julian **Hoefer**, Tim **Hotfilter**, Iris **Walter**, Vladimir **Sidorenko**, Tanja **Harbaum**, and Jürgen **Becker**. "Hardware-aware Partitioning of Convolutional Neural Network Inference for Embedded AI Applications." In: *2022 18th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. ISSN: 2325-2944. Apr. 2022, pp. 133–140. doi:10.1109/DCOSS54816.2022.00034

[Kre+22b] Fabian **Kreß**, Alexey **Serdyuk**, Tim **Hotfilter**, Julian **Hoefer**, Tanja **Harbaum**, Jürgen **Becker**, and Tim **Hamann**. "Hardware-aware Workload Distribution for AI-based Online Handwriting Recognition in a Sensor Pen." In: *2022 11th Mediterranean*

Conference on Embedded Computing (*MECO*). ISSN: 2637-9511. May 2022, pp. 1–4. doi:10.1109/MECO55406.2022.9797131

[Hot+21] Tim **Hotfilter**, Julian **Hoefer**, Fabian **Kreß**, Fabian **Kempf**, and Juergen **Becker**. "FLECSim-SoC: A Flexible End-to-End Co-Design Simulation Framework for System on Chips." In: *2021 IEEE 34th International System-on-Chip Conference* (*SOCC*). ISSN: 2164-1706. Sept. 2021, pp. 83–88. doi:10.1109/SOCC52499.2021.9739212

[Wal+21] Iris **Walter**, Jonas **Ney**, Tim **Hotfilter**, Vladimir **Rybalkin**, Julian **Hoefer**, Norbert **Wehn**, and Jürgen **Becker**. "Embedded Face Recognition for Personalized Services in the Assistive Robotics." en. In: *Machine Learning and Principles and Practice of Knowledge Discovery in Databases*. Ed. by Michael **Kamp** et al. Communications in Computer and Information Science. Cham: Springer International Publishing, 2021, pp. 339–350. isbn:978-3-030-93736-2. doi:10.1007/978-3-030-93736-2_26

[Hot+20] Tim **Hotfilter**, Fabian **Kempf**, Jürgen **Becker**, Dominik **Reinhardt**, and Imen **Baili**. "Embedded Image Processing the European Way: A new platform for the future automotive market." In: *2020 IEEE 6th World Forum on Internet of Things* (*WF-IoT*). May 2020, pp. 1–6. doi:10.1109/WF-IoT48130.2020.9221396

[Sto+20] Simon Claus **Stock**, Jürgen **Becker**, Daniel **Grimm**, Tim **Hotfilter**, Gabriela **Molinar**, Marco **Stang**, and Wilhelm **Stork**. "QUA^{3}CK - A Machine Learning Development Process." en. In: *Proceedings of Artificial Intelligence for Science, Industry and Society — PoS*(*AISIS2019*). Universidad Nacional Autónoma de México, Mexico City, México: Sissa Medialab, July 2020, p. 026. doi:10.22323/1.372.0026. https://pos.sissa.it/372/026

[Yig+19] Luo **Yigui**, Sun **Youteng**, Florian **Schade**, Tim **Hotfilter**, Jürgen **Becker**, Zhu **Yuan**, Ouyang **Zizhou**, and Liang **Weiming**. "Evaluation of a high-throughput communication link for future automotive ADAS controllers." en. In: *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering* 233.9, July 2019. Publisher: IMECHE, pp. 2371–2378. issn: 0954-4070. doi:10.1177/0954407019851334. https://doi.org/10.1177/0954407019851334

List of Figures

1.1 Development of parameter count and FLOPs of recent CNNs 2

2.1 Overview of the types of machine learning 11
2.2 Components of an artificial neuron 12
2.3 Commonly used activation functions for DNNs 13
2.4 Example of a Multilayer Perceptron 14
2.5 Principle of a convolution operation 21
2.6 CNN layer parameters and nomenclature 22
2.7 Basic principle of the pooling operation 24
2.8 Typical tasks of CNNs. 25
2.9 Samples of ImageNet and CIFAR-10 28
2.10 Overview of ResNet 33
2.11 Overview of SqueezeNet 34
2.12 Overview of U-Nets 37
2.13 Development of recent microprocessors 39
2.14 Simplified overview of a CPU 41
2.15 Performance development of recent GPUs 42
2.16 Overview of an FPGA 44
2.17 Overview of different computation devices. 45
2.18 Principle of im2col and col2im transformations 49
2.19 Simple MAC unit with its inputs and outputs 50
2.20 Overview of the Roofline model 51
2.21 Energy consumption of common operations 52
2.22 Typical memory hierarchy of SoCs with DNN accelerators 53
2.23 Overview of a systolic array with its processing elements 56
2.24 Different types of dataflows in DNN accelerators 58

3.1 Overview of recent DNN accelerators 60
3.2 Block diagram of NVDLA 63
3.3 DNN Mapping process 67
3.4 Principle of a DNN mapper 68

3.5 Overview of Accelergy 72
3.6 Examples of different number representations 74
3.7 Steps of DNN quantization 76
3.8 Clipping calibration example for quantization of ResNet-50 77
3.9 Impact of quantization on ResNet-50's accuracy 78
3.10 Histogram of all weights in a pretrained ResNet-50 81
3.11 Basic principle of weight pruning 82
3.12 Histogram of ImageNet-1K input feature maps in a ResNet-50 83
3.13 Basic principle of magnitude activation pruning 84
3.14 Steps of Design Space Exploration for DNN accelerators 87

4.1 Basic concept of our simulator for algorithm accelerator co-design . . 95
4.2 Overview of STONNE 99
4.3 Overview of MAESTRO 101
4.4 Overview of FLECSim 104
4.5 SystemC model of FLECSim 105
4.6 Overview of the implementation of FLECSim 111
4.7 Test infrastructure of FLECSim 113
4.8 Algorithm Accelerator Co-Design implemented with FLECSim . . . 115
4.9 Overview if our DNN accelerator evaluation tool flow 116
4.10 Overview of our analytical model. 117
4.11 Overview of the Gemmini systolic array generator 124
4.12 Detail view of Gemmini's systolic array structure 125
4.13 Result space of a configurable systolic array 129
4.14 Design space results generated by FLECSim 131
4.15 Design space of LeNet-5 using an NSGA-II algorithm 132
4.16 Evaluation of different layer configurations across different array sizes 133
4.17 ResNet-34 cycle estimation for different memory configurations . . . 135
4.18 Design space evaluation of ResNet-34 137
4.19 Overview of the CNN accelerator architecture 139
4.20 Overview of our eFPGA architecture exploration design flow 141
4.21 Architecture layout of the found eFPGA configuration 143

5.1 Distribution of output feature map values in ResNet-50 149
5.2 Visualization of the output of the first two ResNet-50 layers 150
5.3 Interaction of Spex and Sparse-Blox 151
5.4 Principle of our blockwise regular pruning approach 152

5.5 PEs in SCNN . . . 155
5.6 High-level overview of STICKER . . . 157
5.7 System architecture of SNAP . . . 158
5.8 Components of Spex . . . 160
5.9 Overview of Sparse-Blox's components . . . 170
5.10 Result space of different GA parameters . . . 175
5.11 Pareto optimal sparsity to accuracy solutions . . . 178
5.12 Amount of sparse blocks per block size . . . 180
5.13 Already present and created sparse blocks per ResNet-50 layer . . . 181

6.1 Overview of our mixed-precision tool . . . 190
6.2 Envisaged dataflow for mixed-precision inference . . . 192
6.3 Overview of BitBlade's PEs . . . 194
6.4 Overview of the Structured Dynamic Precision accelerator . . . 196
6.5 Components of our mixed-precision tool . . . 199
6.6 Overview of the dequantization flow for mixed-precision inference . 205
6.7 Overview of the quantization flow for mixed-precision inference . . 205
6.8 Pareto optimal precision to accuracy solutions . . . 211
6.9 Energy consumption improvements with our mixed-precision flow . 212
6.10 Bit-width distribution per layer in DeepLabV3 with MobileNetV2 . . 213
6.11 Model accuracy and precision solutions for different perturbations . 214

List of Tables

2.1 Parameter nomenclature for CONV layers used in this thesis 23
2.2 Overview of popular machine learning datasets 29
2.3 Overview of popular CNN models 31
2.4 Details on a selection of well-established CNNs 35

4.1 Comparison of analytical models' simulation time 134
4.2 Characteristics of the found eFPGA design 142
4.3 Comparison of cycle-accurate simulation frameworks for DNNs . . . 143
4.4 Comparison of analytical DNN performance models 145

5.1 Workload details used for evaluation with Spex and Sparse-Blox . . 173
5.2 State-of-the-art area overhead comparison of Sparse-Blox 182
5.3 Comparison of DNN accelerators for dynamic pruning 184

6.1 Workload details used for evaluation 208
6.2 Resource utilization of our quantization and dequantization architecture 217

Listings and Algorithms

2.1 Python-like pseudocode implementation of im2col and col2im. 47
2.2 Pseudocode implementation of different dataflows. 57

4.1 Example of a CPU configuration in FLECSim. 107
4.2 Exemplary accelerator configuration in FLECSim. 108
4.3 Memory configuration example in FLECSim. 109

5.1 Exemplary workload description file. 163
5.2 Pseudocode representation of Sparse-Blox's dataflow. 171

6.1 Pseudocode representation of the quantization unit. 206

Tim Hotfilter

Efficient Optimization of Convolutional Neural Networks for Modern Embedded High-Performance Applications

The advancement of AI and DNNs has been propelled by recent technological developments, enabling applications like autonomous vehicles and medical diagnostics. Moreover, in the last two decades tools to optimize DNN topologies and methods like quantization and pruning to enhance the efficiency were presented. Despite the progress, however, challenges for embedded systems remain in terms of performance, energy efficiency, safety, and accuracy.

This thesis presents three innovative approaches to increase efficiency of DNN accelerators. First, it introduces FLECSim-SoC, a cycle-accurate simulation tool and an analytical counterpart. Both assess the impact of architectural parameters on energy efficiency and performance. These tools complement each other, offering guidance to configure CNN accel-erators, as demonstrated in two case studies.

Additionally, a new hardware-centric pruning method to boost the energy efficiency of CNN accelerators is conceived, implemented and evaluated. It prunes regular regions in CNNs that match the accelerator's dimensions. A dedicated tool identifies those regions systematically, leading to 20% less operations and up to 19% energy savings, while occupying less chip area than other solutions.

In the context of robust and efficient CNN execution, this thesis explores mixed-precision quantization. A new hardware architecture is introduced to support mixed-precision with low latency. This technique enhances CNN robustness, particularly in challenging conditions like input perturbations. By dynamically increasing the operand's precision, the original model accuracy of could be restored, while reducing memory access energy consumption by 45%.